Guide définitif pour Yii 2.0

http://www.yiiframework.com/doc/guide

Qiang Xue,
Alexander Makarov,
Carsten Brandt,
Klimov Paul,
and
many contributors from the Yii community

Français translation provided by:
Benoît,
Kevin LEVRON,
José FOURNIER

This tutorial is released under the Terms of Yii Documentation.

Copyright 2014 Yii Software LLC. All Rights Reserved.

Table des matières

Chapitre 1

Introduction

1.1 Qu'est ce que Yii ?

Yii est un framework PHP hautes performances à base de composants qui permet de développer rapidement des applications Web modernes. Le nom Yii (prononcer `Yee` ou `[ji:]`) signifie "simple et évolutif" en Chinois. Il peut également être considéré comme un acronyme de **Yes It Is** !

1.1.1 Pour quel usage Yii est il optimal ?

Yii est un framework Web générique, c'est à dire qu'il peut être utilisé pour développer tous types d'applications Web basées sur PHP. De par son architecture à base de composants et son système de cache sophistiqué, il est particulièrement adapté au développement d'applications à forte audience telles que des portails, des forums, des systèmes de gestion de contenu (CMS), des sites e-commerce, des services Web RESTFul, etc.

1.1.2 Comment se positionne Yii vis-à-vis des autres Frameworks ?

— Comme la plupart des frameworks PHP, Yii est basé sur le modèle de conception MVC (Modèle-Vue-Contrôleur) et encourage à une organisation du code basée sur ce modèle.
— Yii repose sur l'idée que le code doit être écrit de façon simple et élégante. Il ne sera jamais question de compliquer le code de Yii uniquement pour respecter un modèle de conception.
— Yii est un framework complet offrant de nombreuses caractéristiques éprouvées et prêtes à l'emploi, telles que : constructeur de requêtes et ActiveRecord, à la fois pour les bases de données relationnelles et NoSQL ; prise en charge RESTful API ; prise en charge de caches multi-niveaux ; et plus.

— Yii est extrêmement flexible. Vous pouvez personnaliser ou remplacer presque chaque partie du code du noyau. Vous pouvez également profiter de son architecture extensible solide, afin d'utiliser ou développer des extensions distribuables.
— La haute performance est toujours un des principaux objectifs de Yii.

Yii n'est pas un one-man show, il est soutenu par une solide équipe de développement du noyau [1] ainsi que d'une grande communauté avec de nombreux professionnels qui contribuent constamment au développement de Yii. L'équipe de développeurs de Yii garde un œil attentif sur les dernières tendances en développement Web, et sur les meilleures pratiques et caractéristiques trouvées dans d'autres frameworks ou projets. Les meilleures pratiques et caractéristiques les plus pertinentes trouvées ailleurs sont régulièrement intégrées dans le code du noyau et utilisables via des interfaces simples et élégantes.

1.1.3 Versions de Yii

Yii est actuellement disponible en deux versions majeures : 1.1 et 2.0. La version 1.1, l'ancienne génération, est désormais en mode maintenance. La version 2.0 est une réécriture complète de Yii, adoptant les dernières technologies et protocoles, y compris Composer, PSR, les espaces de noms, les traits, et ainsi de suite. La version 2.0 est la dernière génération du framework et recevra nos principaux efforts de développement dans les prochaines années. Ce guide est principalement pour la version 2.0.

1.1.4 Configuration nécessaire

Yii 2.0 nécessite PHP 7.3.0 ou plus. Vous pouvez trouver plus de détails sur la configuration requise pour chaque fonctionnalité en utilisant le script de test de la configuration inclus dans chaque distribution de Yii.

Utiliser Yii requiert des connaissances de base sur la programmation objet (OOP), en effet Yii est un framework basé sur ce type de programmation. Yii 2.0 utilise aussi des fonctionnalités récentes de PHP, telles que les espaces de noms [2] et les traits [3]. Comprendre ces concepts vous aidera à mieux prendre en main Yii.

1.2 Mise à jour depuis la version 1.1

Il y a beaucoup de différences entre les versions 1.1 et 2.0 de Yii, le framework ayant été complètement réécrit pour la 2.0. En conséquence, la mise à jour depuis la version 1.1 n'est pas aussi triviale que la mise à jour

1. https://www.yiiframework.com/team
2. https://www.php.net/manual/fr/language.namespaces.php
3. https://www.php.net/manual/fr/language.oop5.traits.php

entre deux versions mineures. Dans ce guide, vous trouverez les principales différences entre les deux versions.

Si vous n'avez pas utilisé Yii 1.1 avant, vous pouvez ignorer cette section et passer directement à la section "Mise en route".

Merci de noter que Yii 2.0 introduit plus de nouvelles fonctionnalités que celles abordées ici. Il est fortement recommandé de lire tout le guide de référence pour en apprendre davantage. Il y a des chances que certaines fonctionnalités, que vous aviez préalablement développées pour vous, fassent désormais partie du code du noyau.

1.2.1 Installation

Yii 2.0 exploite pleinement Composer [4], le gestionnaire de paquet PHP. L'installation du framework, ainsi que des extensions, sont gérées par Composer. Reportez-vous à la section Installer Yii pour apprendre comment installer Yii 2.0. Si vous voulez créer de nouvelles extensions, ou rendre vos extensions existantes 1.1 compatibles 2.0, reportez-vous à la section Créer des extensions de ce guide.

1.2.2 Pré-requis PHP

Yii 2.0 requiert PHP 5.4 ou plus, ce qui est une grosse amélioration par rapport à PHP 5.2 qui était requis pour Yii 1.1.

Par conséquent, il y a beaucoup de différences au niveau du langage auxquelles vous devriez prêter attention. Voici un résumé des principaux changements concernant PHP :

— Espaces de noms [5].
— Fonctions anonymes [6].
— Syntaxe courte pour les tableaux : `[...éléments...]` est utilisé au lieu de `array(...éléments...)`.
— Syntaxe courte pour echo : `<?=` est utilisé dans les vues. Cela ne pose aucun problème à partir de PHP 5.4.
— Classes SPL et interfaces [7].
— Late Static Bindings (résolution statique à la volée) [8].
— Date et heure [9].
— Traits [10].

4. https://getcomposer.org/
5. https://www.php.net/manual/fr/language.namespaces.php
6. https://www.php.net/manual/fr/functions.anonymous.php
7. https://www.php.net/manual/fr/book.spl.php
8. https://www.php.net/manual/fr/language.oop5.late-static-bindings.php
9. https://www.php.net/manual/fr/book.datetime.php
10. https://www.php.net/manual/fr/language.oop5.traits.php

— intl[11]. Yii 2.0 utilise l'extension PHP `intl` pour les fonctionnalités d'internationalisation.

1.2.3 Espaces de noms

Le changement le plus évident dans Yii 2.0 est l'utilisation des espaces de noms. La majorité des classes du noyau utilise les espace de noms, par exemple, `yii\web\Request`. Le préfixe «C» n'est plus utilisé dans les noms de classe. Le schéma de nommage suit maintenant la structure des répertoires. Par exemple, `yii\web\Request` indique que le fichier de classe correspondant est `web/Request.php` dans le dossier du framework.

(Vous pouvez utiliser n'importe quelle classe du noyau sans inclure explicitement le fichier correspondant, grâce au chargeur de classe de Yii.)

1.2.4 Composants et objets

Yii 2.0 décompose la classe `CComponent` 1.1 en deux classes : `yii\base\BaseObject` et `yii\base\Component`. Le classe `BaseObject` est une classe de base légère qui permet de définir les Propriétés de l'objet via des accesseurs. La classe `Component` est une sous classe de `BaseObject` et prend en charge les [Événements](concept events.md) et les Comportements.

Si votre classe n'a pas besoin des événements et des comportements, vous devriez envisager d'utiliser `BaseObject` comme classe de base. C'est généralement le cas pour les classes qui représentent une structure de données basique.

1.2.5 Configuration d'objets

La classe `BaseObject` introduit une manière uniforme de configurer les objets. Toute sous-classe de `BaseObject` doit déclarer son constructeur (si besoin) de la manière suivante afin qu'elle puisse être configurée correctement :

```
class MyClass extends \yii\base\BaseObject
{
    public function __construct($param1, $param2, $config = [])
    {
        // ... initialisation avant que la configuration ne soit appliquée

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();
```

11. https://www.php.net/manual/fr/book.intl.php

```
        // ... initialisation après que la configuration est appliquée
    }
}
```

Dans ce qui précède, le dernier paramètre du constructeur doit être un tableau de configuration qui contient des entrées nom-valeur pour initialiser les propriétés à la fin du constructeur. Vous pouvez remplacer la méthode `init()` pour le travail d'initialisation qui doit être fait après que la configuration a été appliquée.

En suivant cette convention, vous serez en mesure de créer et de configurer de nouveaux objets en utilisant un tableau de configuration :

```
$object = Yii::createObject([
    'class' => 'MyClass',
    'property1' => 'abc',
    'property2' => 'cde',
], [$param1, $param2]);
```

Plus de détails sur les configurations peuvent être trouvés dans la section Configurations.

1.2.6 Événements

Avec Yii 1, les événements étaient créés par la définition d'une méthode `on` (par exemple `onBeforeSave`). Avec Yii 2, vous pouvez maintenant utiliser n'importe quel nom de l'événement. Vous déclenchez un événement en appelant la méthode `trigger()` :

```
$event = new \yii\base\Event;
$component->trigger($eventName, $event);
```

Pour attacher un gestionnaire à un événement, utilisez la méthode `on()` :

```
$component->on($eventName, $handler);
// Pour détacher le gestionnaire, utilisez :
// $component->off($eventName, $handler);
```

Il y a de nombreuses améliorations dans la gestion des événements. Pour plus de détails, reportez-vous à la section Événements.

1.2.7 Alias

Yii 2.0 étend l'utilisation des alias aux fichiers/répertoires et aux URL. Yii 2.0 impose maintenant aux alias de commencer par le caractère `@`, pour différencier les alias de fichiers/répertoires ou URL. Par exemple, l'alias `@yii` fait référence au répertoire d'installation de Yii. Les alias ? ?sont supportés dans la plupart du code de Yii. Par exemple, `yii\caching\FileCache::$cachePath` peut prendre à la fois un alias et un chemin de répertoire normal.

Un alias est aussi étroitement lié aux espaces de noms des classes. Il est recommandé de définir un alias pour chaque espace de noms racine, ce qui vous permet d'utiliser le chargeur automatique de classe de Yii sans sans devoir en faire d'avantage. Par exemple, vu que `@yii` fait référence au dossier d'installation de Yii, une classe comme `yii\web\Request` peut être chargée automatiquement. Si vous utilisez une librairie tierce, telle que Zend Framework, vous pouvez définir un alias de chemin `@Zend` qui fera référence au dossier d'installation de Zend Framework. Une fois que vous avez fait cela, Yii sera aussi en mesure de charger automatiquement une classe de ce framework.

Pour en savoir plus, consultez la section Alias.

1.2.8 Vues

Le changement le plus significatif à propos des vues dans Yii 2 est que la variable spéciale `$this` dans une vue ne fait plus référence au contrôleur ou à l'objet graphique. Au lieu de cela, `$this` correspond maintenant à un objet *vue*, un nouveau concept introduit dans la version 2.0. L'objet *vue* est de type `yii\web\View`, et représente la partie vue du modèle MVC. Si vous souhaitez accéder au contrôleur ou à l'objet graphique dans une vue, vous pouvez utiliser `$this->context`.

Pour afficher une vue depuis une autre vue, utilisez `$this->render()`, et non `$this->renderPartial()`. Le résultat retourné par la méthode `render()` doit être explicitement envoyé à la sortie, en effet `render()` retournera la vue au lieu de l'afficher. Par exemple :

```
echo $this->render('_item', ['item' => $item]);
```

Outre l'utilisation de PHP comme langage principal de gabarit, Yii 2.0 prend également en charge deux moteurs de gabarit populaires : Smarty et Twig. Le moteur de gabarit Prado n'est plus pris en charge. Pour utiliser ces moteurs de gabarit, vous devez configurer le composant `view` de l'application en définissant la propriété `View::$renderers`. Reportez-vous à la section Moteur de gabarit pour en savoir plus.

1.2.9 Modèles

Yii 2.0 utilise la classe `yii\base\Model` comme modèle de base, similaire à la classe `CModel` dans la version 1.1. La classe `CFormModel` a été supprimée. Vous pouvez, à la place, étendre la classe `yii\base\Model` afin de créer une classe modèle pour un formulaire.

Yii 2.0 introduit une nouvelle méthode appelée `scenarios()` pour déclarer les scénarios pris en charge, indiquer dans quel scénario un attribut doit être validé et si cet attribut peut être considéré comme sûr ou non, etc. Par exemple :

```
public function scenarios()
{
    return [
        'backend' => ['email', 'role'],
        'frontend' => ['email', '!name'],
    ];
}
```

Dans ce qui précède, deux scénarios sont déclarés : `backend` et `frontend`. Pour le scénario `backend` les attribut `email` et `role` sont sûrs et peuvent être affectés massivement. Pour le scénario `frontend`, `email` peut être affecté massivement tandis que `role` ne le peut pas. `email` et `rôle` doivent être validées en utilisant des règles.

La méthode `rules()` est toujours utilisée pour déclarer les règles de validation. Remarque : suite à l'introduction de la méthode `scenarios()`, le validateur `unsafe` n'as plus de raison d'être.

Dans la plupart des cas, vous n'avez pas besoin de surcharger la méthode `scenarios()` lorsque les scénarios existants sont déclarés via la méthode `rules()`, et il n'y a pas besoin de déclarer de propriétés `unsafe`.

Pour en savoir plus sur les modèles, reportez-vous à la section Modèles.

1.2.10 Contrôleurs

Yii 2.0 utilise la classe `yii\web\Controller` comme classe de base des contrôleurs, similaire à la classe `CController` dans la version Yii 1.1. `yii\base\Action` est la classe de base pour les actions.

L'impact le plus évident de ces changements sur votre code est qu'une action de contrôleur doit retourner le contenu que vous voulez afficher au lieu de l'envoyer vers la sortie :

```
public function actionView($id)
{
    $model = \app\models\Post::findOne($id);
    if ($model) {
        return $this->render('view', ['model' => $model]);
    } else {
        throw new \yii\web\NotFoundHttpException;
    }
}
```

Reportez-vous à la section Contrôleurs pour plus de détails.

1.2.11 Objets graphiques

Yii 2.0 utilise la classe `yii\base\Widget` comme classe de base pour les objets graphiques, similaire à la classe `CWidget` de Yii 1.1.

Pour avoir une meilleure prise en charge du framework dans les EDI, Yii 2 introduit une nouvelle syntaxe pour utiliser les objets graphiques. Les

méthodes statiques `begin()`, `end()`, et `widget()` ont été créées et sont utilisables comme suit :

```
use yii\widgets\Menu;
use yii\widgets\ActiveForm;

// Remarque : vous devez utiliser echo pour afficher le résultat
echo Menu::widget(['items' => $items]);

// Utilisation d'un tableau pour initialiser les propriétés de l'objet
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge']],
]);
... champs du formulaire ici ...
ActiveForm::end();
```

Reportez-vous à la section Objets graphiques pour en savoir plus.

1.2.12 Thèmes

Les thèmes fonctionnent tout à fait différemment dans la version 2.0. Ils sont maintenant basés sur un mécanisme de mise en correspondance de chemin qui met un chemin de fichier de vue en correspondance avec un chemin de fichier de vue thématisée. Par exemple, si la mise en correspondance pour un thème est `['/web/views' => '/web/themes/basic']`, alors la version thématisée du fichier de vue `/web/views/site/index.php` sera `/web/themes/basic/site/index.php`. Pour cette raison, les thèmes peuvent maintenant être appliqués à n'importe quel fichier de vue, même une vue utilisée en dehors du contexte d'un contrôleur ou d'un objet graphique.

En outre, il n'y a plus de composant `CThemeManager`. A la place, `theme` est une propriété configurable du composant `view` de l'application.

Reportez-vous à la section Thématisation pour plus de détails.

1.2.13 Applications en ligne de commande

Les applications en ligne de commande (console) sont désormais organisées en contrôleurs, comme les applications Web. ces contrôleurs doivent étendre la classe `yii\console\Controller`, similaire à la classe `CConsoleCommand` de la version 1.1.

Pour exécuter une commande console, utilisez `yii <route>`, où `<route>` correspond à une route vers un contrôleur (par exemple `sitemap/index`). Les arguments anonymes supplémentaires sont passés comme paramètres à l'action du contrôleur correspondant, alors que les arguments nommés sont analysés selon les options déclarées dans la méthode `yii\console\Controller::options()`.

Yii 2.0 prend en charge la génération automatique d'aide à partir des blocs de commentaire.

Reportez-vous à la section Commandes console pour plus de détails.

1.2.14 I18N

Yii 2.0 supprime les fonctionnalités internes de formatage des dates et des nombres, en faveur du module PHP PECL intl [12].

La traduction des messages est désormais effectuée via le composant d'application `i18n`. Ce composant gère un ensemble de sources de messages, ce qui vous permet d'utiliser différentes sources de messages en fonction de catégories.

Reportez-vous à la section Internationalisation pour plus de détails.

1.2.15 Filtres d'action

Les filtres d'action sont maintenant implémentés comme des comportements. Pour définir un nouveau filtre personnalisé, étendez la classe `yii\base\ActionFilter`. Pour utiliser un filtre, déclarez le comme un comportement du contrôleur. Par exemple, pour utiliser le filtre `yii\filters\AccessControl`, vous aurez le code suivant dans le contrôleur :

```
public function behaviors()
{
    return [
        'access' => [
            'class' => 'yii\filters\AccessControl',
            'rules' => [
                ['allow' => true, 'actions' => ['admin'], 'roles' => ['@']],
            ],
        ],
    ];
}
```

Reportez-vous à la section Filtres pour plus de détails.

1.2.16 Ressources

Yii 2.0 introduit un nouveau concept de paquet de ressources (*asset bundle*) qui remplace le concept de gestionnaire de ressources (*asset manager*) de la version 1.1.

Un paquet de ressources est une collection de fichiers (par exemple : fichier JavaScript, CSS, image, etc.) dans un dossier. Chaque paquet est représenté par une classe étendant `yii\web\AssetBundle`. En *enregistrant* un paquet de ressources via `yii\web\AssetBundle::register()`, vous rendez les ressources du paquet accessibles via le Web. Contrairement à Yii 1.1, la page *enregistrant* le paquet contiendra automatiquement les références vers les fichiers déclarés dans le paquet.

Reportez-vous à la section Assets pour plus de détails.

12. https://pecl.php.net/package/intl

1.2.17 Assistants

Yii 2.0 introduit de nombreux assistants couramment utilisés, sous la forme de classes statiques, y compris :

- `yii\helpers\Html`
- `yii\helpers\ArrayHelper`
- `yii\helpers\StringHelper`
- `yii\helpers\FileHelper`
- `yii\helpers\Json`

Reportez-vous à la section Assistants pour plus de détails.

1.2.18 Formulaires

Yii 2.0 introduit le concept de *champ* pour la construction d'un formulaire à l'aide de la classe `yii\widgets\ActiveForm`. Un champ est un conteneur constitué d'une étiquette, d'une entrée, d'un message d'erreur, et/ou d'un texte d'aide. Un champ est représenté comme un objet de la classe `ActiveField`. En utilisant des champs, vous pouvez construire un formulaire plus proprement qu'avant :

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <div class="form-group">
        <?= Html::submitButton('Login') ?>
    </div>
<?php yii\widgets\ActiveForm::end(); ?>
```

Reportez-vous à la section Créer des formulaires pour plus de détails.

1.2.19 Constructeur de requêtes

Dans la version 1.1, la construction des requêtes était dispersée dans plusieurs classes, y compris `CDbCommand`, `CDbCriteria` et `CDbCommandBuilder`. Avec Yii 2.0, une requête de base de données est représentée par un objet de la classe `Query` qui peut être transformé en une instruction SQL à l'aide de la classe `QueryBuilder`. Par exemple :

```
$query = new \yii\db\Query();
$query->select('id, name')
      ->from('user')
      ->limit(10);

$command = $query->createCommand();
$sql = $command->sql;
$rows = $command->queryAll();
```

De plus, ces méthodes de construction de requête peuvent également être utilisées lorsque vous travaillez avec Active Record.

Reportez-vous à la section Constructeur de requête pour plus de détails.

1.2.20 Active Record

Yii 2.0 introduit beaucoup de modifications au modèle Active Record. Les deux plus évidentes concernent la construction des requêtes et la manipulation de requêtes relationnelles.

La classe `CDbCriteria` en 1.1 est remplacée par `yii\db\ActiveQuery` dans Yii 2. Cette classe étend `yii\db\Query`, et hérite donc de toutes les méthodes de construction de requête. Pour commencer à construire une requête il suffit d'utiliser `yii\db\ActiveRecord::find()` :

```
// Pour récupérer tous les clients *actifs* et les trier selon leur
identifiant
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

Pour déclarer une relation, il suffit de définir un accesseur qui renvoie un objet `ActiveQuery`. Le nom de la propriété définie par l'accesseur représente le nom de la relation. Par exemple, le code suivant déclare une relation `orders` (en 1.1, vous aviez à déclarer les relations dans la méthode `relations()`) :

```
class Customer extends \yii\db\ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany('Order', ['customer_id' => 'id']);
    }
}
```

Maintenant vous pouvez utiliser `$customer->orders` pour accéder aux commandes du client depuis la table liée. Vous pouvez aussi utiliser le code suivant pour effectuer une requête relationnelle à la volée avec une condition personnalisée :

```
$orders = $customer->getOrders()->andWhere('status=1')->all();
```

Lors du chargement anticipé (*eager loading*) d'une relation, Yii 2.0 fonctionne différemment de la version 1.1. En particulier avec Yii 1.1, une jointure était créée pour sélectionner à la fois l'enregistrement principal et les enregistrements liés. Avec Yii 2.0, deux instructions SQL sont exécutées sans utiliser de jointure : la première récupère les enregistrements principaux et la seconde récupère les enregistrements liés en filtrant selon les clés primaires des enregistrements principaux.

Au lieu de retourner des objets `ActiveRecord`, vous pouvez utiliser la méthode `asArray()` lors de la construction d'une requête pour renvoyer un grand nombre d'enregistrements. Ainsi le résultat sera retourné sous forme de tableaux, ce qui peut réduire considérablement le temps de calcul et la

mémoire nécessaires dans le cas d'un grand nombre d'enregistrements. Par exemple :

```
$customers = Customer::find()->asArray()->all();
```

Un autre changement fait que vous ne pouvez plus définir les valeurs par défaut des attributs en utilisant des propriétés publiques. Si vous en avez besoin, vous devez utiliser la méthode `init` de la classe de votre modèle.

```
public function init()
{
    parent::init();
    $this->status = self::STATUS_NEW;
}
```

Il y avait des problèmes de surcharge du constructeur de la classe ActiveRecord 1.1. Ces problèmes n'existent plus dans la version 2.0. Notez que lorsque vous ajoutez des paramètres au constructeur, vous avez éventuellement à surcharger la méthode `yii\db\ActiveRecord::instantiate()`.

Il y a beaucoup d'autres modifications et améliorations à Active Record. Reportez-vous à la section Active Record pour en savoir plus.

1.2.21 Comportement des Enregistrements actifs)

Dans la version 2.0, nous avons la classe de base des *behaviors* (comportements) `CActiveRecordBehavior`. Si vous voulez créer une classe de comportement d'enregistrement actif (Active Record), vous devez étendre directement la classe `yii\base\Behavior`. Si la classe de comportement doit réagir à certains événements du propriétaire, vous devez redéfinir les méthodes `events()` comme suit :

```
namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
        // ...
    }
}
```

1.2.22 User et IdentityInterface

La classe `CWebUser` 1.1 est maintenant remplacée par `yii\web\User`, et il n'y a plus de classe `CUserIdentity`. Au lieu de cela, vous devez implémenter `yii\web\IdentityInterface` qui est beaucoup plus simple à utiliser. Le modèle de projet avancé fournit un exemple.

Reportez-vous aux sections Authentification, Authorisation, et Modèle de projet avancé [13] pour en savoir plus.

1.2.23 Gestion des URL

La gestion des URL dans Yii 2 est similaire à celle disponible dans la version 1.1. Une amélioration majeure est que la gestion des URL prend désormais en charge les paramètres optionnels. Par exemple, si vous avez une règle déclarée comme suit, cela fera correspondre `post/popular` et `post/1/popular`. Dans la version 1.1, il fallait utiliser deux règles pour atteindre le même objectif.

```
[
    'pattern' => 'post/<page:\d+>/<tag>',
    'route' => 'post/index',
    'defaults' => ['page' => 1],
]
```

Reportez-vous à la section Documentation de la gestion des URL pour en savoir plus.

Un changement important dans la convention de nommage pour les routes est que les noms utilisant la *casse en dos de chameau* des contrôleurs et des actions utilisent désormais uniquement des mots en bas de casse séparés par des tirets, p. ex. l'identifiant du contrôleur Reportez-vous à la section traitant des Identifiants de contrôleur et Identifiants d'action pour plus de détails.

1.2.24 Utiliser Yii 1.1 et 2.x ensemble

Si vous avez du code Yii 1.1 que vous souhaitez réutiliser avec Yii 2, reportez-vous à la section Utiliser Yii 1.1 et 2.0 ensemble.

13. https://www.yiiframework.com/extension/yiisoft/yii2-app-advanced/doc/guide

Chapitre 2

Mise en Route

2.1 Installer Yii

Vous pouvez installer Yii de deux façons, en utilisant le gestionnaire de paquets Composer [1] ou en téléchargeant une archive. La première méthode est conseillée, étant donné qu'elle permet d'installer de nouvelles extensions ou de mettre Yii à jour en exécutant simplement une commande.

Les installations standard de Yii provoquent le téléchargement et l'installation d'un modèle de projet. Un modèle de projet et un projet Yii fonctionnel qui met en œuvre quelques fonctionnalités de base, telles que la connexion, le formulaire de contact, etc. Son code est organisé de la façon recommandée. En conséquence, c'est un bon point de départ pour vos propres projets.

Dans cette section et quelques-unes de ses suivantes, nous décrirons comment installer Yii avec le modèle baptisé *Basic Project Template* (modèle de projet de base) et comment mettre en œuvre de nouvelles fonctionnalités sur cette base. Yii vous offre également un autre modèle de projet appelé Advanced Project Template [2] (modèle de projet avancé) qui convient mieux à un environnement de développement en équipe impliquant des tiers multiples.

> **Note :** le modèle de projet de base conviendra à 90 pourcent des application Web. Il diffère du modèle de projet avancé essentiellement sur la manière dont le code est organisé. Si vous débutez avec Yii, nous vous conseillons fortement de vous en tenir au modèle de projet de base pour sa simplicité tout en disposant des fonctionnalités suffisantes.

1. https://getcomposer.org/
2. https://www.yiiframework.com/extension/yiisoft/yii2-app-advanced/doc/guide

2.1.1 Installer via Composer

Installer Composer

Si vous n'avez pas déjà installé Composer, vous pouvez le faire en suivant les instructions du site getcomposer.org [3]. Sous Linux et Mac OS X, vous pouvez exécuter les commandes :

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

Sous Windows, téléchargez et exécutez Composer-Setup.exe [4].

En cas de problèmes, consultez la section Troubleshooting (résolution des problèmes) de la documentation de Composer [5],

Si vous débutez avec Composer, nous vous recommandons au minimum la lecture de la section Basic usage (utilisation de base) [6] de la documentation de Composer

Dans ce guide, toutes les commandes de Composer suppose que vous avez installé Composer globalement [7] et qu'il est disponible par la commande `composer`. Si, au lieu de cela, vous utilisez `composer.phar` depuis un dossier local, vous devez adapter les exemples fournis en conséquence.

Si Composer était déjà installé auparavant, assurez-vous d'utiliser une version à jour. Vous pouvez mettre Composer à jour avec la commande `composer self-update`.

> **Note :** durant l'installation de Yii, Composer aura besoin d'obtenir de nombreuses informations de l'API de Github. Le nombre de requêtes dépend du nombre de dépendances de votre application et peut excéder la **Github API rate limit**. Si vous arrivez à cette limite, Composer peut vous demander vos identifiants de connexion pour obtenir un jeton d'accès à l'API de Github. Avec une connexion rapide, vous pouvez atteindre cette limite plus vite que Composer n'est capable de gérer. C'est pourquoi, nous vous recommandons de configurer ce jeton d'accès avant d'installer Yii. Reportez-vous à la documentation de Composer sur les jetons de l'API Github [8] pour savoir comment procéder.

Installer Yii

Avec Composer installé, vous pouvez installer le modèle de projet Yii en exécutant la commande suivante dans un dossier accessible via le Web :

3. https://getcomposer.org/download/
4. https://getcomposer.org/Composer-Setup.exe
5. https://getcomposer.org/doc/articles/troubleshooting.md
6. https://getcomposer.org/doc/01-basic-usage.md
7. https://getcomposer.org/doc/00-intro.md#globally
8. https://getcomposer.org/doc/articles/troubleshooting.md#api-rate-limit-and-oauth-tokens

```
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

Cette commande installera la dernière version stable du modèle de projet Yii dans le dossier `basic`. Vous êtes libre de choisir un autre dossier si vous le désirez.

> **Note :** si la commande `composer create-project` échoue, reportez-vous à la section Troubleshooting (résolution des problèmes) de la documentation de Composer [9] pour les erreurs communes. Une fois l'erreur corrigée, vous pouvez reprendre l'installation avortée en exécutant `composer update` dans le dossier `basic` (ou celui que vous aviez choisi).
>
> **Conseil :** si vous souhaitez installer la dernière version de développement de Yii, vous pouvez utiliser la commande suivante qui ajoutera l'option stability [10] :
>
> >`bash >composer create-project –prefer-dist –stability=dev yiisoft/yii2-app-basic basic >`
>
> Notez que la version de développement de Yii ne doit pas être utilisée en production, vu qu'elle pourrait *casser* votre code existant.

2.1.2 Installer depuis une archive

Installer Yii depuis une archive se fait en trois étapes :

1. Télécharger l'archive sur le site yiiframework.com [11].
2. Décompresser l'archive dans un dossier accessible via le Web.
3. Modifier le fichier `config/web.php` en entrant une clé secrète pour la configuration de `cookieValidationKey` (cela est fait automatiquement si vous installez Yii avec Composer) :

`php // !!! insert a secret key in the following (if it is empty) - this is required by cookie validation 'cookieValidationKey' => 'enter your secret key here',
`

2.1.3 Autres options d'installation

Les instructions d'installation ci-dessus montrent comment installer Yii, ce qui installe également une application Web de base qui fonctionne *out of the box* (sans configuration supplémentaire). Cette approche est un bon point de départ pour les petits projets, en particulier si vous débutez avec Yii.

Mais il y a d'autres options d'installation disponibles :

9. https://getcomposer.org/doc/articles/troubleshooting.md
10. https://getcomposer.org/doc/04-schema.md#minimum-stability
11. https://www.yiiframework.com/download/

— Si vous voulez installer uniquement le framework et que vous souhaitez créer une application à partir de zéro, vous pouvez suivre les instructions dans la partie Créer votre propre structure d'application.
— Si vous voulez commencer par une application plus sophistiquée, mieux adaptée aux environnements d'équipe de développement, vous pouvez envisager l'installation du Modèle d'application avancée [12].

2.1.4 Installer les Assets (ici bibliothèques CSS et JavaScript)

Yii s'appuie sur les paquets Bower [13] et/ou NPM [14] pour l'installation des bibliothèques CSS et JavaScript.

Il utilise Composer pour les obtenir, permettant ainsi aux versions de paquet de PHP et à celles de CSS/JavaScript, d'être résolues en même temps. Cela peut être obtenue soit en utilisant asset-packagist.org [15] ou composer asset plugin [16].

Reportez-vous à la documentation sur les Assets pour plus de détail.

Vous pouvez souhaiter gérer vos « assets », soit via le client natif Bower/NPM, soit via CDN, soit éviter totalement leur installation. Afin d'empêcher l'installation des « assets » via Composer, ajoutez les lignes suivantes à votre fichier 'composer.json' :

```
"replace": {
    "bower-asset/jquery": ">=1.11.0",
    "bower-asset/inputmask": ">=3.2.0",
    "bower-asset/punycode": ">=1.3.0",
    "bower-asset/yii2-pjax": ">=2.0.0"
},
```

> **Note :** en cas de neutralisation de l'installation des « assets » via Composer, c'est à vous d'en assurer l'installation et de résoudre les problèmes de collision de versions. Attendez-vous à des incohérences possibles parmi les fichiers d'assets issus de vos différentes extensions.

2.1.5 Vérifier l'installation

Après l'installation, vous pouvez, soit configurer votre serveur Web (voir section suivante), soit utiliser le serveur PHP web incorporé [17] en utilisant la commande en console suivante depuis le dossier racine de votre projet :

12. https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md
13. https://bower.io/
14. https://www.npmjs.com/
15. https://asset-packagist.org
16. https://github.com/fxpio/composer-asset-plugin
17. https://www.php.net/manual/fr/features.commandline.webserver.php

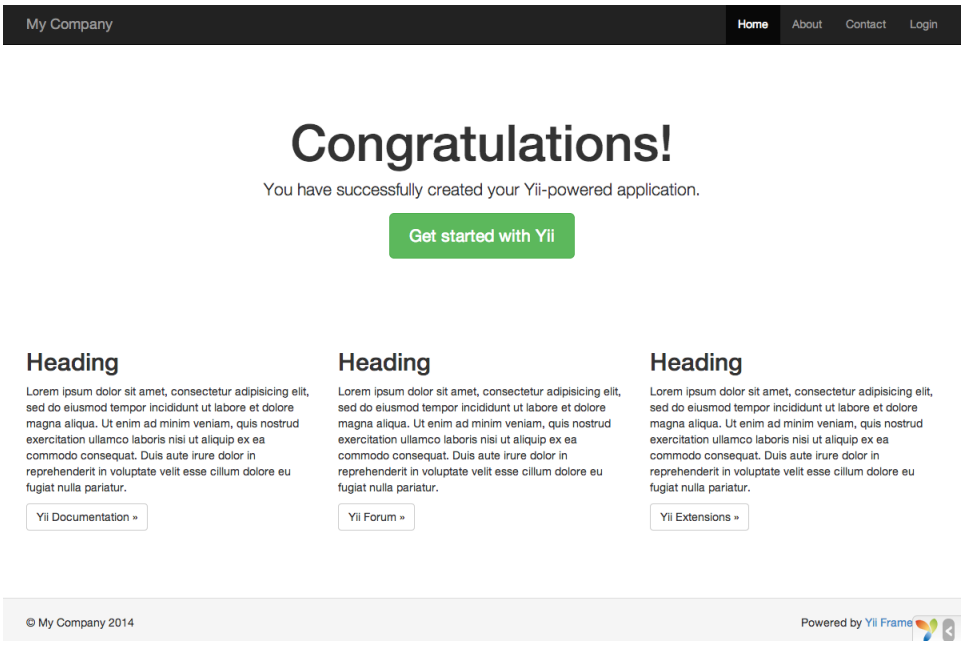
```
php yii serve
```

> **Note :** par défaut le serveur HTTP écoute le port 8080. Néanmoins, si ce port est déjà utilisé ou si vous voulez servir plusieurs applications de cette manière, vous pouvez spécifier le port à utiliser en ajoutant l'argument –port à la commande :

```
php yii serve --port=8888
```

Pour accéder à l'application Yii pointez votre navigateur sur l'URL suivante :

```
http://localhost:8080/
```



Vous devriez voir dans votre navigateur la page ci-dessus. Sinon, merci de vérifier que votre installation remplit bien les pré-requis de Yii. Vous pouvez vérifier cela en utilisant l'une des approches suivantes :

— Utilisez un navigateur pour accéder à l'URL `http://localhost/basic/requirements.php`
— Exécutez les commandes suivantes :

```
cd basic
php requirements.php
```

Vous devez configurer votre installation de PHP afin qu'elle réponde aux exigences minimales de Yii. Le plus important étant que vous ayez PHP 5.4 ou plus, idéalement PHP 7. Si votre application a besoin d'une base de données, vous devez également installer l'extension PHP PDO [18] ainsi qu'un pilote correspondant à votre système de base de données (par exemple `pdo_mysql` pour MySQL).

18. https://www.php.net/manual/fr/pdo.installation.php

2.1.6 Configuration du serveur Web

> **Note :** si vous voulez juste tester Yii sans intention de l'utiliser sur un serveur de production, vous pouvez ignorer ce paragraphe.

L'application installée selon les instructions ci-dessus devrait fonctionner *out of the box* (sans configuration supplémentaire) avec le serveur HTTP Apache [19] ou le serveur HTTP Nginx [20], sous Windows, Mac OX X, ou Linux avec PHP 5.4 ou plus récent. Yii 2.0 est aussi compatible avec HHVM [21] de Facebook. Cependant, il existe des cas marginaux pour lesquels HHVM se comporte différemment du PHP natif ; c'est pourquoi vous devez faire plus attention en utilisant HHVM..

Sur un serveur de production, vous pouvez configurer votre serveur Web afin que l'application soit accessible via l'URL `https://www.example.com/index.php` au lieu de `https://www.example.com/basic/web/index.php`. Cela implique que le dossier racine de votre serveur Web pointe vers le dossier `basic/web`. Vous pouvez également cacher `index.php` dans l'URL, comme décrit dans la partie Génération et traitement des URL, vous y apprendrez comment configurer votre serveur Apache ou Nginx pour atteindre ces objectifs.

> **Note :** en utilisant `basic/web` comme dossier racine, vous empêchez également aux utilisateurs finaux d'accéder à votre code d'application privé et fichiers de données sensibles qui sont stockés dans le dossier `basic`. Refuser l'accès à ces ressources est une amélioration de la sécurité.
>
> **Note :** si votre application s'exécute dans un environnement d'hébergement mutualisé où vous n'avez pas la permission de modifier la configuration du serveur Web, vous pouvez ajuster la structure de votre application pour une meilleure sécurité. Reportez-vous à la partie Environnement d'hébergement mutualisé pour en savoir plus.
>
> **Note :** si vous exécutez votre application Yii derrière un mandataire inverse, vous pourriez avoir besoin de configurer les mandataires de confiance et entêtes dans le composant « request ».

Configuration Apache recommandée

Utilisez la configuration suivante dans le fichier `httpd.conf`, ou dans la configuration de votre hôte virtuel. Notez que vous devez remplacer `path/to/basic/web` par le chemin vers le dossier `basic/web`.

```
# Configuration du dossier racine
DocumentRoot "path/to/basic/web"
```

19. https://httpd.apache.org/
20. https://nginx.org/
21. https://hhvm.com/

```
<Directory "path/to/basic/web">
    # utiliser mod_rewrite pour la prise en charge des URL élégantes
    ("pretty URL")
    RewriteEngine on

    # Si le dossier ou fichier existe, répondre directement
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    # Sinon on redirige vers index.php
    RewriteRule . index.php

    # si $showScriptName est à "false" dans UrlManager, ne pas autoriser
    l'accès aux URL incluant le nom du script
    RewriteRule ^index.php/ - [L,R=404]

    # ...other settings...
</Directory>
```

Configuration Nginx recommandée

Pour utiliser Nginx, vous devez avoir installé PHP en utilisant FPM SAPI [22]. Utilisez la configuration Nginx suivante, en remplaçant `path/to/basic/web` par le chemin vers le dossier `basic/web` et `mysite.test` par le nom d'hôte de votre serveur.

```
server {
    charset utf-8;
    client_max_body_size 128M;

    listen 80; ## listen for ipv4
    #listen [::]:80 default_server ipv6only=on; ## listen for ipv6

    server_name mysite.test;
    root        /path/to/basic/web;
    index       index.php;

    access_log  /path/to/basic/log/access.log;
    error_log   /path/to/basic/log/error.log;

    location / {
        # Rediriger tout ce qui n'est pas un fichier réel index.php
        try_files $uri $uri/ /index.php$is_args$args;
    }

    # enlevez les commentaires de ces lignes pour évitez que Yii ne gère les
    requêtes vers des fichiers statiques inexistants
    #location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.html;
```

22. https://www.php.net/manual/fr/install.fpm.php

```
    # refuser l'accès aux fichiers php pour le dossier /assets
    location ~ ^/assets/.*\.php$ {
        deny all;
    }

    location ~ \.php$ {
        include fastcgi.conf;
        fastcgi_pass   127.0.0.1:9000;
        #fastcgi_pass unix:/var/run/php5-fpm.sock;
    }

    location ~ /\.(ht|svn|git) {
        deny all;
    }
}
```

Lorsque vous utilisez cette configuration, vous devez aussi mettre l'option `cgi.fix_pathinfo=0` dans le fichier `php.ini` afin d'éviter de nombreux appels système à `stat()`.

Notez également que lors de l'utilisation d'un serveur HTTPS, vous devez ajouter l'option `fastcgi_param HTTPS on;` afin que Yii puisse détecter correctement si une connexion est sécurisée.

2.2 Fonctionnement des applications

Après avoir installé Yii, vous obtenez une application Yii fonctionnelle accessible via l'URL `https://hostname/basic/web/index.php` ou `https://hostname/index.php`, en fonction de votre configuration. Cette section vous initiera aux fonctionnalités intégrées à l'application, à la manière dont le code est organisé et à la gestion des requêtes par l'application.

> **Info :** pour simplifier, au long de ce tutoriel de démarrage, nous supposerons que `basic/web` est la racine de votre serveur Web, et que vous avez configuré l'URL pour accéder à votre application comme suit ou de façon similaire : `https://hostname/index.php`. Pour vos besoins, merci d'ajuster les URLs dans notre description comme il convient.

Notez que contrairement au framework lui-même, après avoir installé un modèle de projet, vous êtes entièrement libre d'en disposer. Vous êtes libre d'ajouter ou de supprimer du code selon vos besoins.

2.2.1 Fonctionnalité

L'application basique installée contient quatre pages :

— La page d'accueil, affichée quand vous accédez à l'URL `https://hostname/index.php`,
— la page "About" (À Propos),

— la page “Contact”, qui présente un formulaire de contact permettant aux utilisateurs finaux de vous contacter par courriel,
— et la page “Login” (Connexion), qui présente un formulaire de connexion qui peut être utilisé pour authentifier des utilisateurs finaux. Essayez de vous connecter avec “admin/admin”, et vous verrez l'élément “Login” du menu principal être remplacé par “Logout” (Déconnexion).

Ces pages ont en commun une entête et un pied de page. L'entête contient une barre de menu principal qui permet la navigation entre les différentes pages.

Vous devriez également voir une barre d'outils en bas de votre fenêtre de navigation. C'est un outil de débogage [23] utile fourni par Yii pour enregistrer et afficher de nombreuses informations de débogage, telles que des messages de journaux, les statuts de réponses, les requêtes lancées vers la base de données, etc.

En plus de l'application Web, il existe,dans le dossier de base de l'application, un script en console appelé `yii`. Ce script peut être utilisé pour exécuter des tâches de fond et de maintenance pour l'application ; ces tâches sont décrites à la section Applications en console.

2.2.2 Structure de l'application

Les répertoires et fichiers les plus importants de votre application sont (en supposant que le répertoire racine de l'application est `basic`) :

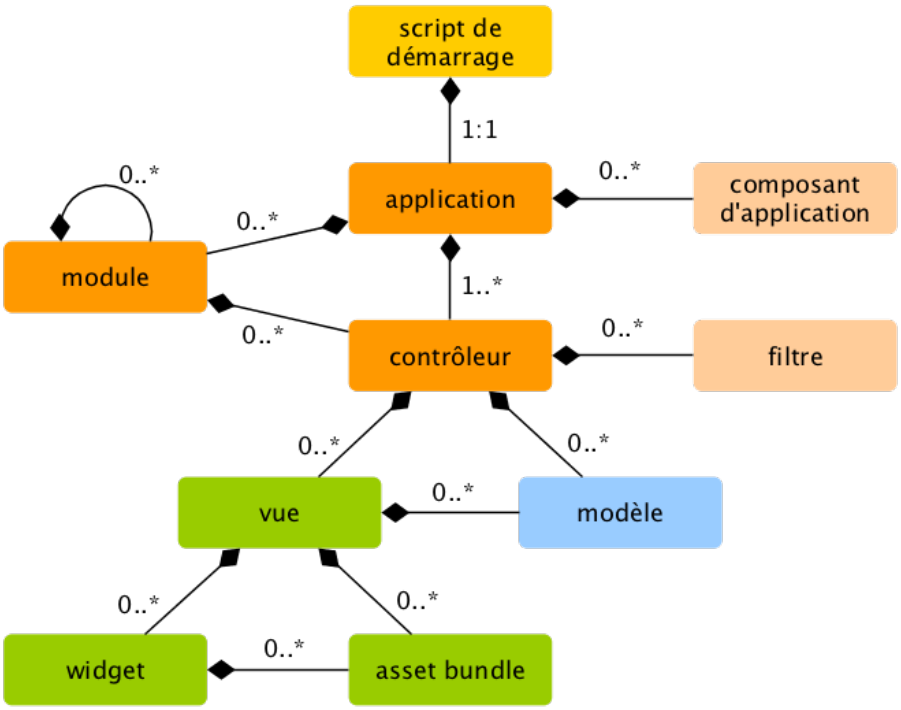
```
basic/                  chemin de base de l'application
    composer.json       utilisé par Composer, décrit les information de
    paquets
    config/             contient les configurations de l'application et
    autres
        console.php     configuration de l'application console
        web.php         configuration de l'application Web
    commands/           contient les classes de commandes console
    controllers/        contient les classes de contrôleurs
    models/             contient les classes de modèles
    runtime/            contient les fichiers générés par Yii au cours de
    l'exécution, tels que les fichiers de logs ou de cache and cache
    vendor/             contient les paquets Composer installés, y compris
    le framework Yii
    views/              contient les fichiers de vues
    web/                racine Web de l'application, contient les fichiers
    accessibles via le Web
        assets/         contient les fichiers assets (javascript et css)
        publiés par Yii
        index.php       le script de démarrage (ou bootstrap) pour
        l'application
    yii                 le script d'exécution de Yii en commande console
```

23. https://github.com/yiisoft/yii2-debug/blob/master/docs/guide/README.md

Dans l'ensemble, les fichiers de l'application peuvent être séparés en deux types : ceux situés dans `basic/web` et ceux situés dans d'autres répertoires. Les premiers peuvent être atteints directement en HTTP (c'est à dire dans un navigateur), tandis que les seconds ne peuvent et ne doivent pas l'être.

Yii est mis en œuvre selon le modèle de conception modèle-vue-contrôleur (MVC) [24], ce qui se reflète dans l'organisation des répertoires ci-dessus. Le répertoire `models` contient toutes les classes modèles, le répertoire `views` contient tous les scripts de vue, et le répertoire `controllers` contient toutes les classes contrôleurs.

Le schéma suivant présente la structure statique d'une application.

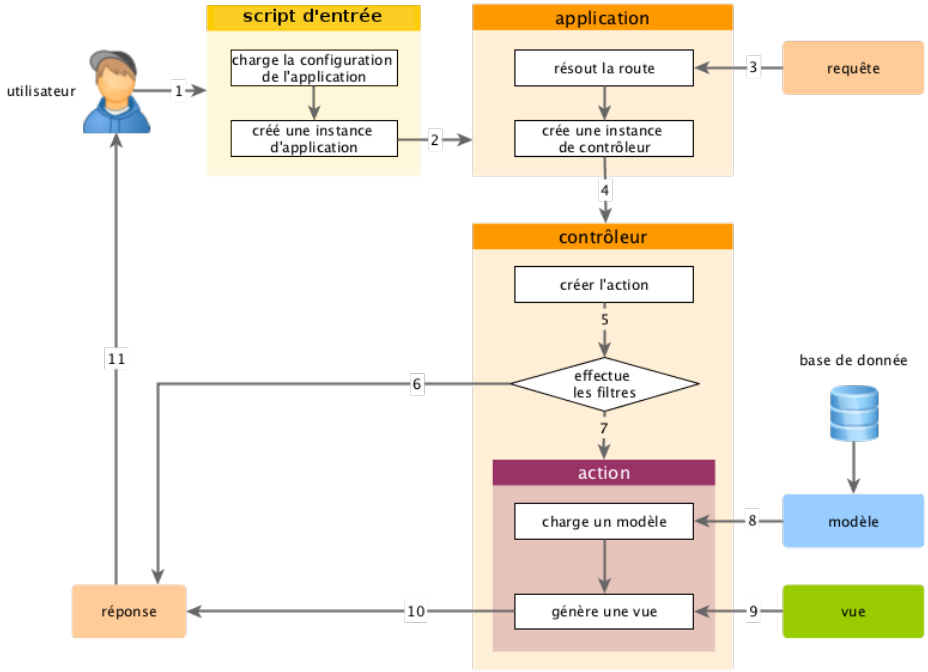


Chaque application dispose d'un script de démarrage `web/index.php` qui est le seul script PHP de l'application accessible depuis le Web. Le script de démarrage reçoit une requête entrante et crée une instance d'application pour la traiter. L'application résout la requête avec l'aide de ses composants, et distribue la requête aux éléments MVC. Les composants graphiques (widgets) sont utilisés dans les vues pour aider à créer des éléments d'interface complexes et dynamiques.

24. https://wikipedia.org/wiki/Model-view-controller

2.2.3 Cycle de vie d'une requête

Le diagramme suivant présente la manière dont une application traite une requête.



1. Un utilisateur fait une requête au script de démarrage `web/index.php`.
2. Le script de démarrage charge la configuration de l'application et crée une instance d'application pour traiter la requête.
3. L'application résout la route requise avec l'aide du composant d'application requête.
4. L'application créé une instance de contrôleur pour traiter la requête.
5. Le contrôleur crée une instance d'action et applique les filtres pour l'action.
6. Si un filtre échoue, l'action est annulée.
7. Si tous les filtres sont validés, l'action est exécutée.
8. L'action charge un modèle de données, potentiellement depuis une base de données.
9. L'action génère une vue, lui fournissant le modèle de données.
10. Le résultat généré est renvoyé au composant d'application response.
11. Le composant « response » (réponse) envoie le résultat généré au navigateur de l'utilisateur.

2.3 Hello World

Cette section decrit la méthode pour créer une nouvelle page "Hello" dans votre application. Pour ce faire, vous allez créer une action et une vue :

— L'application distribuera la requête à l'action
— et à son tour, l'action générera un rendu de la vue qui affiche le mot "Hello" à l'utilisateur.

A travers ce tutoriel, vous apprendrez trois choses :

1. Comment créer une action pour répondre aux requêtes,
2. comment créer une vue pour composer le contenu de la réponse, et
3. comment une application distribue les requêtes aux actions.

2.3.1 Créer une Action

Pour la tâche "Hello", vous allez créer une action `dire` qui reçoit un paramètre `message` de la requête et affiche ce message à l'utilisateur. Si la requête ne fournit pas de paramètre `message`, l'action affichera le message par défaut "Hello".

> **Info :** Les actions sont les objets auxquels les utilisateurs peuvent directement se référer pour les exécuter. Les actions sont groupées par contrôleurs. Le résultat de l'exécution d'une action est la réponse que l'utilisateur recevra.

Les actions doivent être déclarées dans des contrôleurs. Par simplicité, vous pouvez déclarer l'action `dire` dans le contrôleur existant `SiteController`. Ce contrôleur est défini dans le fichier classe `controllers/SiteController.php`. Voici le début de la nouvelle action :

```
<?php

namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    // ...code existant...

    public function actionDire($message = 'Hello')
    {
        return $this->render('dire', ['message' => $message]);
    }
}
```

Dans le code ci-dessous, l'action `dire` est définie par une méthode nommée `actionDire` dans la classe `SiteController`. Yii utilise le préfixe `action` pour faire la différence entre des méthodes actions et des méthodes non-actions dans

une classe contrôleur. Le nom suivant le préfixe `action` est associé à l'ID de l'action.

Quand vous choisissez le nom de vos actions, gardez à l'esprit comment Yii traite les IDs d'action. Les références aux IDs d'actions sont toujours effectuées en minuscules. Si un ID d'action nécessite plusieurs mots, ils seront concaténés à l'aide de tirets (par exemple `creer-commentaire`). Les noms de méthodes actions sont associés aux IDs d'actions en retirant tout tiret des IDs, en mettant la première lettre de chaque mot en majuscule, et en ajoutant un préfixe `action` au résultat. Par exemple, l'ID d'action `creer-commentaire` correspond à l'action nommée `actionCreerCommentaire`.

La méthode action de notre exemple prend un paramètre `$message`, dont la valeur par défaut est `"Hello"` (de la même manière qu'on affecte une valeur par défaut à n'importe quel argument de fonction ou méthode en PHP). Quand l'application reçoit une requête et détermine que l'action `dire` est responsable de gérer ladite requête, l'application peuplera ce paramètre avec le paramètre du même nom trouvé dans la requête. En d'autres termes, si la requête contient un paramètre `message` ayant pour valeur `"Goodbye"`, la variable `$message` au sein de l'action recevra cette valeur.

Au sein de la méthode action, `render()` est appelé pour effectuer le rendu d'un fichier vue appelé `dire`. Le paramètre `message` est également transmis à la vue afin qu'il puisse y être utilisé. Le résultat du rendu est renvoyé à l'utilisateur par la méthode action. Ce résultat sera reçu par l'application et présenté à l'utilisateur dans le navigateur (en tant qu'élément d'une page HTML complète).

2.3.2 Créer une Vue

Les vues sont des scripts qu'on écrit pour générer le contenu d'une réponse. Pour la tâche "Hello", vous allez créer une vue `dire` qui affiche le paramètre `message` reçu de la méthode action, et passé par l'action à la vue :

```
<?php
use yii\helpers\Html;
?>
<?= Html::encode($message) ?>
```

La vue `dire` doit être enregistrée dans le fichier `views/site/dire.php`. Quand la méthode `render()` est appelée dans une action, elle cherchera un fichier PHP nommé `views/ControllerID/NomDeLaVue.php`.

Notez que dans le code ci-dessus, le paramètre `message` est `Encodé-HTML` avant d'être affiché. Cela est nécessaire car le paramètre vient de l'utilisateur, le rendant vulnérable aux attaques cross-site scripting (XSS) [25] en intégrant du code Javascript malicieux dans le paramètre.

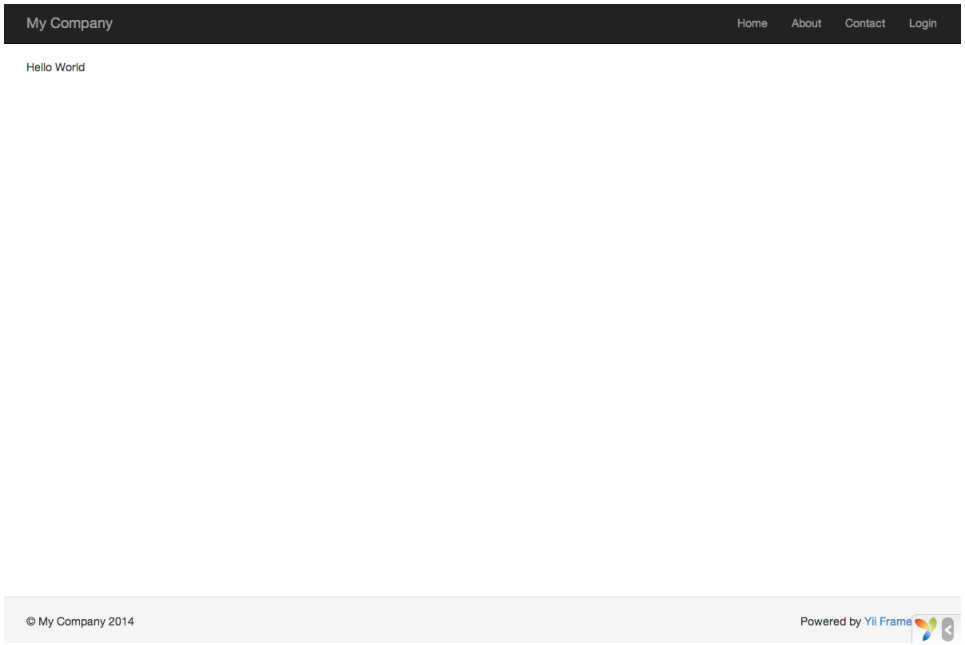
25. https://fr.wikipedia.org/wiki/Cross-site_scripting

Bien entendu, vous pouvez insérer plus de contenu dans la vue `dire`. Le contenu peut être des tags HTMML, du texte brut, ou même des expressions PHP. En réalité, la vue `dire` est simplement un script PHP exécuté par la méthode `render()`. Le contenu affiché par le script de vue sera renvoyé à l'application en tant que résultat de réponse. L'application renverra à son tour ce résultat à l'utilisateur.

2.3.3 Essayer

Après avoir créé l'action et la vue, vous pouvez accéder à la nouvelle page en accédant à l'URL suivant :

```
https://hostname/index.php?r=site/dire&message=Hello+World
```



Le résultat de cet URL sera une page affichant "Hello World". La page a les mêmes entête et pied de page que les autres pages de l'application.

Si vous omettez le paramètre `message` dans l'URL, La page devrait simplement afficher "Hello". C'est parce que `message` est passé en paramètre de la méthode `actionDire()`, et quand il est omis, la valeur par défaut `"Hello"` sera employée à la place.

> **Info :** L nouvelle page a les mêmes entête et pied de page que les autres pages parce que la méthode `render()` intègrera automatiquement le résultat de la vue `dire` dans une pseudo mise en page qui dans notre cas est située dans `views/layouts/main.php`.

Le paramètre `r` dans l'URL ci-dessus nécessite plus d'explications. Il signifie route, un ID unique commun toute l'application qui fait référence

à une action. Le format de la route est `IDContrôleur/IDAction`. Quand l'application reçoit une requête, elle vérifie ce paramêtre, en utilisant la partie `IDContrôleur` pour déterminer quel classe contrôleur doit être instanciée pour traiter la requête. Ensuite, le contrôleur utilisera la partie `IDAction` pour déterminer quelle action doit être instanciée pour effectuer le vrait travail. Dans ce cas d'exemple, la route `site/dire` sera comprise comme la classe contrôleur `SiteController` et l'action `dire`. Il en resultera que la méthode `SiteController::actionDire()` sera appelée pour traiter la requête.

> **Info :** De même que les actions, les contrôleurs ont des IDs qui les identifient de manière unique dans une application. Les IDs de contrôleurs emploie les mêmes règles de nommage que les IDs d'actions. Les noms de classes Contrôleurs dérivent des IDs de contrôleurs en retirant les tirets des IDs, en mettant la première lettre de chaque mot en majuscule, et en suffixant la chaîne résultante du mot `Controller`. Par exemple, l'ID de contrôlleur `poster-commentaire` correspond au nom de classe contrôleur `PosterCommentaireController`.

2.3.4 Résumé

Dans cette section, vous avez touché aux parties contrôleur et vue du patron de conception MVC. Vous avez créé une action au sein d'un contrôleur pour traiter une requête spécifique. Vous avez également créé une vue pour composer le contenu de la réponse. Dans ce simple exemple, aucun modèle n'a été impliqué car les seules données utilisées étaient le paramètre `message`.

Vous avez également appris ce que sont les routes dans Yii, qu'elles font office de pont entre les requêtes utilisateur et les actions des contrôleurs.

Dans la prochaine section, vous apprendrez comment créer un modèle, et ajouter une nouvelle page contenant un formulaire HTML.

2.4 Travailler avec les formulaires

Cette section décrit la création d'une nouvelle page comprenant un formulaire pour recevoir des données des utilisateurs. La page affichera un formulaire composé d'un champ de saisie nom et un champ de saisie email. Une fois ces deux informations reçues de l'utilisateur, la page affichera les valeurs entrées pour confirmation.

Pour atteindre ce but, vous créerez non seulement une action et deux vues, mais aussi un modèle.

Au long de ce tutoriel, vous apprendrez à :

— Créer un modèle pour représenter les données saisies par un utilisateur à travers un formulaire
— Déclarer des règles pour valider les données entrées
— Construire un formulaire HTML dans une vue

2.4.1 Créer un Modèle

Les données demandées à l'utilisateur seront représentées par une classe de modèle `EntryForm` comme montrée ci-dessous enregistrée dans le fichier `models/EntryForm.php`. Merci de vous référer à la section Auto-chargement de Classes pour plus de détails sur la convention de nommage de fichiers classes.

```
<?php

namespace app\models;

use yii\base\Model;

class EntryForm extends Model
{
    public $nom;
    public $email;

    public function rules()
    {
        return [
            [['nom', 'email'], 'required'],
            ['email', 'email'],
        ];
    }
}
```

La classe étend `yii\base\Model`, une classe de base fournie par Yii, communément utilisée pour représenter des données de formulaire.

> **Info :** `yii\base\Model` est utilisée en tant que parent pour des classes modèles qui ne sont *pas* associées à des tables de base de données. `yii\db\ActiveRecord` est normalement le parent pour les classes modèles qui correspondent à des tables de bases de données.

La classe `EntryForm` contient deux membres publics, `nom` et `email`, qui sont utilisés pour stocker les données saisies par l'utilisateur. Elle contient également une méthode nommée `rules()`, qui renvoie un assortiment de règles pour valider les données. Les règles de validation déclarées ci-dessus énoncent que

— à la fois les valeurs de `nom` et `email` sont requises
— la donnée `email` doit être une adresse email syntaxiquement valide

Si vous avez un objet `EntryForm` peuplé par les données saisies par un utilisateur, vous pouvez appeler sa méthode `validate()` pour déclencher les routines de validation de données. Un échec de validation de données affectera la valeur `true` à la propriété `hasErrors`, et vous pourrez connaître quelles erreurs de validations sont apparues via `errors`.

```
<?php
$model = new EntryForm();
```

```
$model->nom = 'Qiang';
$model->email = 'mauvais';
if ($model->validate()) {
    // Bien!
} else {
    // Echec!
    // Use $model->getErrors()
}
```

2.4.2 Créer une Action

Maintenant, vous allez créer une action `entry` dans le contrôleur `site` qui utilisera le nouveau modèle. Le processus de création et d'utilisation d'actions a été expliqué dans la section Hello World.

```
<?php

namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\EntryForm;

class SiteController extends Controller
{
    // ...code existant...

    public function actionEntry()
    {
        $model = new EntryForm;

        if ($model->load(Yii::$app->request->post()) && $model->validate())
        {
            // données valides reçues dans $model

            // faire quelque chose de significatif ici avec $model ...

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // soit la page est affichée au début soit il y a des erreurs de
            validation
            return $this->render('entry', ['model' => $model]);
        }
    }
}
```

L'action commence par créer un objet `EntryForm`. Puis, elle tente de peupler le modèle avec les données de `$_POST`, fournies dans yii par `yii\web\Request::post()`. Si le modèle est peuplé avec succès (c'est à dire, si l'utilisateur a soumis le formulaire HTML), l'action appellera `validate()` pour s'assurer de la validité de toutes les valeurs.

> **Info :** L'expression `Yii::$app` représente l'instance d'application, qui est un singleton accessible de manière globale. C'est aussi un annuaire de services qui fournit des composants tels que `request`, `response`, `db`, etc. pour assister des fonctionnalités spécifiques. Dans le code ci-dessus, le composant `request` de l'instance d'application est utilisé pour accéder aux données `$_POST`.

Si tout se passe bien, l'action effectuera le rendu d'une vue nommée `entry-confirm` pour confirmer le succès de la soumission à l'utilisateur. Si aucune donnée n'est soumise ou si les données contiennent des erreurs, la vue `entry` sera générée, dans laquelle le formulaire HTML sera affiché, ainsi que tout message d'erreur de validation.

> **Note :** Dans cet exemple très simple, nous effectuons le rendu de la page de confirmation après soumission de données valides. En pratique, vous devriez envisager d'utiliser `refresh()` ou `redirect()` pour éviter les problèmes de multiple soumission de formulaire [26].

2.4.3 Créer des Vues

Enfin, créez deux fichiers de vue nommés `entry-confirm` et `entry`. Ceux-ci seront rendus par l'action `entry`, comme décrit précédemment.

La vue `entry-confirm` affiche simplement les données de nom et email. Elle doit être placée dans le fichier `views/site/entry-confirm.php`.

```
<?php
use yii\helpers\Html;
?>
<p>Vous avez entré les informations suivantes :</p>

<ul>
    <li><label>Nom</label>: <?= Html::encode($model->nom) ?></li>
    <li><label>Email</label>: <?= Html::encode($model->email) ?></li>
</ul>
```

La vue `entry` affiche un formulaire HTML. Elle doit être stockée dans le placée `views/site/entry.php`.

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
?>
<?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'nom') ?>

    <?= $form->field($model, 'email') ?>
```

26. https://fr.wikipedia.org/wiki/Post-Redirect-Get

```
    <div class="form-group">
        <?= Html::submitButton('Soumettre', ['class' => 'btn btn-primary'])
?>
    </div>

<?php ActiveForm::end(); ?>
```

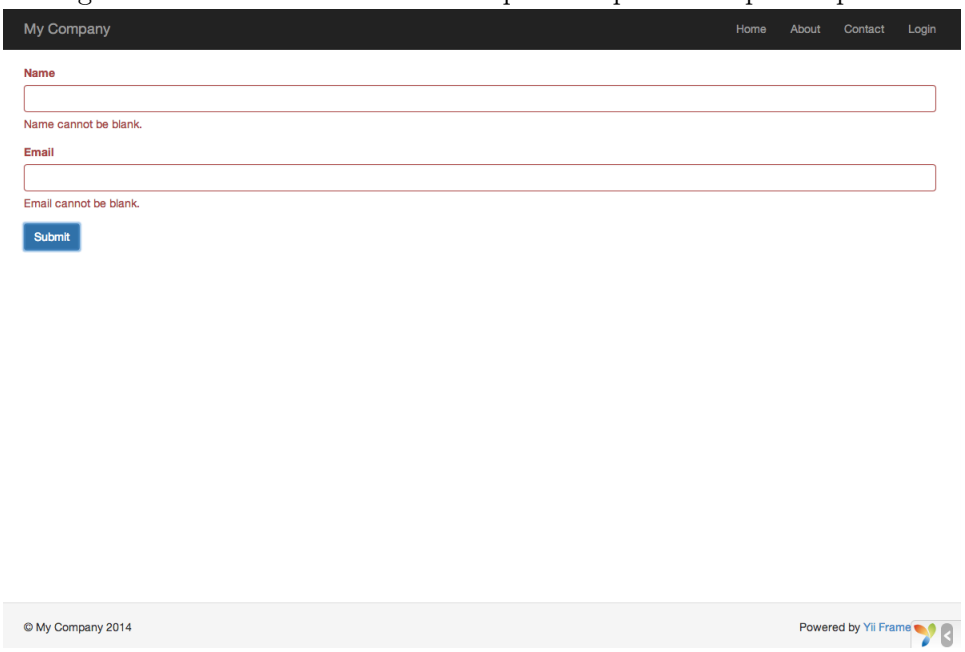
La vue utilise un widget puissant appelé `ActiveForm` pour construire le formulaire HTML. Les méthodes `begin()` et `end()` du widget effectuent respectivement le rendu des tags ouvrant et fermant du formulaire. Entre les deux appels de méthode, des champs de saisie sont créés par la méthode `field()`. Le premier champ de saisie concerne la donnée "nom", et le second la donnée "email". Après les champs de saisie, la méthode `yii\helpers\Html::submitButton()` est appelée pour générer un bouton de soumission.

2.4.4 Essayer

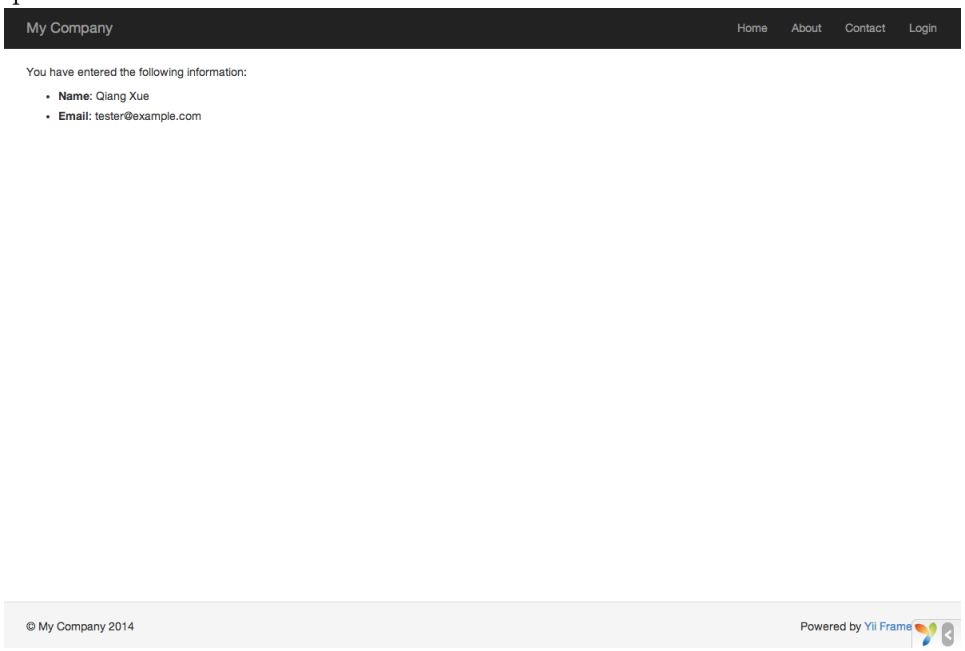
Pour voir comment ça fonctionne, utilisez votre navigateur pour accéder à l'URL suivante :

```
https://hostname/index.php?r=site/entry
```

Vous verrez une page affichant un formulaire comportant deux champs de saisie. Devant chaque champ de saisie, une étiquette indique quelle donnée est attendue. Si vous cliquez sur le bouton de soumission sans entrer quoi que ce soit, ou si vous ne fournissez pas d'adresse email valide, vous verrez un message d'erreur s'afficher à coté de chaque champ de saisie posant problème.



Après avoir saisi un nom et une adresse email valide et cliqué sur le bouton de soumission, vous verrez une nouvelle page affichant les données que vous venez de saisir.



La Magie expliquée

Vous vous demandez peut-être comment le formulaire HTML fonctionne en coulisse, parce qu'il semble presque magique qu'il puisse afficher une étiquette pour chaque champ de saisie et afficher sans rafraichir la page des messages d'erreur si vous n'entrez pas les données correctement.

Oui, la validation de données est initialement faite coté client en Javascript, et ensuite effectuée coté serveur en PHP. `yii\widgets\ActiveForm` est suffisamment intelligent pour extraire les règles de validation que vous avez déclarées dans `EntryForm`, le transformer en code Javascript exécutable, et utiliser le Javascript pour effectuer la validation des données. Dans le cas où vous auriez désactivé le Javascript sur votre navigateur, la validation sera tout de même effectuée coté serveur, comme montré dans la méthode `actionEntry()`. Cela garantit la validité des données en toutes circonstances.

> **Attention :** La validation coté client est un confort qui permet une meilleure expérience utilisateur. La validation coté serveur est toujours nécessaire, que la validation coté client soit ou non en place.

Les étiquettes des champs de saisie sont générés par la méthode `field()`, en utilisant les noms des propriété du modèle. Par exemple, l'étiquette `Nom` sera générée à partir de la propriété `nom`.

Vous pouvez personnaliser une étiquette dans une vue en employant le code suivant :

```
<?= $form->field($model, 'nom')->label('Votre Nom') ?>
<?= $form->field($model, 'email')->label('Votre Email') ?>
```

> **Info :** Yii fournit ne nombreux widgets pour vous aider à construire rapidement des vues complexes et dynamiques. Comme vous l'apprendrez plus tard, écrire un widget et aussi extrêmement simple. Vous voudrez sans doute transformer une grande partie de votre code de vues en widgets réutilisables pour simplifier les développements de vues futurs.

2.4.5 Résumé

Dans cette section du guide, vous avez touché toutes les parties du patron de conception MVC. Vous avez appris à créer une classe modèle pour représenter les données utilisateur et valider lesdites données.

Vous avez également appris comment recevoir des données des utilisateurs et comment les réafficher dans le navigateur. C'est une tâche qui peut prendre beaucoup de temps lors du développement d'une application, mais Yii propose des widgets puissants pour rendre cette tâche très facile.

Dans la prochaine section, vous apprendrez comment travailler avec des bases de données, qui sont nécessaires dans presque toutes les applications.

2.5 Travailler avec des bases de données

Cette section décrit comment créer une nouvelle page qui affiche des données pays récupérées dans une table de base de données nommée `country`. Pour ce faire, vous allez configurer une connexion à une base de données, créer une classe Active Record, définir une action, et créer une vue.

Au long de ce tutoriel, vous apprendrez comment :

— Configurer une connexion à une base de données
— Définir une classe Active Record
— Demander des données en utilisant la classe Active Record (enregistrement actif)
— Afficher des données dans une vue paginée

Notez que, pour finir cette section, vous aurez besoin d'avoir une connaissance basique des bases de données. En particulier, vous devez savoir créer une base de données et exécuter des déclarations SQL en utilisant un client de gestion de bases de données.

2.5.1 Préparer la Base de Données

Pour commencer, créez une base de données appelée `yii2basic`, depuis laquelle vous irez chercher les données dans votre application. Vous pouvez

créer une base de données SQLite, MySQL, PostgreSQL, MSSQL ou Oracle, car Yii gère nativement de nombreuses applications de base de données. Pour simplifier, nous supposerons que vous utilisez MySQL dans les descriptions qui suivent.

>Note : bien que MariaDB a été un remplacement direct de MySQL, cela n'est plus complètement vrai. Dans le cas où vous auriez besoin de fonctionnalités avancées telles que la prise en charge de `JSON`, jetez un coup d'œil à la liste des extensions de MariaDB ci-dessous.

Créez maintenant une table nommée `country` dans la base de données et insérez-y quelques données exemples. Vous pouvez exécuter l'instruction SQL suivante pour le faire :

```
CREATE TABLE `country` (
  `code` CHAR(2) NOT NULL PRIMARY KEY,
  `name` CHAR(52) NOT NULL,
  `population` INT(11) NOT NULL DEFAULT '0'
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `country` VALUES ('AU','Australia',24016400);
INSERT INTO `country` VALUES ('BR','Brazil',205722000);
INSERT INTO `country` VALUES ('CA','Canada',35985751);
INSERT INTO `country` VALUES ('CN','China',1375210000);
INSERT INTO `country` VALUES ('DE','Germany',81459000);
INSERT INTO `country` VALUES ('FR','France',64513242);
INSERT INTO `country` VALUES ('GB','United Kingdom',65097000);
INSERT INTO `country` VALUES ('IN','India',1285400000);
INSERT INTO `country` VALUES ('RU','Russia',146519759);
INSERT INTO `country` VALUES ('US','United States',322976000);
```

Vous avez désormais une base de données appelée `yii2basic` comprenant une table `country` comportant trois colonnes et contenant dix lignes de données.

2.5.2 Configurer une Connexion à la BDD

Avant de continuer, vérifiez que vous avez installé à la fois l'extension PHP PDO[27] et le pilote PDO pour la base de données que vous utilisez (c'est à dire `pdo_mysql` pour MySQL). C'est une exigence de base si votre application utilise une base de données relationnelle.

Une fois ces éléments installés, ouvrez le fichier `config/db.php` et modifiez les paramètres pour qu'ils correspondent à votre base de données. Par défaut, le fichier contient ce qui suit :

```
<?php

return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
```

27. https://www.php.net/manual/fr/book.pdo.php

```
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

Le fichier `config/db.php` est un exemple type d'outil de configuration basé sur un fichier. Ce fichier de configuration en particulier spécifie les paramètres nécessaires à la création et l'initialisation d'une instance de `yii\db\Connection` grâce à laquelle vous pouvez effectuer des requêtes SQL dans la base de données sous-jacente.

On peut accéder à connexion à la BDD configurée ci-dessus depuis le code de l'application via l'expression `Yii::$app->db`.

> **Info :** le fichier `config/db.php` sera inclus par la configuration principale de l'application `config/web.php`, qui spécifie comment l'instance d'application doit être initialisée. Pour plus d'informations, reportez-vous à la section Configurations.

Si vous avez besoin de fonctionnalités de base de données dont la prise en charge n'est pas comprise dans Yii, examinez les extensions suivantes :

— Informix [28]
— IBM DB2 [29]
— Firebird [30]
— MariaDB [31]

2.5.3 Créer un Active Record

Pour représenter et aller chercher des données dans la table `country`, créez une classe dérivée d'Active Record appelée `Country`, et enregistrez-la dans le fichier `models/Country.php`.

```
<?php

namespace app\models;

use yii\db\ActiveRecord;

class Country extends ActiveRecord
{
}
```

La classe `Country` étend `yii\db\ActiveRecord`. Vous n'avez pas besoin d'y écrire le moindre code ! Simplement, avec le code ci-dessus, Yii devine le nom de la table associée au nom de la classe.

28. https://github.com/edgardmessias/yii2-informix
29. https://github.com/edgardmessias/yii2-ibm-db2
30. https://github.com/edgardmessias/yii2-firebird
31. https://github.com/sam-it/yii2-mariadb

> **Info :** si aucune correspondance directe ne peut être faite à partir du nom de la classe, vous pouvez outrepasser la méthode `yii\db\ActiveRecord::tableName()` pour spécifier explicitement un nom de table.

A l'aide de la classe `Country`, vous pouvez facilement manipuler les données de la table `country`, comme dans les bribes suivantes :

```
use app\models\Country;

// chercher toutes les lignes de la table pays et les trier par "name"
$countries = Country::find()->orderBy('name')->all();

// chercher la ligne dont la clef primaire est "US"
$country = Country::findOne('US');

// afficher "United States"
echo $country->name;

// remplace le nom du pays par "U.S.A." et le sauvegarde dans la base de
données
$country->name = 'U.S.A.';
$country->save();
```

> **Info :** Active Record (enregistrement actif) est un moyen puissant pour accéder et manipuler des données d'une base de manière orientée objet. Vous pouvez trouver plus d'informations dans la section Active Record. Sinon, vous pouvez également interagir avec une base de données en utilisant une méthode de plus bas niveau d'accès aux données appelée Database Access Objects.

2.5.4 Créer une Action

Pour exposer les données pays aux utilisateurs, vous devez créer une action. Plutôt que de placer la nouvelle action dans le contrôleur `site`, comme vous l'avez fait dans les sections précédentes, il est plus cohérent de créer un nouveau contrôleur spécifique à toutes les actions liées aux données pays. Nommez ce contrôleur `CountryController`, et créez-y une action `index`, comme suit.

```
<?php

namespace app\controllers;

use yii\web\Controller;
use yii\data\Pagination;
use app\models\Country;

class CountryController extends Controller
{
```

```
    public function actionIndex()
    {
        $query = Country::find();

        $pagination = new Pagination([
            'defaultPageSize' => 5,
            'totalCount' => $query->count(),
        ]);

        $countries = $query->orderBy('name')
            ->offset($pagination->offset)
            ->limit($pagination->limit)
            ->all();

        return $this->render('index', [
            'countries' => $countries,
            'pagination' => $pagination,
        ]);
    }
}
```

Enregistrez le code ci-dessus dans le fichier `controllers/CountryController.php`.

L'action `index` appelle `Country::find()`. Cette méthode Active Record construit une requête de BDD et récupère toutes les données de la table `country`. Pour limiter le nombre de pays retournés par chaque requête, la requête est paginée à l'aide d'un objet `yii\data\Pagination`. L'objet `Pagination` dessert deux buts :

— Il ajuste les clauses `offset` et `limit` de la déclaration SQL représentée par la requête afin qu'elle en retourne qu'une page de données à la fois (au plus 5 colonnes par page).

— Il est utilisé dans la vue pour afficher un sélecteur de pages qui consiste en une liste de boutons de page, comme nous l'expliquerons dans la prochaine sous-section.

A la fin du code, l'action `index` effectue le rendu d'une vue nommée `index`, et lui transmet les données pays ainsi que les informations de pagination.

2.5.5 Créer une Vue

Dans le dossier `views`, commencez par créer un sous-dossier nommé `country`. Ce dossier sera utilisé pour contenir toutes les vues rendues par le contrôleur `country`. Dans le dossier `views/country`, créez un fichier nommé `index.php` contenant ce qui suit :

```
<?php
use yii\helpers\Html;
use yii\widgets\LinkPager;
?>
<h1>Countries</h1>
<ul>
```

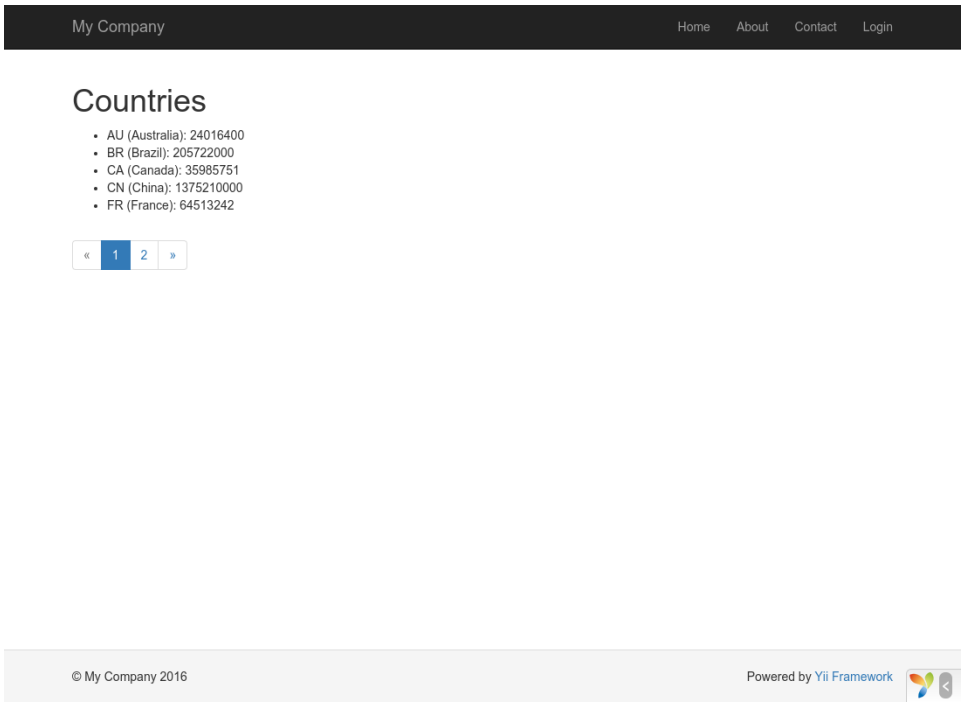
```
<?php foreach ($countries as $country): ?>
    <li>
        <?= Html::encode("{$country->code} ({$country->name})") ?>:
        <?= $country->population ?>
    </li>
<?php endforeach; ?>
</ul>

<?= LinkPager::widget(['pagination' => $pagination]) ?>
```

La vue comprend deux sections relatives à l'affichage des données pays. Dans la première partie, les données pays fournies sont parcourues et rendues sous forme de liste non ordonnée HTML. Dans la deuxième partie, un objet graphique `yii\widgets\LinkPager` est rendu en utilisant les informations de pagination transmises par l'action. L'objet graphique `LinkPager` affiche une liste de boutons de page. Le fait de cliquer sur l'un deux rafraichit les données pays dans la page correspondante.

2.5.6 Essayer

Pour voir comment tout le code ci-dessus fonctionne, pointez votre navigateur sur l'URL suivante :

```
https://hostname/index.php?r=country/index
```



Au début, vous verrez une page affichant cinq pays. En dessous des pays, vous verrez un sélecteur de pages avec quatre boutons. Si vous cliquez sur

le bouton "2", vous verrez la page afficher cinq autres pays de la base de données : la deuxième page d'enregistrements. Observez plus attentivement et vous noterez que l'URL dans le navigateur devient

```
https://hostname/index.php?r=country/index&page=2
```

En coulisse, `Pagination` fournit toutes les fonctionnalités permettant de paginer un ensemble de données :

— Au départ, `Pagination` représente la première page, qui reflète la requête SELECT de country avec la clause `LIMIT 5 OFFSET 0`. Il en résulte que les cinq premiers pays seront trouvés et affichés.

— L'objet graphique `LinkPager` effectue le rendu des boutons de pages en utilisant les URL créées par `Pagination`. Les URL contiendront le paramètre de requête `page`, qui représente les différents numéros de pages.

— Si vous cliquez sur le bouton de page "2", une nouvelle requête pour la route `country/index` sera déclenchée et traitée. `Pagination` lit le paramètre de requête `page` dans l'URL et met le numéro de page à 2. La nouvelle requête de pays aura donc la clause `LIMIT 5 OFFSET 5` et retournera les cinq pays suivants pour être affichés.

2.5.7 Résumé

Dans cette section, vous avez appris comment travailler avec une base de données. Vous avez également appris comment chercher et afficher des données dans des pages avec l'aide de `yii\data\Pagination` et de `yii\widgets\LinkPager`.

Dans la prochaine section, vous apprendrez comment utiliser le puissant outil de génération de code, appelé Gii[32], pour vous aider à rapidement mettre en œuvre des fonctionnalités communément requises, telles que les opérations Créer, Lire, Mettre à Jour et Supprimer (CRUD : Create-Read-Update-Delete) pour travailler avec les données dans une table de base de données. En fait, le code que vous venez d'écrire peut être généré automatiquement dans Yii en utilisant l'outil Gii.

2.6 Générer du code avec Gii

Cette section décrit comment utiliser Gii[33] pour générer du code qui met automatiquement en œuvre des fonctionnalités courantes de sites Web. Utiliser Gii pour auto-générer du code consiste simplement à saisir les bonnes informations en suivant les instructions affichées sur les pages Web de Gii.

Au long de ce tutoriel, vous apprendrez comment :

32. https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide
33. https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide

— Activer Gii dans votre application
— Utiliser Gii pour générer des classes Active Record (enregistrement actif)
— Utiliser Gii pour générer du code mettant en œuvre les opérations CRUD pour une table de BDD
— Personnaliser le code généré par Gii

2.6.1 Démarrer Gii

Gii[34] est fourni dans Yii en tant que module. Vous pouvez activer Gii en le configurant dans la propriété `modules` de l'application. En fonction de la manière dont vous avez créé votre application, il se peut que le code suivant soit déjà fourni dans le fichier de configuration `config/web.php` :

```
$config = [ ... ];

if (YII_ENV_DEV) {
    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = [
        'class' => 'yii\gii\Module',
    ];
}
```

La configuration ci-dessus établit que dans un environnement de développement, l'application doit inclure un module appelé `gii`, qui est de classe `yii\gii\Module`.

Si vous vérifiez le script de démarrage `web/index.php` de votre application, vous y trouverez les lignes suivantes, qui en gros, font que `YII_ENV_DEV` est défini à `true` (vrai).

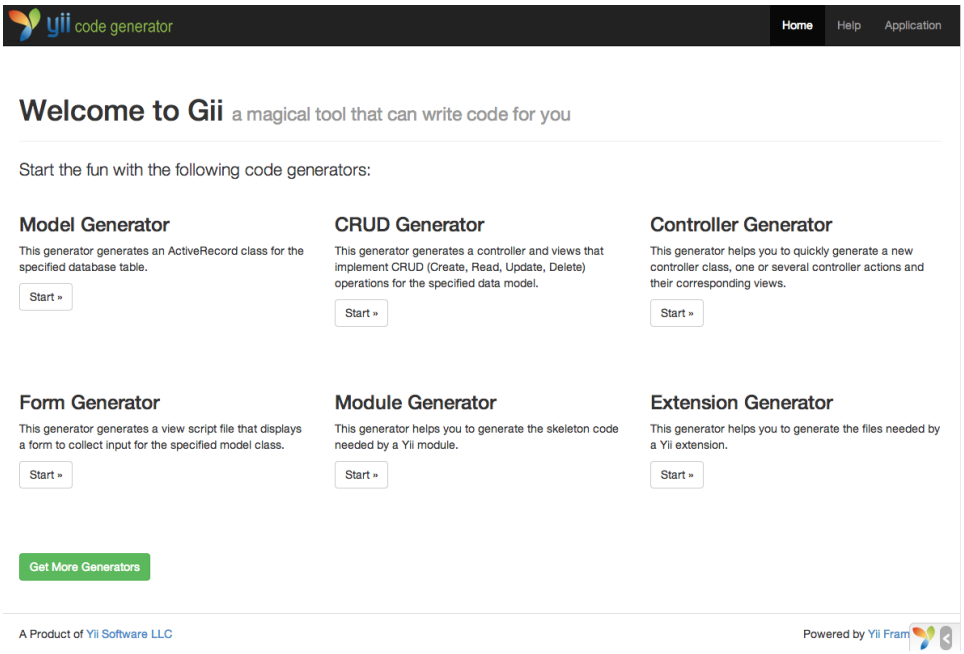
```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

Grâce à cette ligne, votre application est en mode développement, et active Gii, suivant la configuration vue ci-dessus. Vous pouvez maintenant accéder à Gii via l'URL suivante :

```
https://hostname/index.php?r=gii
```

> Note : si vous accédez à Gii depuis une machine autre que localhost, l'accès sera refusé par défaut pour des raisons de sécurité. Vous pouvez configurer Gii pour ajouter les adresses IP autorisées comme suit,
>
> ```
> 'gii' => [
>     'class' => 'yii\gii\Module',
>     'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*',
>     '192.168.178.20'] // ajustez cela suivant vos besoins
> ],
> ```

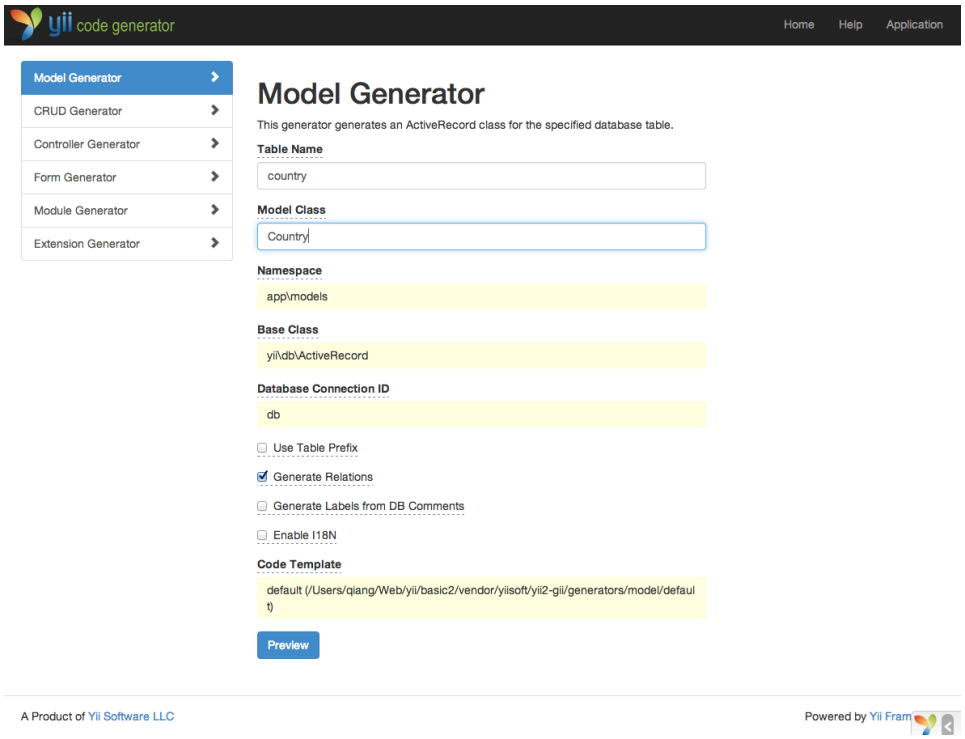
34. https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide



2.6.2 Générer une Classe Active Record

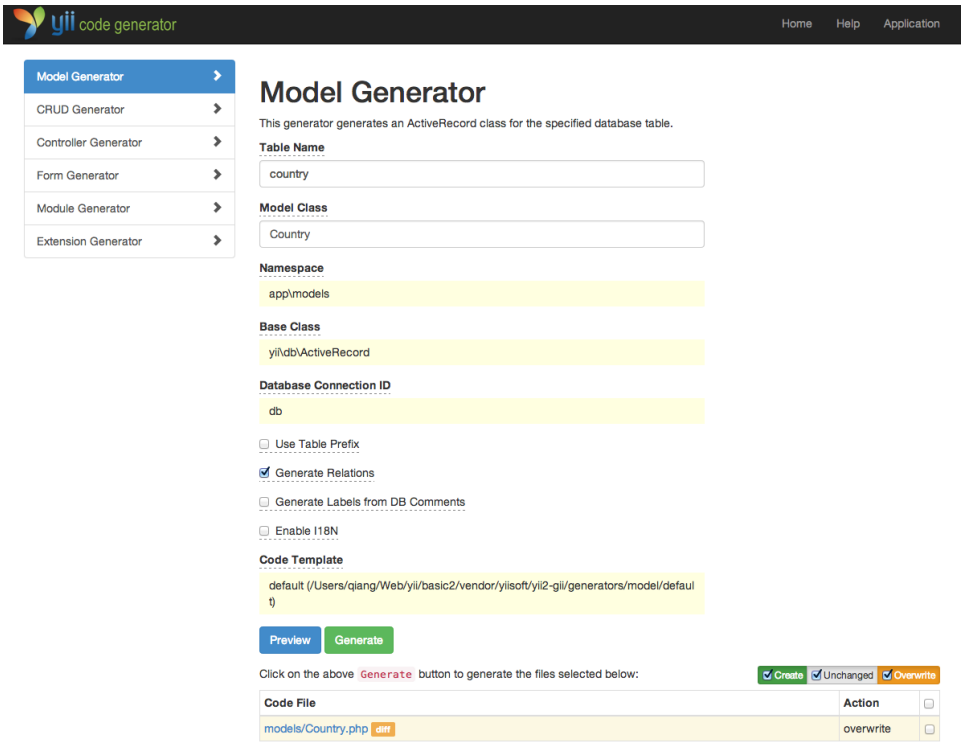
Pour générer une classe Active Record avec Gii, sélectionnez le “Model Generator” (générateur de modèle), en cliquant sur le lien dans la page d’accueil de Gii, puis complétez le formulaire comme suit :

— Table Name : `country`
— Model Class : `Country`



Ensuite, cliquez sur le bouton “Preview” (prévisualiser). Vous verrez que `models/Country.php` est listé comme fichier de classe à créer. Vous pouvez cliquer sur le nom du fichier de classe pour prévisualiser son contenu.

Si vous avez déjà créé le même fichier, il sera écrasé. Cliquez sur le bouton `diff` à côté du nom de fichier pour voir les différences entre le fichier à générer et la version existante.



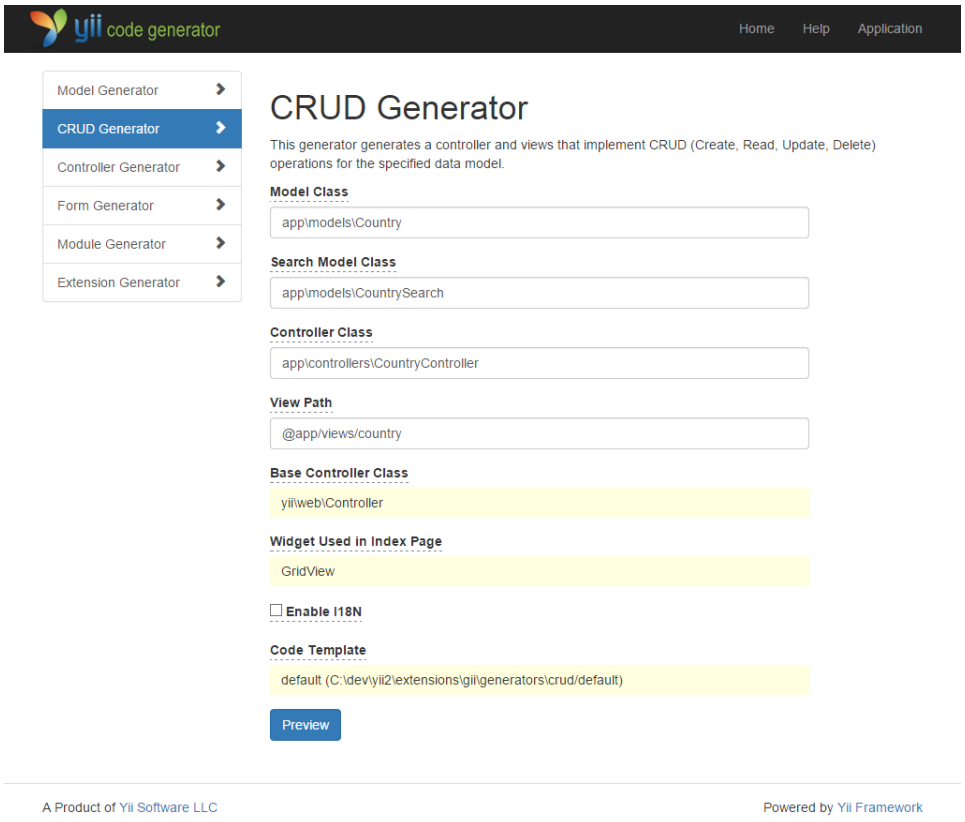
Pour écraser un fichier existant, cochez la case située à côté de “overwrite” (écraser), puis cliquez sur le bouton “Generate” (générer). Pour créer un nouveau fichier, il suffit de cliquer sur “Generate”.

En fin d’opération, vous verrez une page de confirmation indiquant que le code a été généré avec succès. Si vous aviez un fichier existant, vous verrez également un message indiquant qu’il a été écrasé par le code nouvellement généré.

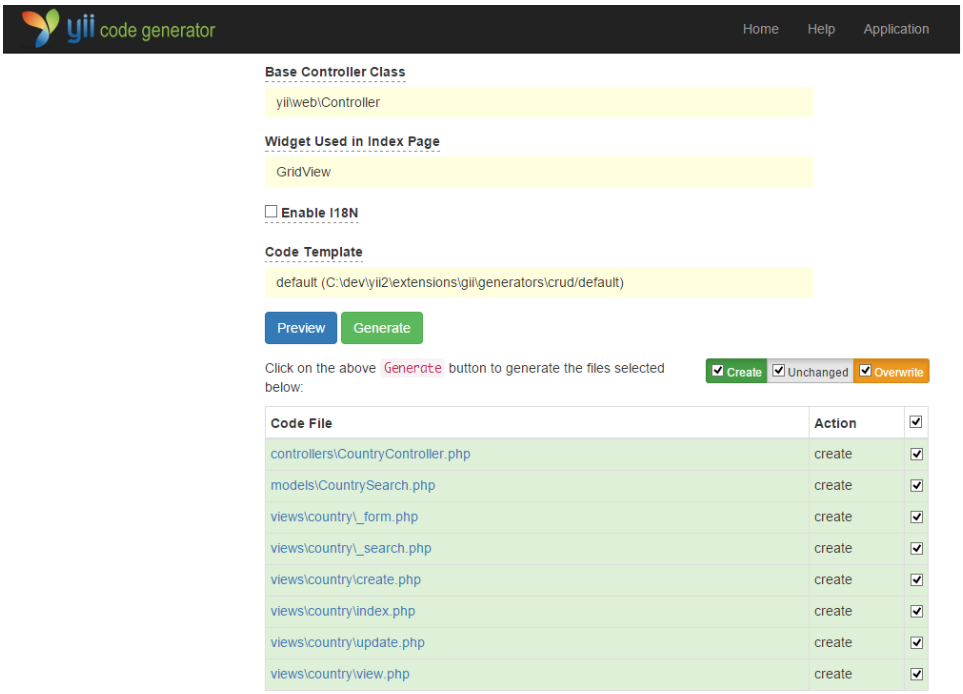
2.6.3 Générer du Code CRUD

CRUD signifie Create, Read, Update, and Delete (Créer, Lire, Mettre à Jour et Supprimer), soit les quatre tâches communes concernant des données sur la plupart des sites Web. Pour créer les fonctionnalités CRUD en utilisant Gii, sélectionnez le “CRUD Generator” en cliquant sur le lien dans la page d’accueil de Gii. Pour l’exemple de “country”, remplissez le formulaire résultant comme suit :

— Model Class : `app\models\Country`
— Search Model Class : `app\models\CountrySearch`
— Controller Class : `app\controllers\CountryController`



Ensuite, cliquez sur le bouton “Preview” (prévisualiser). Vous verrez une liste de fichiers à générer, comme ci-dessous.



Si vous aviez précédemment créé les fichiers `controllers/CountryController.php` et `views/country/index.php` (dans la section bases de données du guide), cochez la case “overwrite” (écraser) pour les remplacer. (Les versions précédentes ne prenaient pas totalement en charge les fonctionnalités CRUD).

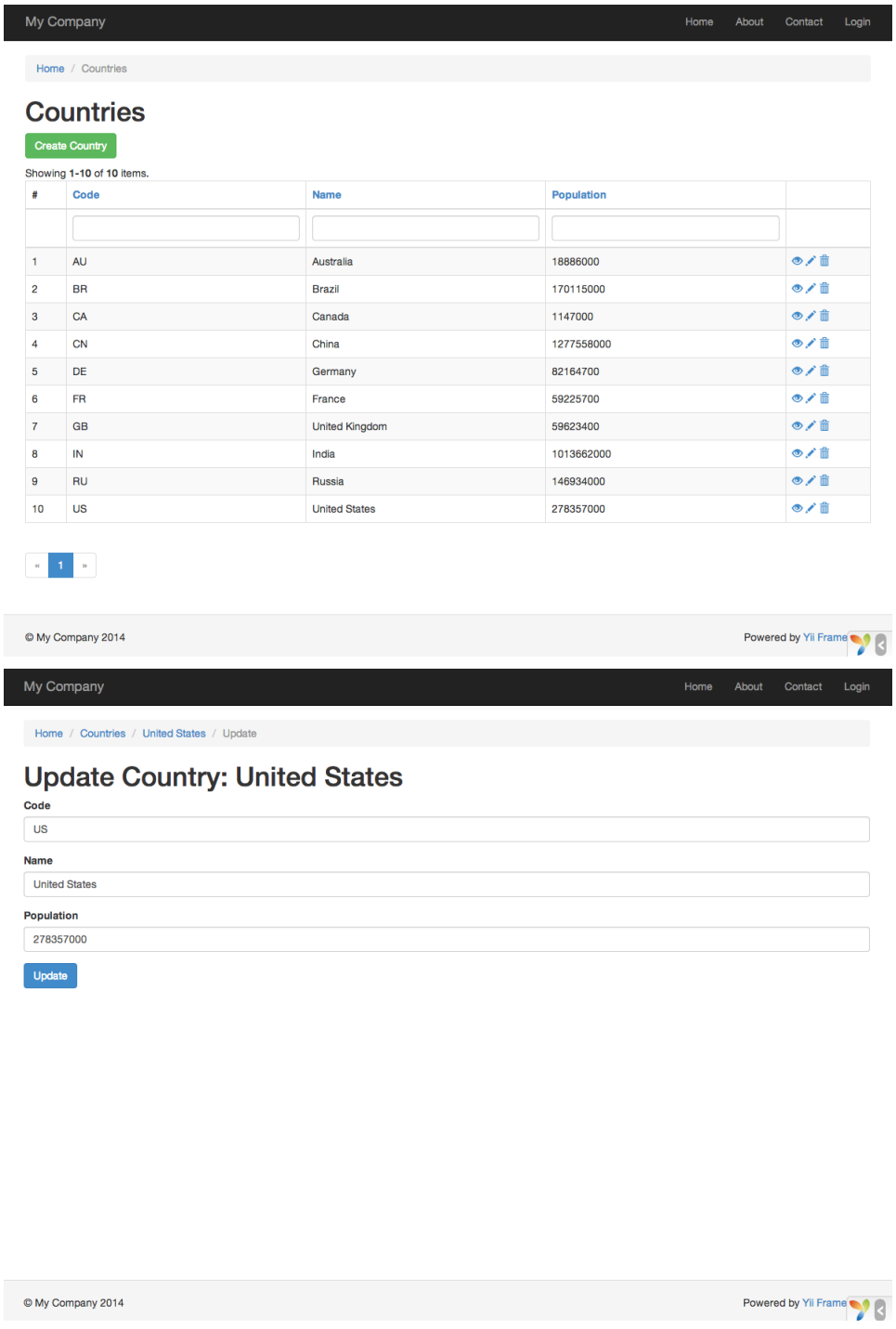
2.6.4 Essayer

Pour voir comment ça fonctionne, utilisez votre navigateur pour accéder à l’URL suivant :

```
https://hostname/index.php?r=country/index
```

Vous verrez une grille de données montrant les pays de la table de la base de données. Vous pouvez trier la table, ou lui appliquer des filtres en entrant des conditions de filtrage dans les entêtes de colonnes.

Pour chaque pays affiché dans la grille, vous pouvez choisir de visualiser les détails, le mettre à jour ou le supprimer. Vous pouvez aussi cliquer sur le bouton “Create Country” (créer un pays) en haut de la grille pour que Yii vous présente un formulaire permettant de créer un nouveau pays.



Ce qui suit est la liste des fichiers générés par Gii, au cas où vous souhaiteriez investiguer la manière dont ces fonctionnalités sont mises en œuvre, ou les personnaliser :

— Contrôleur : `controllers/CountryController.php`

— Modèles : `models/Country.php` et `models/CountrySearch.php`
— Vues : `views/country/*.php`

> **Info :** Gii est conçu pour être un outil de génération de code hautement personnalisable et extensible. L'utiliser avec sagesse peut grandement accélérer le développement de vos applications. Pour plus de détails, merci de vous référer à la section Gii.

2.6.5 Résumé

Dans cette section, vous avez appris à utiliser Gii pour générer le code qui met en œuvre une fonctionnalité CRUD complète pour des contenus stockés dans une table de base de données.

2.7 En savoir plus

Si vous avez entièrement lu la section "Mise en Route", vous avez maintenant créé une application Yii complète. Ce faisant, vous avez appris comment mettre en œuvre des fonctionnalités couramment utilisées, telles que recueillir des données d'un utilisateur via un formulaire HTML, chercher des données dans une base de données, et afficher des données de manière paginée. Vous avez également appris à utiliser Gii [35] pour générer du code automatiquement. Utiliser Gii pour générer du code rend le gros de votre processus de développement Web aussi simple que de remplir de simples formulaires.

Cette section va résumer les ressources Yii disponibles pour vous aider à être plus productif dans l'utilisation du framework.

— Documentation
  — Le Guide complet [36] : Comme son nom l'indique, le guide définit précisément comment Yii fonctionne et fournit des instructions générales sur l'utilisation de Yii. C'est le tutoriel pour Yii le plus important, un que vous devriez lire avant d'écrire le moindre code Yii.
  — Le référentiel des Classes [37] : Il spécifie le mode d'utilisation de toutes les classes fournies par Yii. Il doit être principalement utilisé lorsque vous écrivez du code et souhaitez comprendre le mode d'utilisation d'une classe, méthode ou propriété particulière. L'utilisation du référentiel des classes est plus appropriée quand vous avez une compréhension contextuelle du framework entier.
  — Les Articles du Wiki [38] : Les articles wiki sont écrits par des utilisateurs de Yii sur la base de leurs propres expériences. Ils sont

35. https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide
36. https://www.yiiframework.com/doc-2.0/guide-README.html
37. https://www.yiiframework.com/doc-2.0/index.html
38. https://www.yiiframework.com/wiki/?tag=yii2

en général écrits comme des recettes de cuisine, et montrent comment résoudre des problèmes pratiques en utilisant Yii. Bien que la qualité de ces articles puisse être moindre que celle du Guide complet, ils sont utiles du fait qu'ils couvrent des sujets plus vastes et peuvent fournir des solutions clef-en-main.
    - Livres [39]
- Extensions [40] : Yii est fort d'une librairie de milliers d'extensions créées par les utilisateurs, qui peuvent être facilement ajoutées à votre application, rendant son développement encore plus facile et plus rapide.
- Communauté
    - Forum : https://forum.yiiframework.com/
    - Chat IRC : Les canal #yii sur le réseau Libera (ircs://irc.libera.chat:6697/yii)
    - Slack chanel : https://yii.slack.com
    - Gitter chat : https://gitter.im/yiisoft/yii2
    - GitHub : https://github.com/yiisoft/yii2
    - Facebook : https://www.facebook.com/groups/yiitalk/
    - Twitter : https://twitter.com/yiiframework
    - LinkedIn : https://www.linkedin.com/groups/yii-framework-1483367
    - Stackoverflow : https://stackoverflow.com/questions/tagged/yii2

39. https://www.yiiframework.com/books
40. https://www.yiiframework.com/extensions/

Chapitre 3

Structure Application

3.1 Vue d'ensemble

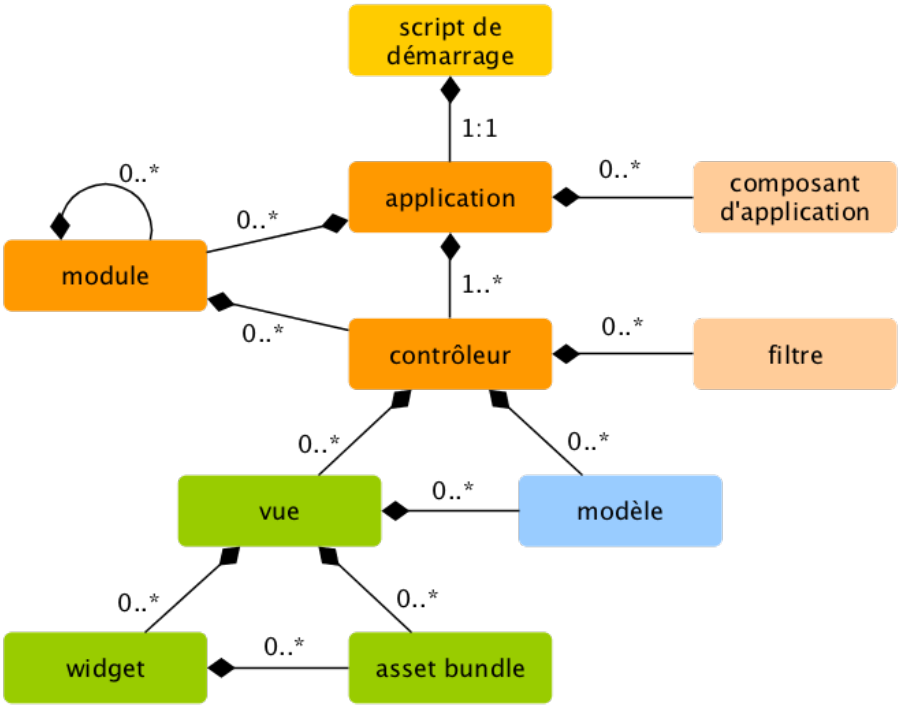
Les applications Yii sont organisées suivant le modèle de conception model-view-controller (MVC) [1]. Les Modèles représentent les données, la logique métier et les règles ; les vues sont les représentations visuelles des modèles, et les contrôleurs prennent une entrée et la convertissent en commandes pour les modèles et les vues.

En plus du MVC, les applications Yii ont les entités suivantes :

— scripts d'entrée : ce sont des scripts PHP qui sont directement accessibles aux utilisateurs. Ils sont responsables de l'amorçage d'un cycle de gestion de requête.
— applications : ce sont des objets globalement accessibles qui gèrent les composants d'application et les coordonnent pour satisfaire des requêtes.
— composants d'application : ce sont des objets enregistrés avec des applications et qui fournissent différents services pour satisfaire des requêtes.
— modules : ce sont des paquets auto-contenus qui contiennent du MVC complet. Une application peut être organisée en de multiples modules.
— filtres : ils représentent du code qui doit être invoqué avant et après la gestion effective de chaque requête par des contrôleurs.
— objets graphiques : ce sont des objets qui peuvent être intégrés dans des vues. Ils peuvent contenir de la logique contrôleur et peuvent être réutilisés dans différentes vues.

Le diagramme suivant montre la structure statique d'une application :

1. https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur



3.2 Scripts d'entrée

Le script d'entrée est le premier rencontré dans le processus d'amorçage de l'application. Une application (qu'elle soit une application Web ou une application console) a un unique script d'entrée. Les utilisateurs font des requêtes au script d'entrée qui instancie un objet *Application* et lui transmet les requêtes.

Les scripts d'entrée des applications Web doivent être placés dans des dossiers accessibles par le Web pour que les utilisateurs puissent y accéder. Ils sont souvent nommés `index.php`, mais peuvent également avoir tout autre nom, du moment que les serveurs Web peuvent les trouver.

Les scripts d'entrée des applications console sont généralement placés dans le répertoire de base des applications et sont nommés `yii` (avec le suffixe `.php`). Ils doivent être rendus exécutables afin que les utilisateurs puissent lancer des applications console grâce à la commande `./yii <route> [arguments] [options]`.

Les scripts d'entrée effectuent principalement les tâches suivantes :

— Définir des constantes globales ;
— Enregistrer le chargeur automatique Composer [2] ;
— Inclure le fichier de classe de `Yii` ;

2. https://getcomposer.org/doc/01-basic-usage.md#autoloading

— Charger la configuration de l'application ;
— Créer et configurer une instance d'application ;
— Appeler `yii\base\Application::run()` pour traiter la requête entrante.

3.2.1 Applications Web

Ce qui suit est le code du script d'entrée du Modèle Basique d'Application Web.

```
<?php

defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

// register Composer autoloader
require __DIR__ . '/../vendor/autoload.php';

// include Yii class file
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

// load application configuration
$config = require __DIR__ . '/../config/web.php';

// create, configure and run application
(new yii\web\Application($config))->run();
```

3.2.2 Applications Console

De même, le code qui suit est le code du script de démarrage d'une application console :

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 *
 * @link https://www.yiiframework.com/
 * @copyright Copyright (c) 2008 Yii Software LLC
 * @license https://www.yiiframework.com/license/
 */

defined('YII_DEBUG') or define('YII_DEBUG', true);

// register Composer autoloader
require __DIR__ . '/vendor/autoload.php';

// include Yii class file
require __DIR__ . '/vendor/yiisoft/yii2/Yii.php';

// load application configuration
$config = require __DIR__ . '/config/console.php';
```

```
$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

3.2.3 Définir des Constantes

Les scripts d'entrée sont l'endroit idéal pour définir des constantes globales. Yii prend en charge les trois constantes suivantes :

— `YII_DEBUG` : spécifie si une application tourne en mode de débogage. Si elle est en mode de débogage, une application enregistrera des journaux plus détaillés, et révélera des piles d'appels d'erreurs détaillées si des exceptions sont levées. C'est pour cette raison que le mode de débogage doit être utilisé principalement pendant la phase de développement. La valeur par défaut de `YII_DEBUG` est `false` (faux).

— `YII_ENV` : spécifie dans quel environnement l'application est en train de tourner. Cela est décrit plus en détails dans la section Configurations. La valeur par défaut de `YII_ENV` est `'prod'`, ce qui signifie que l'application tourne dans l'environnement de production.

— `YII_ENABLE_ERROR_HANDLER` : spécifie si le gestionnaire d'erreurs fourni par Yii doit être activé. La valeur par défaut de cette constante est `true` (vrai).

Quand on définit une constante, on utilise souvent le code suivant :

```
defined('YII_DEBUG') or define('YII_DEBUG', true);
```

qui est l'équivalent du code suivant :

```
if (!defined('YII_DEBUG')) {
    define('YII_DEBUG', true);
}
```

Clairement, le premier est plus succinct et plus aisé à comprendre.

Les définitions de constantes doivent être faites au tout début d'un script d'entrée pour qu'elles puissent prendre effet quand d'autres fichiers PHP sont inclus.

3.3 Applications

Les Applications sont des objets qui gouvernent la structure d'ensemble et le cycle de vie des systèmes mettant en œuvre Yii. Chacun des systèmes mettant en œuvre Yii contient un objet *Application* unique qui est créé par le Script d'entrée et est globalement accessible à l'aide de l'expression `\Yii::$app`.

> **Info :** selon le contexte, lorsque nous utilisons le terme « application », cela peut signifier soit un objet *Application*, soit un système mettant en œuvre Yii.

Il existe deux types d'application : `les applications Web` et `les applications de console`. Comme leur nom l'indique, les premières prennent en charge des requêtes Web tandis que les deuxièmes prennent en charge des requêtes de la console.

3.3.1 Configurations d'application

Lorsqu'un script d'entrée crée une application, il charge une configuration et l'applique à cette application de la manière suivante :

```
require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

// charger la configuration de l'application
$config = require __DIR__ . '/../config/web.php';

// instancier et configurer l'application
(new yii\web\Application($config))->run();
```

Tout comme les configurations habituelles, les configurations d'application spécifient comment initialiser les propriétés des objets *Application*. Comme les configurations d'application sont souvent très complexes, elles sont ordinairement conservées dans des fichiers de configuration, tels que le fichier `web.php` de l'exemple précédent.

3.3.2 Propriétés des applications

Il y a de nombreuses propriétés importantes des applications que vous devez spécifier dans les configurations d'application. Ces propriétés décrivent l'environnement dans lequel ces applications sont exécutées. Par exemple, les applications doivent savoir comment charger les contrôleurs, où ranger les fichiers temporaires, etc. Nous allons passer en revue ces propriétés.

Propriétés requises

Dans toute application, vous devez au moins spécifier deux propriétés : : `id` et `basePath`.

`id` La propriété `id` spécifie un identifiant unique qui distingue une application des autres. On l'utilise principalement dans des instructions. Bien que cela ne soit pas une exigence, l'utilisation des seuls caractères alphanumériques, pour spécifier cet identifiant, est recommandée pour assurer une meilleure interopérabilité.

`basePath` La propriété `basePath` spécifie le dossier racine d'une application. Il s'agit du dossier qui contient tout le code source protégé d'une application mettant en œuvre Yii. Dans ce dossier, on trouve généralement des sous-dossiers tels que `models`, `views` et`controllers`, qui contiennent le code source correspondant au modèle de conception MVC.

Vous pouvez configurer la propriété `basePath` en utilisant un chemin de dossier ou un alias de chemin. Dans les deux cas, le dossier correspondant doit exister, sinon une exception est levée. Le chemin doit être normalisé à l'aide de la fonction `realpath()`.

La propriété `basePath` est souvent utilisée pour dériver d'autres chemins importants (p. ex. le chemin runtime). À cette fin, un alias nommé `@app` est prédéfini pour représenter ce chemin. Les chemins dérivés peuvent être formés à l'aide de cet alias (p. ex. `@app/runtime` pour faire référence au dossier `runtime`).

Propriétés importantes

Les propriétés décrites dans cette sous-section doivent souvent être spécifiées car elles différent à travers les différentes applications.

`alias` Cette propriété vous permet de définir un jeu d' alias sous forme de tableau associatif. Les clés du tableau représentent les noms des alias, tandis que les valeurs représentent la définition des chemins. Par exemple :

```
[
    'aliases' => [
        '@name1' => 'chemin/vers/dossier1',
        '@name2' => 'chemin/vers/dossier2',
    ],
]
```

Cette propriété est mise à votre disposition pour vous éviter d'avoir à définir les alias par programme en appelant la méthode `Yii::setAlias()`.

`bootstrap (amorçage)` Cette propriété est très utile. Elle vous permet de spécifier un tableau de composants qui devraient être exécutés lors du `processus d'amorçage`. Par exemple, si vous désirez utiliser un module pour personnaliser les règles d'URL, vous pouvez indiquer son identifiant (ID) en tant qu'élément de cette propriété.

Chacun des composants listés dans cette propriété peut être spécifié sous une des formes suivantes :

— un identifiant (ID) de composant d'application comme vous le spécifieriez via components,
— un identifiant (ID) de module comme vous le spécifieriez via modules,
— un nom de classe,

— un tableau de configuration,
— une fonction anonyme qui crée et retourne un composant.

Par exemple :

```
[
    'bootstrap' => [
        // un identifiant de composant d'application ou de module
        'demo',

        // un nom de classe
        'app\components\Profiler',

        // un tableau de configuration
        [
            'class' => 'app\components\Profiler',
            'level' => 3,
        ],

        // une fonction anonyme
        function () {
            return new app\components\Profiler();
        }
    ],
]
```

> **Info :** si un identifiant (ID) de module est identique à celui d'un composant d'application, le composant d'application est utilisé lors du processus de démarrage. Si vous désirez utiliser le module, vous pouvez le spécifier via une fonction anonyme comme le montre l'exemple suivant :
>
> ```
> [
>     function () {
>         return Yii::$app->getModule('user');
>     },
> ]
> ```

Los du processus d'amorçage, chaque composant est instancié. Si la classe du composant implémente `yii\base\BootstrapInterface`, sa méthode `bootstrap()` est également appelée.

Un autre exemple pratique se trouve dans la configuration de l'application du Modèle du projet Basic, où les modules `debug` et `gii` sont configurés en tant que composants d'amorçage lorsque l'application est dans l'environnement de développement.

```
if (YII_ENV_DEV) {
    // réglages de configuration pour l'environnement 'dev'
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';

    $config['bootstrap'][] = 'gii';
```

```
    $config['modules']['gii'] = 'yii\gii\Module';
}
```

> **Note :** placer trop de composants dans `bootstrap` dégrade la performance de votre application car, à chaque requête, le même jeu de composants doit être exécuté. C'est pourquoi vous devez utiliser les composants de démarrage avec discernement.

`catchAll (ramasse tout)` Cette propriété est prise en charge par les `applications Web` uniquement. Elle spécifie une action de contrôleur qui prend en charge toutes les requêtes de l'utilisateur. Cela est essentiellement utilisé lorsque l'application est dans le mode maintenance et doit prendre en charge toutes les requêtes avec une action unique. La configuration est un tableau dont le premier élément spécifie la route de l'action. Le reste des éléments du tableau (paires clé-valeur) spécifie les paramètres à associer à l'action. Par exemple :

```
[
    'catchAll' => [
        'offline/notice',
        'param1' => 'valeur1',
        'param2' => 'valeur2',
    ],
]
```

> **Info :** le panneau de débogage dans l'environnement de développement ne fonctionne pas lorsque cette propriété est activée.

`components (composants)` Il s'agit de la seule plus importante propriété. Elle vous permet d'enregistrer par leur nom une liste de composants appelés composants d'application que vous pouvez utiliser partout ailleurs. Par exemple :

```
[
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
        ],
    ],
]
```

Chaque composant d'application est spécifié sous la forme d'un couple clé-valeur dans le tableau. La clé représente l'identifiant (ID) du composant,

tandis que la valeur représente le nom de la classe du composant ou un tableau de configuration.

Vous pouvez enregistrer n'importe quel composant dans une application, et vous pouvez ensuite y accéder globalement via l'expression `\Yii::$app->componentID`.

Reportez-vous à la section Composants d'application pour plus de détails.

`controllerMap (Table de mise en correspondance des contrôleurs)` Cette propriété vous permet de faire correspondre un identifiant (ID) de contrôleur avec une classe de contrôleur arbitraire. Par défaut, Yii fait correspondre un identifiant de contrôleur avec une classe de contrôleur selon une convention (p. ex. l'identifiant `post` correspond à `app\controllers\PostController`). En configurant cette propriété, vous passez outre la convention pour les contrôleurs spécifiés. Dans l'exemple qui suit, `account` correspond à `app\controllers\UserController`, tandis que `article` correspond à `app\controllers\PostController`.

```
[
    'controllerMap' => [
        'account' => 'app\controllers\UserController',
        'article' => [
            'class' => 'app\controllers\PostController',
            'enableCsrfValidation' => false,
        ],
    ],
]
```

Les clés du tableau de cette propriété représentent les identifiants des contrôleurs, tandis que les valeurs représentent les noms des classes mises en correspondance ou les tableaux de configurations.

`controllerNamespace (espaces de noms des contrôleurs` Cette propriété spécifie l'espace de noms par défaut sous lequel les classes des contrôleurs sont situées. Par défaut, il s'agit de `app\controllers`. Si l'identifiant d'un contrôleur est `post`, par convention le contrôleur correspondant (sans l'espace de noms) est `PostController`, et le nom de classe totalement qualifié est `app\controllers\PostController`.

Les classes de contrôleur peuvent aussi résider dans des sous-dossiers du dossier correspondant à cet espace de noms. Par exemple, étant donné un identifiant de contrôleur`admin/post`, le nom de classe de contrôleur totalement qualifié est `app\controllers\admin\PostController`.

Il est important que la classe de contrôleur totalement qualifiée puisse être auto-chargée et que l'espace de noms réel de votre classe de contrôleur corresponde à la valeur de cette propriété. Autrement, vous obtenez une erreur « Page non trouvée » quand vous accédez à votre application.

Si vous désirez passer outre la convention décrite précédemment, vous devez configurer la propriété controllerMap.

`language (langue)` Cette propriété spécifie la langue dans laquelle l'application présente les contenus aux utilisateurs finaux. La valeur par défaut de cette propriété est `en`, pour anglais. Vous devez configurer cette propriété si votre application doit prendre en charge plusieurs langues.

La valeur de cette propriété détermine des aspects variés de l'internationalisation tels que la traduction des messages, le formatage des dates et des nombres, etc. Par exemple, l'objet graphique `yii\jui\DatePicker` utilise la valeur de cette propriété pour déterminer dans quelle langue le calendrier doit être affiché et comment les dates doivent être formatées.

La spécification de la langue par une étiquette IETF d'identification de langue[3] est recommandée. Par exemple, `en` signifie anglais, tandis que `en-US` signifie anglais (États-Unis)..

Pour plus d'informations sur cette propriété, reportez-vous à la section Internationalisation.

`modules` Cette propriété spécifie les modules que comprend l'application.

Cette propriété accepte un tableau de classes de module ou de tableaux de configurations dans lequel les clés sont les identifiants (ID) des modules. Par exemple :

```
[
    'modules' => [
        // un module "booking" (réservations) spécifié par sa classe
        'booking' => 'app\modules\booking\BookingModule',

        // un module  "comment" (commentaires) spécifié par un tableau de
        configuration
        'comment' => [
            'class' => 'app\modules\comment\CommentModule',
            'db' => 'db',
        ],
    ],
]
```

Reportez-vous à la section Modules pour des informations complémentaires.

`name (nom` Cette propriété spécifie le nom de l'application qui est présenté à l'utilisateur final. Contrairement à la propriété `id` qui ne peut prendre qu'une valeur unique, la valeur de cette propriété, qui n'intervient que pour l'affichage, n'a pas besoin d'être unique. Vous n'avez pas besoin de configurer cette propriété si vous ne l'utilisez pas dans votre code.

3. https://fr.wikipedia.org/wiki/%C3%89tiquette_d%27identification_de_langues_IETF

`params (paramètres)` Cette propriété spécifie un tableau de paramètres de l'application accessibles globalement. Plutôt que de parsemer votre code des mêmes nombres et chaînes de caractères formulées `en dur`, une bonne pratique consiste à les définir une fois pour toute sous forme de paramètres et à utiliser ces paramètres ici et là, ce qui évite, si vous devez en modifier la valeur, d'intervenir en de multiples endroits de votre code. À titre d'exemple, vous pouvez définir la taille des vignettes d'images en tant que paramètre de la façon suivante :

```
[
    'params' => [
        'thumbnail.size' => [128, 128],
    ],
]
```

puis dans votre code, là où vous devez utiliser cette taille, procéder de la façon suivante :

```
$size = \Yii::$app->params['thumbnail.size'];
$width = \Yii::$app->params['thumbnail.size'][0];
```

Plus tard, si vous changez d'avis à propos de la taille de ces vignettes, il vous suffit de modifier la valeur du paramètre dans la configuration de l'application sans avoir à toucher à votre code.

`sourceLanguage (langue source)` Cette propriété spécifie la langue dans laquelle l'application est écrite. La valeur par défaut est `'en-US'`, pour (anglais — États-Unis). Vous devriez configurer cette propriété si les textes dans votre code ne sont pas en anglais US.

Comme pour la propriété language (langue), vous devez configurer cette propriété à l'aide d'une étiquette IETF d'identification de langue [4]. Par exemple, `en` signifie `anglais`, tandis que `en-US` signifie for `anglais-États-Unis`).

Pour plus d'informations sur cette propriété, reportez-vous à la section Internationalisation.

`timeZone (fuseau horaire)` Cette propriété est fournie comme une manière alternative de définir le fuseau horaire par défaut au moment de l'exécution du script PHP. En configurant cette propriété, vous ne faites essentiellement qu'appeler la fonction PHP date_default_timezone_set() [5]. Par exemple :

```
[
    'timeZone' => 'America/Los_Angeles',
]
```

4. https://fr.wikipedia.org/wiki/%C3%89tiquette_d%27identification_de_langues_IETF
5. https://www.php.net/manual/fr/function.date-default-timezone-set.php

`version` Cette propriété spécifie la version de l'application. Sa valeur par défaut est `'1.0'`. Il n'est pas nécessaire que vous définissiez cette propriété si vous ne l'utilisez pas dans votre code.

Propriétés utiles

Les propriétés décrites dans cette sous-section ne sont en général pas spécifiées car leur valeur par défaut dérive de conventions ordinaires. Néanmoins, vous pouvez les spécifier pour outrepasser les conventions.

`charset (jeu de caractères)` Cette propriété spécifie le jeu de caractères que l'application utilise. La valeur par défaut est `'UTF-8'`, qui devrait être gardée telle quelle dans la plupart des applications sauf si vous travaillez avec un système ancien qui utilise de nombreuses données non Unicode.

`defaultRoute (route par défaut)` Cette propriété spécifie la route qu'une application devrait utiliser lorsqu'une requête n'en spécifie aucune. La route peut être constituée à partir d'un identifiant de module, d'un identifiant de contrôleur et/ou d'un identifiant d'action. Par exemple, `help`, `post/create` ou `admin/post/create`. Si un identifiant d'action n'est pas fourni, cette propriété prend la valeur par défaut spécifiée dans `yii\base\Controller::$defaultAction`

Pour les `applications Web`, la valeur par défaut de cette propriété est `'site'`, ce qui donne le contrôleur `SiteController` et son action par défaut est utilisée. En conséquence, si vous accédez à l'application sans spécifier de route, vous aboutissez à ce que retourne l'action `app\controllers\SiteController::actionIndex()`.

Pour les `applications de console`, la valeur par défaut est `'help'` (aide), ce qui conduit à `yii\console\controllers\HelpController::actionIndex()`. Par conséquent, si vous exécutez la commande `yii` sans lui fournir d'argument, l'application affiche l'information d'aide.

`extensions` Cette propriété spécifie la liste des extensions installées et utilisées par l'application. Par défaut, elle reçoit le tableau retourné par le fichier `@vendor/yiisoft/extensions.php`. Le fichier `extensions.php` est généré et maintenu automatiquement lorsque vous faites appel à Composer [6] pour installer des extensions. Ainsi, dans la plupart des cas, vous n'avez pas besoin de spécifier cette propriété.

Dans le cas particulier où vous souhaitez maintenir les extensions à la main, vous pouvez configurer cette propriété de la manière suivante :

```
[
    'extensions' => [
        [
```

6. https://getcomposer.org

```
            'name' => 'extension name', //nom de l'extension
            'version' => 'version number',//numéro de version
            'bootstrap' => 'BootstrapClassName',  // facultatif, peut aussi
            être un tableau de configuration
            'alias' => [  // facultatif
                '@alias1' => 'vers/chemin1',
                '@alias2' => 'vers/chemin2',
            ],
        ],

        // ... configuration d'autres extensions similaires à ce qui précède
        ...

    ],
]
```

Comme vous pouvez le constater, la propriété reçoit un tableau de spécifications d'extension. Chacune des extensions est spécifiée par un tableau constitué du nom (`name`) et de la `version` de l'extension. Si une extension doit être exécutée durant le processus d'amorçage, un élément `bootstrap` doit être spécifié par un nom de classe d'amorçage (`bootstrap`) ou un tableau de configuration. Une extension peut aussi définir quelques alias.

`layout (disposition de page)` Cette propriété spécifie le nom de la disposition de page par défaut (`layout`) qui doit être utilisée lors du rendu d'une vue. La valeur par défaut est `'main'`, ce qui signifie que le fichier de disposition de page `main.php` sous le chemin layout path est utilisé. Si, à la fois, le chemin de la disposition de page layout path et le chemin de la vue view path prennent leur valeur par défaut, le fichier de disposition de page par défaut peut être représenté par l'alias `@app/views/layouts/main.php`.

Vous pouvez définir cette propriété à la valeur `false` pour désactiver la disposition de page par défaut, bien que cela se fasse rarement.

`layoutPath (chemin de la disposition de page)` Cette propriété spécifie le chemin du dossier où rechercher les fichiers de disposition de page. La valeur par défaut `layouts` correspond à un sous-dossier de view path. Si view path prend sa valeur par défaut, le chemin de la disposition de page par défaut peut être représenté par l'alias `@app/views/layouts`.

Vous pouvez le définir comme un dossier ou un alias de chemin.

`runtimePath (chemin du dossier d'exécution)` Cette propriété spécifie le chemin du dossier où les fichiers temporaires, tels que les journaux et les fichiers de cache, sont placés. La valeur par défaut est `@app/runtime`.

Vous pouvez configurer cette propriété comme un dossier ou un alias de chemin. Notez que le dossier d'exécution `runtimePath` doit être accessible en écriture par le processus qui exécute l'application et rendu inaccessible aux

utilisateurs finaux, parce que les fichiers temporaires qu'il contient peuvent contenir des informations sensibles.

Pour simplifier l'accès à ce chemin, Yii a prédéfini un alias de chemin nommé `@runtime`.

`viewPath (chemin des vues)` Cette propriété spécifie le dossier racine des fichiers de vues. La valeur par défaut est le dossier représenté par l'alias `@app/views`. Vous pouvez le définir sous forme de dossier ou comme un alias de chemin.

`vendorPath (chemin des vendeurs)` Cette propriété spécifie le dossier des vendeurs gérés par Composer[7]. Il contient toutes les bibliothèques de tierces parties utilisées par l'application, y compris le *framework* Yii. La valeur par défaut est le dossier représenté par `@app/vendor`.

Vous pouvez configurer cette propriété comme un dossier ou un alias de chemin. Lorsque vous modifiez cette propriété, assurez-vous d'ajuster la configuration de Composer en conséquence.

Pour simplifier l'accès à ce chemin, Yii a prédéfini un alias de chemin nommé `@vendor`.

`enableCoreCommands (activer les commandes du noyau)` Cette propriété est prise en charge par les `applications de console` uniquement. Elle spécifie si les commandes du noyau de la version de Yii sont activées ou pas. La valeur par défaut est `true` (vrai).

3.3.3 Événements d'application

Une application déclenche plusieurs événements tout au long de son cycle de vie pour prendre en compte une requête. Vous pouvez attacher des gestionnaires d'événement à ces événements dans la configuration de l'application de la manière suivante :

```
[
    'on beforeRequest' => function ($event) {
        // ...
    },
]
```

L'utilisation de la syntaxe `on eventName` (on Non d'événement) est décrite dans la section Configurations.

En alternative, vous pouvez attacher les gestionnaires d'événement lors du processus d'amorçage après que l'objet Application a été instancié. Par exemple :

7. https://getcomposer.org

```
\Yii::$app->on(\yii\base\Application::EVENT_BEFORE_REQUEST, function
($event) {
    // ...
});
```

EVENT_BEFORE_REQUEST

Cette événement est déclenché *avant* que l'application ne prenne la requête en charge. Le nom réel de l'événement est `beforeRequest`.

Lorsque cet événement est déclenché, l'objet Application a été configuré et initialisé. C'est donc un bon endroit pour insérer votre code personnalisé via le mécanisme événementiel pour intercepter le processus de prise en charge de la requête. Par exemple, dans le gestionnaire d'événement, vous pouvez définir dynamiquement la propriété [[yii\base\Application : :language (langue)]] en fonction de certains paramètres.

EVENT_AFTER_REQUEST

Cet événement est déclenché *après* que l'application a fini de prendre la requête en charge mais *avant* que la réponse ne soit envoyée. Le nom réel de l'événement est `afterRequest`.

Lorsque cet événement est déclenché, la prise en charge de la requête est terminée et vous pouvez profiter de cette opportunité pour effectuer quelques post-traitements de la requête et personnaliser la réponse.

Notez que le composant `response (réponse)` déclenche également quelques événements tandis qu'il envoie la réponse au navigateur. Ces événements sont déclenchés *après* cet événement.

EVENT_BEFORE_ACTION

Cet événement est déclenché *avant* d'exécuter toute action de contrôleur. Le nom réel de l'événement est `beforeAction`. Le paramètre de l'événement est une instance de `yii\base\ActionEvent`. Un gestionnaire d'événement peut définir la propriété [[yii\base\ActionEvent : :isValid (est valide)]] à `false` pour arrêter l'exécution de l'action. Par exemple :

```
[
    'on beforeAction' => function ($event) {
        if (some condition) {
            $event->isValid = false;
        } else {
        }
    },
]
```

Notez que le même événement `beforeAction` est également déclenché par les modules et les contrôleurs.L'objet *Application* est le premier à déclencher cet

événement, suivis des modules (s'il en existe) et, pour finir, des contrôleurs. Si un gestionnaire d'événement défini la propriété `yii\base\ActionEvent::$isValid` à `false`, tous les événements qui devraient suivre ne sont PAS déclenchés.

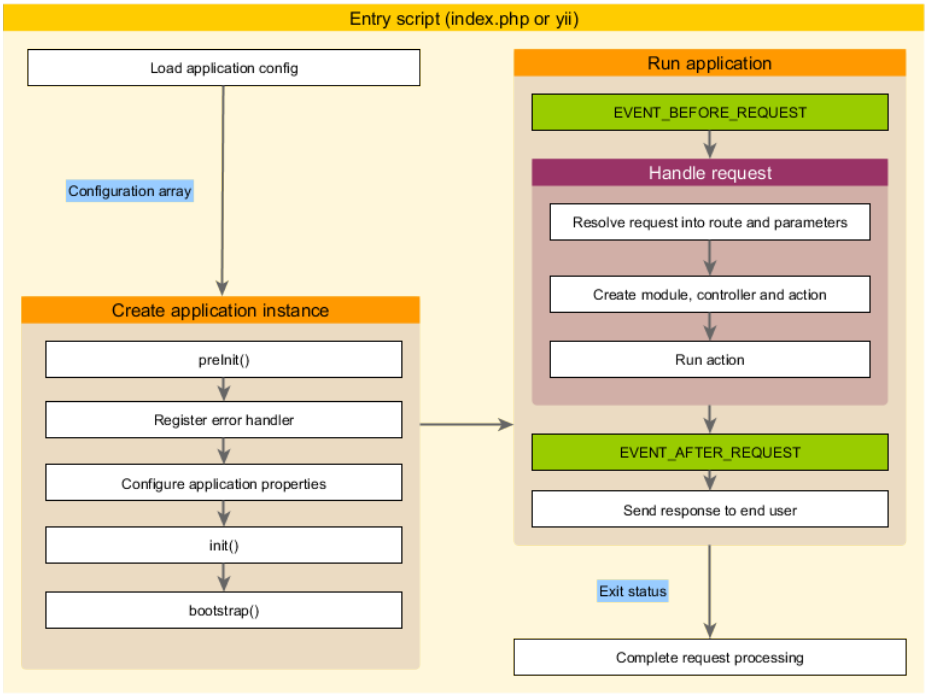
`EVENT_AFTER_ACTION`

Cet événement est déclenché *après* que chacune des actions de contrôleur a été exécutée. Le paramètre de l'événement est `yii\base\ActionEvent`. Un gestionnaire d'événement peut accéder au résultat de l'action et le modifier via la propriété `yii\base\ActionEvent::$result`. Par exemple :

```
[
    'on afterAction' => function ($event) {
        if (some condition) {
            // modify $event->result
        } else {
        }
    },
]
```

Notez que le même événement `afterAction` est également déclenché par les modules et les contrôleurs. Ces objets déclenchent ces événements dans l'ordre inverse de celui des événements déclenchés par `beforeAction`. En clair, les contrôleurs sont les premiers objets à déclencher cet événement, suivis des modules (s'il en existe) et, finalement, de l'application.

3.3.4 Cycle de vie d'une application



Lorsqu'un script d'entrée est exécuté pour prendre en compte une requête, une application entame le cycle de vie suivant :

1. Le script d'entrée charge la configuration de l'application sous forme de tableau.
2. Le script d'entrée crée un nouvel objet *Application* :
   - Sa méthode `preInit()` est appelée pour configurer quelques propriétés de haute priorité de cette application, comme `basePath`.
   - Il enregistre `le gestionnaire d'erreurs`.
   - Il configure les propriétés de l'application.
   - Sa méthode `init()` est appelée qui appelle ensuite la méthode `bootstrap()` pour exécuter les composants du processus d'amorçage.
3. Le script d'entrée appelle la méthode `yii\base\Application::run()` pour exécuter l'application qui :
   - déclenche l'événement `EVENT_BEFORE_REQUEST` ;
   - prend en charge la requête : résout la requête en une route et ses paramètres associés ;
   - crée le module, le contrôleur et l'action spécifiés par la route et exécute l'action ;
   - déclenche l'événement `EVENT_AFTER_REQUEST` ;
   - renvoie la réponse au navigateur.

4. Le script d'entrée reçoit l'état de sortie de l'exécution de l'application et complète le processus de prise en charge de la requête.

3.4 Composants d'application

Les applications sont des (localisateurs de services (service locators). Elles hébergent un jeu composants appelés « composants d'application » qui procurent différents services pour la prise en charge des requêtes. Par exemple, le composant `urlManager` (gestionnaire d'url) est chargé de router les requêtes Web vers les contrôleurs appropriés ; le composant `db` (base de données) fournit les services relatifs à la base de données ; et ainsi de suite.

Chaque composant d'application possède un identifiant unique qui le distingue des autres composants d'application de la même application. Vous pouvez accéder à un composant d'application via l'expression :

```
\Yii::$app->componentID
```

Par exemple, vous pouvez utiliser `\Yii::$app->db` pour obtenir la `connexion à la base de données`, et `\Yii::$app->cache` pour accéder au `cache primaire` enregistré dans l'application.

Un composant d'application est créé la première fois qu'on veut y accéder en utilisant l'expression ci-dessus. Les accès ultérieurs retournent la même instance du composant.

Les composants d'application peuvent être n'importe quel objet. Vous pouvez les enregistrer en configurant la propriété `yii\base\Application::$components` dans la configuration de l'application.

Par exemple,

```
[
    'components' => [
        // enregistre le composant  "cache" à partir du nom de classe
        'cache' => 'yii\caching\ApcCache',

        // enregistre le composant "db" à l'aide d'un tableau de
        configuration
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],

        // enregistre le composant "search" en utilisant une fonction
        anonyme
        'search' => function () {
            return new app\components\SolrService;
        },
    ],
]
```

> **Info :** bien que vous puissiez enregistrer autant de composants d'application que vous le désirez, vous devriez le faire avec discernement. Les composants d'application sont comme les variables globales, une utilisation trop importante de composants d'application est susceptible de rendre votre code plus difficile à tester et à maintenir. Dans beaucoup de cas, vous pouvez simplement créer un composant localement et l'utiliser lorsque vous en avez besoin.

3.4.1 Composants du processus d'amorçage

Comme il a été dit plus haut, un composant d'application n'est instancié que lorsqu'on y accède pour la première fois. S'il n'est pas du tout accédé dans le traitement de la requête, il n'est pas instancié. Parfois, vous désirez peut être instancier un composant d'application pour chacune des requêtes, même s'il n'est pas explicitement accédé. Pour cela, vous pouvez lister son identifiant (ID) dans la propriété `bootstrap` de l'application.

Vous pouvez également utiliser des « Closures » (Fermetures) pour amorcer des composants personnalisés. Il n'est pas nécessaire de retourner une instance de composant. Une « Closure » peut également être utilisée pour exécuter du code après l'instanciation de `yii\base\Application`.

Par exemple, la configuration d'application suivante garantit que le composant `log` est toujours chargé.

```
[
    'bootstrap' => [
        'log',
        function($app){
            return new ComponentX();
        },
        function($app){
            // some code
           return;
        }
    ],
    'components' => [
        'log' => [
            // configuration le composant "log"
        ],
    ],
]
```

3.4.2 Composants d'application du noyau

Yii définit un jeu de composants d'application dit *core application components* (composants d'application du noyau ou du cœur) avec des identifiants fixés et des configurations par défaut. Par exemple, le composant `request`

`(requête)` est utilisé pour collecter les informations sur une requête utilisateur et la résoudre en une route ; le composant `db (base de données)` représente une connexion à une base de données à l'aide de laquelle vous pouvez effectuer des requêtes de base de données. C'est à l'aide des ces composants d'application du noyau que les applications Yii sont en mesure de prendre en charge les requêtes des utilisateurs.

Vous trouverez ci-après la liste des composants d'application prédéfinis du noyau. Vous pouvez les configurer et les personnaliser comme tout composant d'application. Lorsque vous configurez une composant d'application du noyau, vous n'avez pas besoin de spécifier sa classe, celle par défaut est utilisée.

- `assetManager (gestionnaire de ressources` : gère les paquets de ressources et la publication des ressources. Reportez-vous à la section Ressources pour plus de détails.
- `db (base de données)` : représente une connexion à une base de données à l'aide de laquelle vous pouvez effectuer des requêtes de base de données. Notez que lorsque vous configurez ce composant, vous devez spécifier la classe de composant tout comme les autres propriétés de composant, telle que `yii\db\Connection::$dsn`. Reportez-vous à la section Objets d'accès aux bases de données pour plus de détails.
- `errorHandler (gestionnaire d'erreurs)` : gère les erreurs PHP et les exceptions. Reportez-vous à la section Gestion des erreurs pour plus de détails.
- `formatter` : formate les données lorsqu'elles sont présentées à l'utilisateur final. Par exemple, un nombre peut être affiché avec un séparateur de milliers, une date affichée dans un format long, etc. Reportez-vous à la section Formatage des données pour plus de détails.
- `i18n` : prend en charge la traduction et le formatage des messages. Reportez-vous à la section Internationalisation pour plus de détails.
- `log` : gère les journaux cibles. Reportez-vous à la section Journaux pour plus de détails.
- `yii\swiftmailer\Mailer` : prend en charge la composition et l'envoi des courriels. Reportez-vous à la section Mailing pour plus de détails.
- `response` : représente la réponse qui est adressée à l'utilisateur final. Reportez-vous à la section Réponses pour plus de détails.
- `request` : représente la requête reçue de l'utilisateur final. Reportez-vous à la section Requests pour plus de détails.
- `session` : représente les informations de session. Ce composant n'est disponible que dans les `applications Web`. Reportez-vous à la section Sessions et Cookies pour plus de détails.
- `urlManager (gestionnaire d'url)` : prend en charge l'analyse des URL et leur création. Reportez-vous à la section Routage et création d'URL pour plus de détails.
- `user` : représente les informations d'authentification de l'utilisateur. Ce

composant n'est disponible que dans les `applications Web`. Reportez-vous à la section Authentification pour plus de détails.
— `view` : prend en charge le rendu des vues. Reportez-vous à la section Vues pour plus de détails.

3.5 Contrôleurs

Les contrôleurs font partie du modèle d'architecture MVC[8] (Modèle Vue Contrôleur). Ce sont des objets dont la classe étend `yii\base\Controller`. Ils sont chargés de traiter les requêtes et de générer les réponses. En particulier, après que l'objet application leur a passé le contrôle, ils analysent les données de la requête entrante, les transmettent aux modèles, injectent le résultat des modèles dans les vues et, pour finir, génèrent les réponses sortantes.

3.5.1 Actions

Les contrôleurs sont constitués d'*actions* qui sont les unités les plus élémentaires dont l'utilisateur final peut demander l'exécution. Un contrôleur comprend une ou plusieurs actions.

L'exemple qui suit présente un contrôleur `post` avec deux actions : `view` et `create` :

```
namespace app\controllers;

use Yii;
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);
        if ($model === null) {
            throw new NotFoundHttpException;
        }

        return $this->render('view', [
            'model' => $model,
        ]);
    }

    public function actionCreate()
    {
        $model = new Post;
```

8. https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur

```
        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {
            return $this->render('create', [
                'model' => $model,
            ]);
        }
    }
}
```

Dans l'action `view` (définie par la méthode `actionView()`), le code commence par charger le modèle en fonction de l'identifiant (ID) du modèle requis. Si le chargement du modèle réussit, l'action l'affiche en utilisant une vue nommée `view`. Autrement, elle lève une exception.

Dans l'action `create` (définie par le méthode `actionCreate()`), le code est similaire. Elle commence par essayer de peupler une nouvelle instance du modèle avec les données de la requête et sauvegarde le modèle. Si les deux opérations réussissent, elle redirige le navigateur vers l'action `view` en lui passant l'identifiant (ID) du nouveau modèle. Autrement, elle affiche la vue `create` dans laquelle l'utilisateur peut saisir les entrées requises.

3.5.2 Routes

L'utilisateur final demande l'exécution des actions via ce qu'on appelle des *routes*. Une route est une chaîne de caractères constituée des parties suivantes :

- un identifiant (ID) de module : cette partie n'est présente que si le contrôleur appartient à un module qui n'est pas en soi une application ;
- un identifiant de contrôleur : une chaîne de caractères qui distingue le contrôleur des autres contrôleurs de la même application — ou du même module si le contrôleur appartient à un module ;
- un identifiant d'action : une chaîne de caractères qui distingue cette action des autres actions du même contrôleur.

Les routes se présentent dans le format suivant :

```
identifiant_de_contrôleur/identifiant_d_action
```

ou dans le format suivant si le contrôleur appartient à un module :

```
identifiant_de_module/identifiant_de_contrôleur/identifiant_d_action
```

Ainsi si un utilisateur requiert l'URL `https://hostname/index.php?r=site/index`, l'action `index` dans le contrôleur `site` sera exécutée. Pour plus de détails sur la façon dont les routes sont résolues, reportez-vous à la section Routage et génération d'URL.

3.5.3 Création des contrôleurs

Dans les `applications Web`, les contrôleur doivent étendre la classe `yii\web\Controller` ou ses classes filles. De façon similaire, dans les `applications de console`, les contrôleurs doivent étendre la classe `yii\console\Controller` ou ses classes filles. Le code qui suit définit un contrôleur nommé `site` :

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
}
```

Identifiant des contrôleurs

Généralement, un contrôleur est conçu pour gérer les requêtes concernant un type particulier de ressource. Pour cette raison, l'identifiant d'un contrôleur est souvent un nom faisant référence au type de ressources que ce contrôleur gère. Par exemple, vous pouvez utiliser `article` comme identifiant d'un contrôleur qui gère des données d'articles.

Par défaut, l'identifiant d'un contrôleur ne peut contenir que les caractères suivants : lettres de l'alphabet anglais en bas de casse, chiffres, tiret bas, trait d'union et barre oblique de division. Par exemple, `article` et `post-comment` sont tous deux des identifiants de contrôleur valides, tandis que `article?`, `PostComment` et `admin\post` ne le sont pas. Un identifiant de contrôleur peut aussi contenir un préfixe de sous-dossier. Par exemple `admin/article` représente un contrôleur `article` dans le dossier `admin` dans l'`espace de noms du contrôleur`. Les caractères valides pour le préfixe des sous-dossiers incluent : les lettres de l'alphabet anglais dans les deux casses, les chiffres, le tiret bas et la barre oblique de division, parmi lesquels les barres obliques de division sont utilisées comme séparateurs pour les sous-dossiers à plusieurs niveaux (p. ex. `panels/admin`).

Nommage des classes de contrôleur

Les noms de classe de contrôleur peut être dérivés de l'identifiant du contrôleur selon la procédure suivante :

1. Mettre la première lettre de chacun des mots séparés par des trait d'union en capitale. Notez que si l'identifiant du contrôleur contient certaines barres obliques, cette règle ne s'applique qu'à la partie après la dernière barre oblique dans l'identifiant.
2. Retirer les traits d'union et remplacer toute barre oblique de division par une barre oblique inversée.
3. Ajouter le suffixe `Controller`.

4. Préfixer avec l'`espace de noms du contrôleur`.

Ci-après sont présentés quelques exemples en supposant que l'`espace de noms du contrôleur` prend la valeur par défaut, soit `app\controllers` :

— `article` donne `app\controllers\ArticleController` ;
— `post-comment` donne `app\controllers\PostCommentController` ;
— `admin/post-comment` donne `app\controllers\admin\PostCommentController` ;
— `adminPanels/post-comment` donne `app\controllers\adminPanels\PostCommentController`.

Les classes de contrôleur doivent être auto-chargeables. Pour cette raison, dans les exemples qui précèdent, la classe de contrôleur `article` doit être sauvegardée dans le fichier dont l'alias est `@app/controllers/ArticleController.php` ; tandis que la classe de contrôleur `admin/post-comment` doit se trouver dans `@app/controllers/admin/PostCommentController.php`.

> **Info :** dans le dernier exemple, `admin/post-comment` montre comment placer un contrôleur dans un sous-dossier de l'`espace de noms du contrôleur`. Cela est utile lorsque vous voulez organiser vos contrôleurs en plusieurs catégories et que vous ne voulez pas utiliser de modules.

Table de mise en correspondance des contrôleurs

Vous pouvez configurer `controller map (table de mise en correspondance des contrôleurs)` pour outrepasser les contraintes concernant les identifiants de contrôleur et les noms de classe décrites plus haut. Cela est principalement utile lorsque vous utilisez des contrôleurs de tierces parties et que vous n'avez aucun contrôle sur le nommage de leur classe. Vous pouvez configurer `controller map` dans la configuration de l'application. Par exemple :

```
[
    'controllerMap' => [
        // declares "account" controller using a class name
        'account' => 'app\controllers\UserController',

        // declares "article" controller using a configuration array
        'article' => [
            'class' => 'app\controllers\PostController',
            'enableCsrfValidation' => false,
        ],
    ],
]
```

Contrôleur par défaut

Chaque application possède un contrôleur par défaut spécifié via la propriété `yii\base\Application::$defaultRoute`. Lorsqu'une requête ne précise aucune route, c'est la route spécifiée par cette propriété qui est utilisée. Pour

les `applications Web`, sa valeur est `'site'`, tandis que pour les `applications de console`, c'est `help`. Par conséquent, si une URL est de la forme `https://hostname/index.php`, c'est le contrôleur `site` qui prend la requête en charge.

Vous pouvez changer de contrôleur par défaut en utilisant la configuration d'application suivante :

```
[
    'defaultRoute' => 'main',
]
```

3.5.4 Création d'actions

Créer des actions est aussi simple que de définir ce qu'on appelle des *méthodes d'action* dans une classe de contrôleur. Une méthode d'action est une méthode *publique* dont le nom commence par le mot `action`. La valeur retournée par une méthode d'action représente les données de la réponse à envoyer à l'utilisateur final. Le code qui suit définit deux actions, `index` et `hello-world` :

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actionIndex()
    {
        return $this->render('index');
    }

    public function actionHelloWorld()
    {
        return 'Hello World';
    }
}
```

Identifiants d'action

Une action est souvent conçue pour effectuer une manipulation particulière d'une ressource. Pour cette raison, les identifiants d'action sont habituellement des verbes comme `view` (voir), `update` (mettre à jour), etc.

Par défaut, les identifiants d'action ne doivent contenir rien d'autre que les caractères suivants : les lettres de l'alphabet anglais en bas de casse, les chiffres, le tiret bas et le trait d'union. Vous pouvez utiliser le trait d'union pour séparer les mots. Par exemple : `view`, `update2`, et `comment-post` sont des identifiants d'action valides, tandis que `view?` et `Update` ne le sont pas.

Vous pouvez créer des actions sous deux formes : les actions en ligne (*inline*) et les actions autonomes (*standalone*). Une action en ligne est définie

en tant que méthode dans un contrôleur, alors qu'une action autonome est une classe qui étend la classe `yii\base\Action` ou une des ses classes filles. La définition d'une action en ligne requiert moins d'efforts et est souvent préférée lorsqu'il n'y a pas d'intention de réutiliser cette action. Par contre, les actions autonomes sont essentiellement créées pour être utilisées dans différents contrôleurs ou pour être redistribuées dans des extensions.

Actions en ligne

Les actions en ligne sont les actions qui sont définies en terme de méthodes d'action comme nous l'avons décrit plus haut.

Les noms des méthodes d'action sont dérivés des identifiants d'action selon la procédure suivante :

1. Mettre la première lettre de chaque mot de l'identifiant en capitale.
2. Supprimer les traits d'union.
3. Préfixer le tout par le mot `action`.

Par exemple, `index` donne `actionIndex`, et `hello-world` donne `actionHelloWorld`.

> **Note :** les noms des méthodes d'action sont *sensibles à la casse*. Si vous avez une méthode nommée `ActionIndex`, elle ne sera pas considérée comme étant une méthode d'action et, par conséquent, la requête de l'action `index` aboutira à une exception. Notez également que les méthodes d'action doivent être publiques. Une méthode privée ou protégée ne définit PAS une action en ligne.

Les actions en ligne sont les actions les plus communément définies parce qu'elle ne requièrent que peu d'efforts pour leur création. Néanmoins, si vous envisagez de réutiliser la même action en différents endroits, ou si vous voulez redistribuer cette action, vous devriez envisager de la définir en tant qu'*action autonome*.

Actions autonomes

Les actions autonomes sont définies comme des classes d'action qui étendent la classe `yii\base\Action` ou une de ses classes filles. Par exemple, dans les versions de Yii, il y a `yii\web\ViewAction` et `yii\web\ErrorAction`, qui sont toutes les deux des actions autonomes.

Pour utiliser une action autonome, vous devez la déclarer dans la *table de mise en correspondance des actions* en redéfinissant les méthodes de la classe `yii\base\Controller::actions()` dans la classe de votre contrôleur de la manière suivante :

```
public function actions()
{
    return [
```

```
        // déclare une action "error" en utilisant un nom de classe
        'error' => 'yii\web\ErrorAction',

        // déclare une action  "view" action en utilisant un tableau de
        configuration
        'view' => [
            'class' => 'yii\web\ViewAction',
            'viewPrefix' => '',
        ],
    ];
}
```

Comme vous pouvez l'observer, les méthodes `actions()` doivent retourner un tableau dont les clés sont les identifiants d'action et les valeurs le nom de la classe d'action correspondant ou des tableaux de configuration. Contrairement aux actions en ligne, les identifiants d'action autonomes peuvent comprendre n'importe quels caractères du moment qu'ils sont déclarés dans la méthode `actions()`.

Pour créer une classe d'action autonome, vous devez étendre la classe `yii\base\Action` ou une de ses classes filles, et implémenter une méthode publique nommée `run()`. Le rôle de la méthode `run()` est similaire à celui d'une méthode d'action. Par exemple :

```
<?php
namespace app\components;

use yii\base\Action;

class HelloWorldAction extends Action
{
    public function run()
    {
        return "Hello World";
    }
}
```

Valeur de retour d'une action

Le valeur de retour d'une méthode d'action, ou celle de la méthode `run()` d'une action autonome, représente le résultat de l'action correspondante.

La valeur de retour peut être un objet response qui sera transmis à l'utilisateur final en tant que réponse.

— Pour les `applications Web`, la valeur de retour peut également être des données arbitraires qui seront assignées à l'objet `yii\web\Response::$data` et converties ensuite en une chaîne de caractères représentant le corps de la réponse.

— Pour les `applications de console`, la valeur de retour peut aussi être un entier représentant l'`état de sortie` de l'exécution de la commande.

Dans les exemples ci-dessus, les valeurs de retour des actions sont toutes des chaînes de caractères qui seront traitées comme le corps de la réponse envoyée à l'utilisateur final. Les exemples qui suivent montrent comment une action peut rediriger le navigateur vers une nouvelle URL en retournant un objet *response* (parce que la méthode `redirect()` retourne un objet *response*) :

```
public function actionForward()
{
    // redirect the user browser to https://example.com
    return $this->redirect('https://example.com');
}
```

Paramètres d'action

Les méthodes d'action pour les actions en ligne et la méthode `run()` d'une action autonome acceptent des paramètres appelés *paramètres d'action*. Leurs valeurs sont tirées des requêtes. Pour les `applications Web`, la valeur de chacun des paramètres d'action est obtenue de la méthode `$_GET` en utilisant le nom du paramètre en tant que clé. Pour les `applications de console`, les valeurs des paramètres correspondent aux argument de la commande. Dans d'exemple qui suit, l'action `view` (une action en ligne) déclare deux paramètres : `$id` et `$version`.

```
namespace app\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    public function actionView($id, $version = null)
    {
        // ...
    }
}
```

En fonction de la requête, les paramètres de l'action seront établis comme suit :

- `https://hostname/index.php?r=post/view&id=123` : le paramètre `$id` reçoit la valeur `'123'`, tandis que le paramètre `$version` reste `null` (sa valeur par défaut) car la requête ne contient aucun paramètre `version`.
- `https://hostname/index.php?r=post/view&id=123&version=2` : les paramètres `$id` et `$version` reçoivent les valeurs `'123'` et `'2'`, respectivement.
- `https://hostname/index.php?r=post/view` : une exception `yii\web\BadRequestHttpException` est levée car le paramètre obligatoire `$id` n'est pas fourni par la requête.
- `https://hostname/index.php?r=post/view&id[]=123` : une exception `yii\web\BadRequestHttpException` est levée car le paramètre `$id` reçoit, de manière inattendue, un tableau (`['123']`).

Si vous voulez que votre paramètre d'action accepte un tableau, il faut, dans la définition de la méthode, faire allusion à son type, avec `array`, comme ceci :

```
public function actionView(array $id, $version = null)
{
    // ...
}
```

Désormais, si la requête est `https://hostname/index.php?r=post/view&id[]=123`, le paramètre `$id` accepte la valeur `['123']`. Si la requête est `https://hostname/index.php?r=post/view&id=123`, le paramètre `$id` accepte également la valeur transmise par la requête parce que les valeurs scalaires sont automatiquement convertie en tableau (*array*).

Les exemples qui précèdent montrent essentiellement comment les paramètres d'action fonctionnent dans les applications Web. Pour les applications de console, reportez-vous à la section Commandes de console pour plus de détails.

Action par défaut

Chaque contrôleur dispose d'une action par défaut spécifiée par la propriété `yii\base\Controller::$defaultAction`. Lorsqu'une route ne contient que l'identifiant du contrôleur, cela implique que l'action par défaut de ce contrôleur est requise.

Par défaut, l'action par défaut est définie comme étant `index`. Si vous désirez changer cette valeur par défaut, contentez-vous de redéfinir cette propriété dans la classe du contrôleur, comme indiqué ci-après :

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $defaultAction = 'home';

    public function actionHome()
    {
        return $this->render('home');
    }
}
```

3.5.5 Cycle de vie d'un contrôleur

Lors du traitement d'une requête, une application crée un contrôleur en se basant sur la route requise. Le contrôleur entame alors le cycle de vie suivant pour satisfaire la requête :

1. La méthode `yii\base\Controller::init()` est appelée après que le contrôleur est créé et configuré.

2. Le contrôleur crée un objet *action* en se basant sur l'identifiant d'action de la requête :
   — Si l'identifiant de l'action n'est pas spécifié, l'`identifiant de l'action par défaut` est utilisé.
   — Si l'identifiant de l'action est trouvé dans la `table de mise en correspondance des actions`, une action autonome est créée.
   — Si l'identifiant de l'action est trouvé et qu'il correspond à une méthode d'action, une action en ligne est créée.
   — Dans les autres cas, une exception `yii\base\InvalidRouteException` est levée.
3. Le contrôleur appelle consécutivement la méthode `beforeAction()` de l'application, celle du module (si module si le contrôleur appartient à un module) et celle du contrôleur.
   — Si l'un des appels retourne `false`, les appels aux méthodes `beforeAction()` qui devraient suivre ne sont pas effectués et l'exécution de l'action est annulée.
   — Par défaut, chacun des appels à la méthode `beforeAction()` déclenche un événement `beforeAction` auquel vous pouvez attacher un gestionnaire d'événement.
4. Le contrôleur exécute l'action.
   — Les paramètres de l'action sont analysés et définis à partir des données transmises par la requête.
5. Le contrôleur appelle successivement la méthode `afterAction()` du contrôleur, du module (si le contrôleur appartient à un module) et de l'application.
   — Par défaut, chacun des appels à la méthode `afterAction()` déclenche un événement `afterAction` auquel vous pouvez attacher un gestionnaire d'événement.
6. L'application assigne le résultat de l'action à l'objet response.

3.5.6 Meilleures pratiques

Dans une application bien conçue, les contrôleurs sont souvent très légers avec des actions qui ne contiennent que peu de code. Si votre contrôleur est plutôt compliqué, cela traduit la nécessité de remanier le code pour en déplacer certaines parties dans d'autres classes.

Voici quelques meilleures pratiques spécifiques. Les contrôleurs :

— peuvent accéder aux données de la requête ;
— peuvent appeler les méthodes des modèles et des autres composants de service avec les données de la requête ;
— peuvent utiliser des vues pour composer leurs réponses ;
— ne devraient PAS traiter les données de la requête — cela devrait être fait dans la couche modèle ;

— devraient éviter d'encapsuler du code HTML ou tout autre code relatif à la présentation — cela est plus avantageusement fait dans les vues.

3.6 Modèles

Les modèles font partie du modèle d'architecture MVC[9] (Modèle Vue Contrôleur). Ces objets représentent les données à traiter, les règles et la logique de traitement.

Vous pouvez créer des classes de modèle en étendant la classe `yii\base\Model` ou ses classe filles. La classe de base `yii\base\Model` prend en charge des fonctionnalités nombreuses et utiles :

— Les attributs : ils représentent les données à traiter et peuvent être accédés comme des propriétés habituelles d'objets ou des éléments de tableaux.
— Les étiquettes d'attribut : elles spécifient les étiquettes pour l'affichage des attributs.
— L'assignation massive : elle permet l'assignation de multiples attributs en une seule étape.
— Les règles de validation : elles garantissent la validité des données saisies en s'appuyant sur des règles de validation déclarées.
— L'exportation des données : elle permet au modèle de données d'être exporté sous forme de tableaux dans des formats personnalisables.

La classe `Model` est également la classe de base pour des modèles plus évolués, comme la classe class Active Record (enregistrement actif). Reportez-vous à la documentation ad hoc pour plus de détails sur ces modèles évolués.

> **Info :** vous n'êtes pas forcé de baser vos classes de modèle sur la classe `yii\base\Model`. Néanmoins, comme il y a de nombreux composants de Yii conçus pour prendre en charge la classe `yii\base\Model`, il est généralement préférable de baser vos modèles sur cette classe.

3.6.1 Attributs

Les modèles représentent les données de l'application en termes d'attributs. Chaque attribut est comme un propriété publiquement accessible d'un modèle. La méthode `yii\base\Model::attributes()` spécifie quels attributs une classe de modèle possède.

Vous pouvez accéder à un attribut comme vous accédez à une propriété d'un objet ordinaire :

```
$model = new \app\models\ContactForm;
```

9. https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur

```php
// "name" is an attribute of ContactForm
$model->name = 'example';
echo $model->name;
```

Vous pouvez également accéder aux attributs comme aux éléments d'un tableau, grâce à la prise en charge de ArrayAccess [10] et ArrayIterator [11] par la classe `yii\base\Model` :

```php
$model = new \app\models\ContactForm;

// accès aux attributs comme à des éléments de tableau
$model['name'] = 'example';
echo $model['name'];

// itération sur les attributs avec foreach
foreach ($model as $name => $value) {
    echo "$name: $value\n";
}
```

Définition d'attributs

Par défaut, si votre classe de modèle étend directement la classe `yii\base\Model`, toutes ses variables membres *non statiques et publiques* sont des attributs. Par exemple, la classe de modèle `ContactForm` ci-dessous possède quatre attributs : `name`, `email`, `subject` et `body`. Le modèle `ContactForm` est utilisé pour représenter les données saisies dans un formulaire HTML.

```php
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;
}
```

Vous pouvez redéfinir la méthode `yii\base\Model::attributes()` pour spécifier les attributs d'une autre manière. La méthode devrait retourner le nom des attributs d'un modèle. Par exemple, `yii\db\ActiveRecord` fait cela en retournant le nom des colonnes de la base de données associée en tant que noms d'attribut. Notez que vous pouvez aussi avoir besoin de redéfinir les méthodes magiques telles que `__get()` et `__set()` afin que les attributs puissent être accédés comme les propriétés d'un objet ordinaire.

10. https://www.php.net/manual/en/class.arrayaccess.php
11. https://www.php.net/manual/en/class.arrayiterator.php

Étiquettes d'attribut

Lors de l'affichage de la valeur d'un attribut ou lors de la saisie d'une entrée pour une telle valeur, il est souvent nécessaire d'afficher une étiquette associée à l'attribut. Par exemple, étant donné l'attribut nommé `firstName` (prénom), vous pouvez utiliser une étiquette de la forme `First Name` qui est plus conviviale lorsqu'elle est présentée à l'utilisateur final dans un formulaire ou dans un message d'erreur.

Vous pouvez obtenir l'étiquette d'un attribut en appelant la méthode `yii\base\Model::getAttributeLabel()`. Par exemple :

```
$model = new \app\models\ContactForm;

// displays "Name"
echo $model->getAttributeLabel('name');
```

Par défaut, les étiquettes d'attribut sont automatiquement générées à partir des noms d'attribut. La génération est faite en appelant la méthode `yii\base\Model::generateAttributeLabel()`. Cette méthode transforme un nom de variable avec une casse en dos de chameau en de multiples mots, chacun commençant par une capitale. Par exemple, `username` donne `Username` et `firstName` donne `First Name`.

Si vous ne voulez pas utiliser les étiquettes à génération automatique, vous pouvez redéfinir la méthode `yii\base\Model::attributeLabels()` pour déclarer explicitement les étiquettes d'attribut. Par exemple :

```
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;

    public function attributeLabels()
    {
        return [
            'name' => 'Nom',
            'email' => 'Adresse de courriel',
            'subject' => 'Subjet',
            'body' => 'Contenu',
        ];
    }
}
```

Pour les application prenant en charge de multiples langues, vous désirez certainement traduire les étiquettes d'attribut. Cela peut être fait dans la méthode `attributeLabels()` également, en procédant comme ceci :

```
public function attributeLabels()
{
    return [
        'name' => \Yii::t('app', 'Your name'),
        'email' => \Yii::t('app', 'Your email address'),
        'subject' => \Yii::t('app', 'Subject'),
        'body' => \Yii::t('app', 'Content'),
    ];
}
```

Vous pouvez même définir les étiquettes en fonction de conditions. Par exemple, en fonction du scénario dans lequel le modèle est utilisé, vous pouvez retourner des étiquettes différentes pour le même attribut.

> **Info :** strictement parlant, les étiquettes d'attribut font partie des vues. Mais la déclaration d'étiquettes dans les modèles est souvent très pratique et conduit à un code propre et réutilisable.

3.6.2 Scénarios

Un modèle peut être utilisé dans différents *scénarios*. Par exemple, un modèle `User` (utilisateur) peut être utilisé pour collecter les données d'un utilisateur, mais il peut aussi être utilisé à des fins d'enregistrement d'enregistrement de l'utilisateur. Dans différents scénarios, un modèle peut utiliser des règles de traitement et une logique différente. Par exemple, `email` peut être nécessaire lors de l'enregistrement de l'utilisateur mais pas utilisé lors de sa connexion.

Un modèle utilise la propriété `yii\base\Model::$scenario` pour conserver un trace du scénario dans lequel il est utilisé.

Par défaut, un modèle prend en charge un unique scénario nommé `default`. Le code qui suit montre deux manières de définir le scénario d'un modèle :

```
// le scénario est défini comme une propriété
$model = new User;
$model->scenario = User::SCENARIO_LOGIN;

// le scénario est défini via une configuration
$model = new User(['scenario' => User::SCENARIO_LOGIN]);
```

Par défaut, les scénarios pris en charge par un modèle sont déterminés par les règles de validation déclarées dans le modèle. Néanmoins, vous pouvez personnaliser ce comportement en redéfinissant la méthode `yii\base\Model::scenarios()`, de la manière suivante :

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
```

```
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTER = 'register';

    public function scenarios()
    {
        return [
            self::SCENARIO_LOGIN => ['username', 'password'],
            self::SCENARIO_REGISTER => ['username', 'email', 'password'],
        ];
    }
}
```

> **Info :** dans ce qui précède et dans l'exemple qui suit, les classes de modèle étendent la classe `yii\db\ActiveRecord` parce que l'utilisation de multiples scénarios intervient ordinairement dans les classes Active Record.

La méthode `scenarios()` retourne un tableau dont les clés sont les noms de scénario et les valeurs les *attributs actifs* correspondants. Les attributs actifs peuvent être assignés massivement et doivent respecter des règles de validation. Dans l'exemple ci-dessus, les attributs `username` et `password` sont actifs dans le scénario `login`, tandis que dans le scénario `register` , `email` est, en plus de `username` et`password`, également actif.

La mise en œuvre par défaut des `scenarios()` retourne tous les scénarios trouvés dans la méthode de déclaration des règles de validation `yii\base\Model::rules()`. Lors de la redéfinition des `scenarios()`, si vous désirez introduire de nouveaux scénarios en plus des scénarios par défaut, vous pouvez utiliser un code similaire à celui qui suit :

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTER = 'register';

    public function scenarios()
    {
        $scenarios = parent::scenarios();
        $scenarios[self::SCENARIO_LOGIN] = ['username', 'password'];
        $scenarios[self::SCENARIO_REGISTER] = ['username', 'email',
        'password'];
        return $scenarios;
    }
}
```

La fonctionnalité *scénarios* est d'abord utilisée pour la validation et dans l'assignation massive des attributs. Vous pouvez, cependant l'utiliser à d'autres

fins. Par exemple, vous pouvez déclarer des étiquettes d'attribut différemment en vous basant sur le scénario courant.

3.6.3 Règles de validation

Losque les données pour un modèle sont reçues de l'utilisateur final, elles doivent être validées pour être sûr qu'elles respectent certaines règles (appelées *règles de validation*). Par exemple, étant donné un modèle pour un formulaire de contact (`ContactForm`), vous voulez vous assurer qu'aucun des attributs n'est vide et que l'attribut `email` contient une adresse de courriel valide. Si les valeurs pour certains attributs ne respectent pas les règles de validation, les messages d'erreur appropriés doivent être affichés pour aider l'utilisateur à corriger les erreurs.

Vous pouvez faire appel à la méthode `yii\base\Model::validate()` pour valider les données reçues. La méthode utilise les règles de validation déclarées dans `yii\base\Model::rules()` pour valider chacun des attributs concernés. Si aucune erreur n'est trouvée, elle retourne `true` (vrai). Autrement, les erreurs sont stockées dans la propriété `yii\base\Model::$errors` et la méthode retourne `false` (faux). Par exemple :

```
$model = new \app\models\ContactForm;

// définit les attributs du modèle avec les entrées de l'utilisateur final
$model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // toutes les entrées sont valides
} else {
    // la validation a échoué : le tableau $errors contient les messages
    d'erreur
    $errors = $model->errors;
}
```

Pour déclarer des règles de validation associées à un modèle, redéfinissez la méthode `yii\base\Model::rules()` en retournant les règles que le modèle doit respecter. L'exemple suivant montre les règles de validation déclarées pour le modèle *formulaire de contact `ContactForm` :

```
public function rules()
{
    return [
        // the name, email, subject and body attributes are required
        [['name', 'email', 'subject', 'body'], 'required'],

        // the email attribute should be a valid email address
        ['email', 'email'],
    ];
}
```

Une règle peut être utilisée pour valider un ou plusieurs attributs, et un attribut peut être validé par une ou plusieurs règles. Reportez-vous à la section validation des entrées pour plus de détails sur la manière de déclarer les règles de validation.

Parfois, vous désirez qu'une règle ne soit applicable que dans certains scénarios. Pour cela, vous pouvez spécifier la propriété `on` d'une règle, comme ci-dessous :

```
public function rules()
{
    return [
        // username, email et  password sont tous requis dans le scénario
        "register"
        [['username', 'email', 'password'], 'required', 'on' =>
        self::SCENARIO_REGISTER],

        // username et password sont requis dans le scénario "login"
        [['username', 'password'], 'required', 'on' =>
        self::SCENARIO_LOGIN],
    ];
}
```

Si vous ne spécifiez pas la propriété `on`, la règle sera appliquée dans tous les scénarios. Une règle est dite *règle active* si elle s'applique au scénario courant `scenario`.

Un attribut n'est validé que si, et seulement si, c'est un attribut actif déclaré dans `scenarios()` et s'il est associé à une ou plusieurs règles actives déclarées dans `rules()`.

3.6.4 Assignation massive

L'assignation massive est une façon pratique de peupler un modèle avec les entrées de l'utilisateur final en utilisant une seule ligne de code . Elle peuple les attributs d'un modèle en assignant directement les données d'entrée à la propriété `yii\base\Model::$attributes`. Les deux extraits de code suivants sont équivalent. Ils tentent tous deux d'assigner les données du formulaire soumis par l'utilisateur final aux attributs du modèle `ContactForm`. En clair, le premier qui utilise l'assignation massive, est plus propre et moins sujet aux erreurs que le second :

```
$model = new \app\models\ContactForm;
$model->attributes = \Yii::$app->request->post('ContactForm');

$model = new \app\models\ContactForm;
$data = \Yii::$app->request->post('ContactForm', []);
$model->name = isset($data['name']) ? $data['name'] : null;
$model->email = isset($data['email']) ? $data['email'] : null;
$model->subject = isset($data['subject']) ? $data['subject'] : null;
$model->body = isset($data['body']) ? $data['body'] : null;
```

Attributs sûr

L'assignation massive ne s'applique qu'aux attributs dits *attributs sûrs* qui sont les attributs listés dans la méthode `yii\base\Model::scenarios()` pour le `scénario` courant d'un modèle. Par exemple, si le modèle `User` contient la déclaration de scénarios suivante, alors, lorsque le scénario courant est `login`, seuls les attributs `username` et `password` peuvent être massivement assignés. Tout autre attribut n'est pas touché par l'assignation massive.

```
public function scenarios()
{
    return [
        self::SCENARIO_LOGIN => ['username', 'password'],
        self::SCENARIO_REGISTER => ['username', 'email', 'password'],
    ];
}
```

> **Info :** la raison pour laquelle l'assignation massive ne s'applique qu'aux attributs sûrs est de vous permettre de contrôler quels attributs peuvent être modifiés par les données envoyées par l'utilisateur final. Par exemple, si le modèle `User` possède un attribut `permission` qui détermine les permissions accordées à l'utilisateur, vous préférez certainement que cet attribut ne puisse être modifié que par un administrateur via l'interface d'administration seulement.

Comme la mise en œuvre par défaut de la méthode `yii\base\Model::scenarios()` retourne tous les scénarios et tous les attributs trouvés dans la méthode `yii\base\Model::rules()`, si vous ne redéfinissez pas cette méthode, cela signifie qu'un attribut est *sûr* tant qu'il apparaît dans une des règles de validation actives.

Pour cette raison, un validateur spécial dont l'alias est `safe` est fourni pour vous permettre de déclarer un attribut *sûr* sans réellement le valider. Par exemple, les règles suivantes déclarent que `title` et `description` sont tous deux des attributs sûrs.

```
public function rules()
{
    return [
        [['title', 'description'], 'safe'],
    ];
}
```

Attributs non sûr

Comme c'est expliqué plus haut, la méthode `yii\base\Model::scenarios()` remplit deux objectifs : déterminer quels attributs doivent être validés, et déterminer quels attributs sont *sûrs*. Dans certains cas, vous désirez valider

un attribut sans le marquer comme *sûr*. Vous pouvez le faire en préfixant son nom par un point d'exclamation `!` lorsque vous le déclarez dans la méthode `scenarios()`, comme c'est fait pour l'attribut `secret` dans le code suivant :

```
public function scenarios()
{
    return [
        self::SCENARIO_LOGIN => ['username', 'password', '!secret'],
    ];
}
```

Lorsque le modèle est dans le scénario `login`, les trois attributs sont validés. Néanmoins, seuls les attributs `username` et `password` peuvent être massivement assignés. Pour assigner une valeur d'entrée à l'attribut `secret`, vous devez le faire explicitement, comme montré ci-dessous :

```
$model->secret = $secret;
```

La même chose peut être faite dans la méthode `rules()` :

```
public function rules()
{
    return [
        [['username', 'password', '!secret'], 'required', 'on' => 'login']
    ];
}
```

Dans ce cas, les attributs `username`, `password` et `secret` sont requis, mais `secret` doit être assigné explicitement.

3.6.5 Exportation de données

On a souvent besoin d'exporter les modèles dans différents formats. Par exemple, vous désirez peut-être convertir une collection de modèles dans le format JSON ou dans le format Excel. Le processus d'exportation peut être décomposé en deux étapes indépendantes :

— les modèles sont convertis en tableaux,
— les tableaux sont convertis dans les formats cibles.

Vous pouvez vous concentrer uniquement sur la première étape, parce que la seconde peut être accomplie par des formateurs génériques de données, tels que `yii\web\JsonResponseFormatter`.

La manière la plus simple de convertir un modèle en tableau est d'utiliser la propriété `yii\base\Model::$attributes`. Par exemple :

```
$post = \app\models\Post::findOne(100);
$array = $post->attributes;
```

Par défaut, la propriété `yii\base\Model::$attributes` retourne les valeurs de *tous* les attributs déclarés dans la méthode `yii\base\Model::attributes()`.

Une manière plus souple et plus puissante de convertir un modèle en tableau est d'utiliser la méthode `yii\base\Model::toArray()`. Son comportement par défaut est de retourner la même chose que la propriété `yii\base\Model::$attributes`. Néanmoins, elle vous permet de choisir quelles données, appelées *champs*, doivent être placées dans le tableau résultant et comment elles doivent être formatées. En fait, c'est la manière par défaut pour exporter les modèles dans le développement d'un service Web respectant totalement l'achitecture REST, telle que décrite à la section Formatage de la réponse.

Champs

Un champ n'est rien d'autre qu'un élément nommé du tableau qui est obtenu en appelant la méthode `yii\base\Model::toArray()` d'un modèle.

Par défaut, les noms de champ sont équivalents aux noms d'attribut. Cependant, vous pouvez changer ce comportement en redéfinissant la méthode `fields()` et/ou la méthode `extraFields()`. Ces deux méthodes doivent retourner une liste de définitions de champ. Les champs définis par `fields()` sont des champs par défaut, ce qui signifie que `toArray()` retourne ces champs par défaut. La méthode `extraFields()` définit des champs additionnels disponibles qui peuvent également être retournés par `toArray()` du moment que vous les spécifiez via le paramètre `$expand`. Par exemple, le code suivant retourne tous les champs définis dans la méthode `fields()` ainsi que les champs `prettyName` et `fullAddress`, s'ils sont définis dans `extraFields()`.

```
$array = $model->toArray([], ['prettyName', 'fullAddress']);
```

Vous pouvez redéfinir la méthode `fields()` pour ajouter, retirer, renommer ou redéfinir des champs. La valeur de retour de la méthode `fields()` doit être un tableau associatif. Les clés du tableau sont les noms des champs et les valeurs sont les définitions de champ correspondantes qui peuvent être, soit des noms d'attribut/propriété, soit des fonctions anonymes retournant les valeurs de champ correspondantes. Dans le cas particulier où un nom de champ est identique à celui du nom d'attribut qui le définit, vous pouvez omettre la clé du tableau. Par exemple :

```
// liste explicitement chaque champ ; à utiliser de préférence quand vous
voulez être sûr
// que les changements dans votre table de base de données ou dans les
attributs de votre modèle
// ne créent pas de changement dans vos champs (pour conserver la
rétro-compatibilité de l'API).
public function fields()
```

```
{
    return [
        // le nom du champ est identique à celui de l'attribut
        'id',

        // le nom du champ est "email", le nom d'attribut correspondant est
        "email_address"
        'email' => 'email_address',

        // le nom du champ est  "name", sa valeur est définie par une
        fonction PHP de rappel
        'name' => function () {
            return $this->first_name . ' ' . $this->last_name;
        },
    ];
}

// filtre quelques champs ; à utiliser de préférence quand vous voulez
hériter de l'implémentation du parent
// et mettre quelques champs sensibles en liste noire.
public function fields()
{
    $fields = parent::fields();

    // retire les champs contenant des informations sensibles
    unset($fields['auth_key'], $fields['password_hash'],
    $fields['password_reset_token']);

    return $fields;
}
```

> **Attention :** étant donné que, par défaut, tous les attributs d'un modèle sont inclus dans le tableau exporté, vous devez vous assurer que vos données ne contiennent pas d'information sensible. Si de telles informations existent, vous devriez redéfinir la méthode `fields()` pour les filtrer. Dans l'exemple ci-dessus, nous avons choisi d'exclure `auth_key`, `auth_key`, `password_hash` et `password_reset_token`.

3.6.6 Meilleures pratiques

Les modèles sont les endroits centraux pour représenter les données de l'application, les règles et la logique. Ils doivent souvent être réutilisés à différents endroits. Dans une application bien conçue, les modèles sont généralement plus volumineux que les contrôleurs.

En résumé, voici les caractéristiques essentielles des modèles :

— Ils peuvent contenir des attributs pour représenter les données de l'application.

— Ils peuvent contenir des règles de validation pour garantir la validité et l'intégrité des données.
— Ils peuvent contenir des méthodes assurant le traitement logique des données de l'application.
— Ils ne devraient PAS accéder directement à la requête, à la session ou à n'importe quelle autre donnée environnementale. Ces données doivent être injectées dans les modèles par les contrôleurs.
— Ils devraient éviter d'inclure du code HTML ou tout autre code relatif à la présentation — cela est fait de manière plus avantageuse dans les vues.
— Il faut éviter d'avoir trop de scénarios dans un même modèle.

Vous pouvez ordinairement considérer cette dernière recommandation lorsque vous développez des systèmes importants et complexes. Dans ces systèmes, les modèles pourraient être très volumineux parce que, étant utilisés dans de nombreux endroits, ils doivent contenir de nombreux jeux de règles et de traitement logiques. Cela se termine le plus souvent en cauchemar pour la maintenance du code du modèle parce que le moindre changement de code est susceptible d'avoir de l'effet en de nombreux endroits. Pour rendre le modèle plus maintenable, vous pouvez adopter la stratégie suivante :

— Définir un jeu de classes de base du modèle qui sont partagées par différentes applications ou modules. Ces classes de modèles devraient contenir un jeu minimal de règles et de logique qui sont communes à tous les usages.
— Dans chacune des applications ou modules qui utilise un modèle, définir une classe de modèle concrète en étendant la classe de base de modèle correspondante. Les classes de modèles concrètes devraient contenir certaines règles et logiques spécifiques à cette application ou à ce module.

Par exemple, dans le Modèle avancé de projet [12], vous pouvez définir une classe de modèle de base `common\models\Post`. Puis, pour l'application « interface utilisateur » (*frontend*) vous pouvez définir une classe de base concrète `frontend\models\Post` qui étend la classe `common\models\Post`. De manière similaire, pour l'application « interface d'administration » (*backend*) vous pouvez définir une classe `backend\models\Post`. Avec cette stratégie, vous êtes sûr que le code de `frontend\models\Post` est seulement spécifique à l'application « interface utilisateur », et si vous y faite un changement, vous n'avez à vous soucier de savoir si cela à une influence sur l'application « interface d'administration ».

12. https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md

3.7 Vues

Les vues font partie du modèle d'architecture MVC [13] (Modèle Vue Contrôleur). Elles sont chargées de présenter les données à l'utilisateur final. Dans une application Web, les vues sont ordinairement créées en termes de *modèles de vue* qui sont des script PHP contenant principalement du code HTML et du code PHP relatif à la présentation.

Elles sont gérées par le `view` composant application qui fournit des méthodes d'usage courant pour faciliter la composition des vues et leur rendu. Par souci de simplicité, nous appellerons *vues* les modèles de vue et les fichiers de modèle de vue.

3.7.1 Création des vues

Comme nous l'avons dit ci-dessus, une vue n'est rien d'autre qu'un script PHP incluant du code HTML et du code PHP. Le script ci-dessous correspond à la vue d'un formulaire de connexion. Comme vous pouvez le voir le code PHP est utilisé pour générer le contenu dynamique, dont par exemple le titre de la page et le formulaire, tandis que le code HTML les organise en une page présentable.

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $form yii\widgets\ActiveForm */
/* @var $model app\models\LoginForm */

$this->title = 'Login';
?>
<h1><?= Html::encode($this->title) ?></h1>

<p>Veuillez remplir les champs suivants pour vous connecter:</p>

<?php $form = ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('Login') ?>
<?php ActiveForm::end(); ?>
```

À l'intérieur d'une vue, vous avez accès à `$this` qui fait référence au `composant view (vue)` responsable de le gestion et du rendu de ce modèle de vue.

En plus de `$this`, il peut aussi y avoir d'autres variables prédéfinies dans une vue, telles que `$model` dans l'exemple précédent. Ces variables représentent les données qui sont *poussées* dans la vue par les contrôleurs ou par d'autres objets qui déclenche le rendu d'une vue.

13. https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur

> **Conseil :** les variables prédéfinies sont listées dans un bloc de commentaires au début d'une vue de manière à être reconnues par les EDI. C'est également une bonne manière de documenter vos vues.

Sécurité

Lors de la création de vues qui génèrent des pages HTML, il est important que vous encodiez et/ou filtriez les données en provenance de l'utilisateur final avant des les présenter. Autrement, votre application serait sujette aux attaques par injection de scripts (*cross-site scripting*) [14].

Pour afficher du texte simple, commencez par l'encoder en appelant la méthode `yii\helpers\Html::encode()`. Par exemple, le code suivant encode le nom d'utilisateur (*username*) avant de l'afficher :

```
<?php
use yii\helpers\Html;
?>

<div class="username">
    <?= Html::encode($user->name) ?>
</div>
```

Pour afficher un contenu HTML, utilisez l'objet `yii\helpers\HtmlPurifier` pour d'abord en filtrer le contenu. Par exemple, le code suivant filtre le contenu de la variable *post* avant de l'afficher :

```
<?php
use yii\helpers\HtmlPurifier;
?>

<div class="post">
    <?= HtmlPurifier::process($post->text) ?>
</div>
```

> **Conseil :** bien que l'objet HTMLPurifier effectue un excellent travail en rendant vos sorties sûres, il n'est pas rapide. Vous devriez envisager de mettre le résultat en cache lorsque votre application requiert une performance élevée.

Organisation des vues

Comme les contrôleurs et les modèles, il existe certaines conventions pour organiser les vues.

— Pour les vues rendues par un contrôleur, elles devraient être placées par défaut dans le dossier `@app/views/ControllerID` où `ControllerID` doit

14. https://fr.wikipedia.org/wiki/Cross-site_scripting

être remplacé par l'identifiant du contrôleur. Par exemple, si la classe du contrôleur est `PostController`, le dossier est `@app/views/post` ; si c'est `PostCommentController` le dossier est `@app/views/post-comment`. Dans le cas où le contrôleur appartient à un module, le dossier s'appelle `views/ControllerID` et se trouve dans le `dossier de base du module`.
- Pour les vues rendues dans un objet graphique, elles devraient être placées par défaut dans le dossier `WidgetPath/views` où `WidgetPath` est le dossier contenant le fichier de la classe de l'objet graphique.
- Pour les vues rendues par d'autres objets, il est recommandé d'adopter une convention similaire à celle utilisée pour les objets graphiques.

Vous pouvez personnaliser ces dossiers par défaut en redéfinissant la méthode `yii\base\ViewContextInterface::getViewPath()` des contrôleurs ou des objets graphiques.

3.7.2 Rendu des vues

Vous pouvez rendre les vues dans des contrôleurs, des objets graphiques, ou dans d'autres endroits en appelant les méthodes de rendu des vues. Ces méthodes partagent un signature similaire comme montré ci-dessous : ` /**
- @param string $view nom de la vue ou chemin du fichier, selon la méthode réelle de rendu
- @param array $params les données injectées dans la vue
- @return string le résultat du rendu */ methodName($view, $params = [])

Rendu des vues dans des contrôleurs

Dans les contrôleurs, vous pouvez appeler la méthode de contrôleur suivante pour rendre une vue :
- `render()` : rend une vue nommée et applique une disposition au résultat du rendu.
- `renderPartial()` : rend une vue nommée sans disposition.
- `renderAjax()` : rend une vue nommée sans disposition et injecte tous les scripts et fichiers JS/CSS enregistrés. Cette méthode est ordinairement utilisée en réponse à une requête Web AJAX.
- `renderFile()` : rend une vue spécifiée en terme de chemin ou d'alias de fichier de vue.
- `renderContent()` : rend un chaîne statique en l'injectant dans la disposition courante. Cette méthode est disponible depuis la version 2.0.1.

Par exemple :

```
namespace app\controllers;

use Yii;
```

```php
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);
        if ($model === null) {
            throw new NotFoundHttpException;
        }

        // rend une vue nommée  "view" et lui applique une disposition de
        page
        return $this->render('view', [
            'model' => $model,
        ]);
    }
}
```

Rendu des vues dans les objets graphiques

Dans les objets graphiques, vous pouvez appeler les méthodes suivantes de la classe *widget* pour rendre une vue :

- `render()` : rend une vue nommée.
- `renderFile()` : rend une vue spécifiée en terme de chemin ou d'alias de fichier de vue.

Par exemple :

```php
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class ListWidget extends Widget
{
    public $items = [];

    public function run()
    {
        // rend  une vue nommée "list"
        return $this->render('list', [
            'items' => $this->items,
        ]);
    }
}
```

Rendu des vues dans des vues

Vous pouvez rendre une vue dans une autre vue en appelant les méthodes suivantes du `composant view` :

— `render()` : rend une vue nommée.
— `renderAjax()` : rend une vue nommée et injecte tous les fichiers et scripts JS/CSS enregistrés. On l'utilise ordinairement en réponse à une requête Web AJAX.
— `renderFile()` : rend une vue spécifiée en terme de chemin ou d'alias de fichier de vue.

Par exemple, le code suivant dans une vue, rend le fichier de vue `_overview.php` qui se trouve dans le même dossier que la vue courante. Rappelez-vous que `$this` dans une vue fait référence au composant `view` :

```
<?= $this->render('_overview') ?>
```

Rendu de vues en d'autres endroits

Dans n'importe quel endroit, vous pouvez accéder au composant d'application `view` à l'aide de l'expression `Yii::$app->view` et ensuite appeler une de ses méthodes mentionnées plus haut pour rendre une vue. Par exemple :

```
// displays the view file "@app/views/site/license.php"
echo \Yii::$app->view->renderFile('@app/views/site/license.php');
```

Vues nommées

Lorsque vous rendez une vue, vous pouvez spécifier la vue en utilisant soit un nom de vue, soit un chemin/alias de fichier de vue. Dans la plupart des cas, vous utilisez le nom car il est plus concis et plus souple. Nous appelons les vues spécifiées par leur nom, des *vues nommées*.

Un nom de vue est résolu en le chemin de fichier correspondant en appliquant les règles suivantes :

— Un nom de vue peut omettre l'extension du nom de fichier. Dans ce cas, `.php` est utilisé par défaut en tant qu'extension. Par exemple, le nom de vue `about` correspond au nom de fichier `about.php`.
— Si le nom de vue commence par une double barre de division `//`, le chemin de fichier correspondant est `@app/views/ViewName` où `ViewName` est le nom de la vue. Dans ce cas la vue est recherchée dans le dossier `chemin des vues de l'application`. Par exemple, `//site/about` est résolu en `@app/views/site/about.php`.
— Si le nom de la vue commence par une unique barre de division `/`, le chemin de fichier de la vue est formé en préfixant le nom de vue avec `chemin des vues` du module actif courant . Si aucun module n'est actif, `@app/views/ViewName` — où `ViewName` est le nom de la vue — est utilisé. Par exemple, `/user/create` est résolu en `@app/modules/user/views/user/create.php`, si le module actif courant est `user` et en `@app/views/user/create.php`si aucun module n'est actif.

— Si la vue est rendue avec un `contexte` et que le contexte implémente `yii\base\ViewContextInterface`,le chemin de fichier de vue est formé en préfixant le nom de vue avec le `chemin des vues` du contexte. Cela s'applique principalement aux vues rendues dans des contrôleurs et dans des objets graphiques. Par exemple, `about` est résolu en `@app/views/site/about.php` si le contexte est le contrôleur `SiteController`.
— Si une vue est rendue dans une autre vue, le dossier contenant le nom de la nouvelle vue est préfixé avec le chemin du dossier contenant l'autre vue. Par exemple, la vue `item` est résolue en `@app/views/post/item.php` lorsqu'elle est rendue dans `@app/views/post/index.php`.

Selon les règles précédentes, l'appel de `$this->render('view')` dans le contrôleur `app\controllers\PostController` rend réellement le fichier de vue `@app/views/post/view.php`, tandis que l'appel de `$this->render('_overview')` dans cette vue rend le fichier de vue `@app/views/post/_overview.php`.

Accès aux données dans les vues

Il existe deux approches pour accéder aux données à l'intérieur d'une vue : *pousser* et *tirer*.

En passant les données en tant que second paramètre des méthodes de rendu de vues, vous utilisez la méthode *pousser*. Les données doivent être présentées sous forme de tableau clé-valeur. Lorsque la vue est rendue, la fonction PHP `extract()` est appelée sur ce tableau pour que le tableau soit restitué sous forme de variables dans la vue. Par exemple, le code suivant de rendu d'une vue dans un contrôleur *pousse* deux variables dans la vue `report` :

`$foo = 1` et `$bar = 2`.

```
echo $this->render('report', [
    'foo' => 1,
    'bar' => 2,
]);
```

L'approche *tirer* retrouve les données de manière plus active à partir du `composant view` ou à partir d'autres objets accessibles dans les vues (p. ex. `Yii::$app`). En utilisant le code exemple qui suit, vous pouvez, dans une vue, obtenir l'objet contrôleur `$this->context`. Et, en conséquence, il vous est possible d'accéder à n'importe quelle propriété ou méthode du contrôleur, comme la propriété `id` du contrôleur :

```
The controller ID is: <?= $this->context->id ?>
```

L'approche *pousser* est en général le moyen préféré d'accéder aux données dans les vues, parce qu'elle rend les vues moins dépendantes des objets de contexte. Son revers, et que vous devez construire le tableau de données à chaque fois, ce qui peut devenir ennuyeux et sujet aux erreurs si la vue est rendue en divers endroits.

Partage de données entre vues

Le `composant view` dispose de la propriété `params` que vous pouvez utiliser pour partager des données entre vues.

Par exempe, dans une vue `about` (à propos), vous pouvez avoir le code suivant qui spécifie le segment courant du *fil d'Ariane*.

```
$this->params['breadcrumbs'][] = 'About Us';
```

Ainsi, dans le fichier de la disposition, qui est également une vue, vous pouvez afficher le *fil d'Ariane* en utilisant les données passées par `params` :

```
<?= yii\widgets\Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ?
    $this->params['breadcrumbs'] : [],
]) ?>
```

3.7.3 Dispositions

Les dispositions (*layouts*) sont des types spéciaux de vues qui représentent les parties communes de multiples vues. Par exemple, les pages de la plupart des applications Web partagent le même entête et le même pied de page. Bien que vous puissiez répéter le même entête et le même pied de page dans chacune des vues, il est préférable de le faire une fois dans une disposition et d'inclure le résultat du rendu d'une vue de contenu à l'endroit approprié de la disposition.

Création de dispositions

Parce que les dispositions sont aussi des vues, elles peuvent être créées de manière similaire aux vues ordinaires. Par défaut, les dispositions sont stockées dans le dossier `@app/views/layouts`. Les dispositions utilisées dans un module doivent être stockées dans le dossier `views/layouts` du `dossier de base du module`. Vous pouvez personnaliser le dossier par défaut des dispositions en configurant la propriété `yii\base\Module::$layoutPath` de l'application ou du module.

L'exemple qui suit montre à quoi ressemble une disposition. Notez que dans un but illustratif, nous avons grandement simplifié le code à l'intérieur de cette disposition. En pratique, vous désirerez ajouter à ce code plus de contenu, comme des balises head, un menu principal, etc.

```
<?php
use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $content string */
?>
<?php $this->beginPage() ?>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <?= Html::csrfMetaTags() ?>
    <title><?= Html::encode($this->title) ?></title>
    <?php $this->head() ?>
</head>
<body>
<?php $this->beginBody() ?>
    <header>My Company</header>
    <?= $content ?>
    <footer>&copy; 2014 by My Company</footer>
<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

Comme vous pouvez le voir, la disposition génère les balises HTML qui sont communes à toutes les pages. Dans la section `<body>` la disposition rend la variable `$content` qui représente le résultat de rendu d'une vue de contenu qui est poussée dans la disposition par l'appel à la fonction `yii\base\Controller::render()`.

La plupart des dispositions devraient appeler les méthodes suivantes, comme illustré dans l'exemple précédent. Ces méthodes déclenchent essentiellement des événements concernant le processus de rendu de manière à ce que des balises et des scripts enregistrés dans d'autres endroits puissent être injectés à l'endroit où ces méthodes sont appelées.

- `beginPage()` : cette méthode doit être appelée au tout début de la disposition. Elle déclenche l'événement `EVENT_BEGIN_PAGE` qui signale le début d'une page.
- `endPage()` : cette méthode doit être appelée à la fin de la disposition. Elle déclenche l'événement `EVENT_END_PAGE` qui signale la fin d'une page.
- `head()` : cette méthode doit être appelée dans la section `<head>` d'une page HTML. Elle génère une valeur à remplacer qui sera remplacée par le code d'entête HTML (p. ex. des balises liens, des balises meta, etc.) lorsqu'une page termine son processus de rendu.
- `beginBody()` : cette méthode doit être appelée au début de la section `<body>`. Elle déclenche l'événement `EVENT_BEGIN_BODY` et génère une valeur à remplacer qui sera remplacée par le code HTML enregistré (p. ex. Javascript) dont la cible est le début du corps de la page.
- `endBody()` : cette méthode doit être appelée à la fin de la section `<body>`. Elle déclenche l'événement `EVENT_END_BODY` et génère une valeur à remplacer qui sera remplacée par le code HTML enregistré (p. ex. Javascript) dont la cible est la fin du corps de la page.

Accès aux données dans les dispositions

Dans une disposition, vous avez accès à deux variables prédéfinies : `$this` et `$content`. La première fait référence au composant `view`, comme dans les vues ordinaires, tandis que la seconde contient le résultat de rendu d'une vue de contenu qui est rendue par l'appel de la méthode `render()` dans un contrôleur.

Si vous voulez accéder à d'autres données dans les dispositions, vous devez utiliser l'approche *tirer* comme c'est expliqué à la sous-section Accès aux données dans les vues. Si vous voulez passer des données d'une vue de contenu à une disposition, vous pouvez utiliser la méthode décrite à la sous-section Partage de données entre vues.

Utilisation des dispositions

Comme c'est décrit à la sous-section Rendu des vues dans les contrôleurs, lorsque vous rendez une vue en appelant la méthode `render()` dans un contrôleur, une disposition est appliquée au résultat du rendu. Par défaut, la disposition `@app/views/layouts/main.php` est utilisée.

Vous pouvez utiliser une disposition différente en configurant soit `yii\base\Application::$layout`, soit `yii\base\Controller::$layout`. La première gouverne la disposition utilisée par tous les contrôleurs, tandis que la deuxième redéfinit la première pour les contrôleurs individuels. Par exemple, le code suivant fait que le contrôleur `post` utilise `@app/views/layouts/post.php` en tant qu disposition lorsqu'il rend ses vues. Les autres contrôleurs, en supposant que leur propriété `layout` n'est pas modifiée, continuent d'utiliser la disposition par défaut `@app/views/layouts/main.php`.

```
namespace app\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    public $layout = 'post';

    // ...
}
```

Pour les contrôleurs appartenant à un module ,vous pouvez également configurer la propriété `layout` pour utiliser une disposition particulière pour ces contrôleurs.

Comme la propriété `layout` peut être configurée à différents niveaux (contrôleurs, modules, application), en arrière plan, Yii opère en deux étapes pour déterminer quel est le fichier de disposition réel qui doit être utilisé pour un contrôleur donné.

Dans la première étape, il détermine la valeurs de la disposition et le module du contexte :

— Si la propriété `yii\base\Controller::$layout` du contrôleur n'est pas nulle, il l'utilise en tant que valeur de disposition et le `module` du contrôleur en tant que module du contexte.
— Si la propriété `layout` est nulle, il cherche, à travers tous les modules ancêtres (y compris l'application elle-même) du contrôleur, le premier module dont la propriété `layout` n'est pas nulle. Il utilise alors ce module et la valeur de sa `disposition` comme module du contexte et valeur de disposition, respectivement. Si un tel module n'est pas trouvé, cela signifie qu'aucune disposition n'est appliquée.

Dans la seconde étape, il détermine le fichier de disposition réel en fonction de la valeur de disposition et du module du contexte déterminés dans la première étape. La valeur de disposition peut être :

— Un alias de chemin (p. ex. `@app/views/layouts/main`).
— Un chemin absolu (p. ex. `/main`) : la valeur de disposition commence par une barre oblique de division. Le fichier réel de disposition est recherché dans le `chemin des disposition (` (par défaut `@app/views/layouts`).
— Un chemin relatif (p. ex. `main`) : le fichier réel de disposition est recherché dans le `chemin des dispositions (` du module du contexte (par défaut`views/layouts`) dans le `dossier de base du module`.
— La valeur booléenne `false` : aucune disposition n'est appliquée.

Si la valeur de disposition ne contient pas d'extension de fichier, l'extension par défaut `.php` est utilisée.

Dispositions imbriquées

Parfois, vous désirez imbriquer une disposition dans une autre. Par exemple, dans les différentes sections d'un site Web, vous voulez utiliser des dispositions différentes, bien que ces dispositions partagent la même disposition de base qui génère la structure d'ensemble des pages HTML5. Vous pouvez réaliser cela en appelant la méthode `beginContent()` et la méthode `endContent()` dans les dispositions filles comme illustré ci-après : `php <?php $this->beginContent('@app/views/layouts/base.php'); ?>

...contenu de la disposition fille ici...

<?php $this->endContent(); ?> `

Comme on le voit ci-dessus, le contenu de la disposition fille doit être situé entre les appels des méthodes `beginContent()` et `endContent()`. Le paramètre passé à la méthode `beginContent()` spécifie quelle est la disposition parente. Ce peut être un fichier de disposition ou un alias. En utilisant l'approche ci-dessus, vous pouvez imbriquer des dispositions sur plusieurs niveaux.

Utilisation des blocs

Les blocs vous permettent de spécifier le contenu de la vue à un endroit et l'afficher ailleurs. Ils sont souvent utilisés conjointement avec les dispositions. Par exemple, vous pouvez définir un bloc dans une vue de contenu et l'afficher dans la disposition.

Pour définir un bloc, il faut appeler les méthodes `beginBlock()` et `endBlock()`. Vous pouvez accéder au bloc via son identifiant avec `$view->blocks[$blockID]`, où `$blockID` représente l'identifiant unique que vous assignez au bloc lorsque vous le définissez.

L'exemple suivant montre comment utiliser les blocs pour personnaliser des parties spécifiques dans la disposition d'une vue de contenu.

Tout d'abord, dans une vue de contenu, définissez un ou de multiples blocs :

```
...

<?php $this->beginBlock('block1'); ?>

...contenu de block1...

<?php $this->endBlock(); ?>

...

<?php $this->beginBlock('block3'); ?>

...contenu de block3...

<?php $this->endBlock(); ?>
```

Ensuite, dans la vue de la disposition, rendez les blocs s'ils sont disponibles, ou affichez un contenu par défaut si le bloc n'est pas défini.

```
...
<?php if (isset($this->blocks['block1'])): ?>
    <?= $this->blocks['block1'] ?>
<?php else: ?>
    ... contenu par défaut de  block1 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block2'])): ?>
    <?= $this->blocks['block2'] ?>
<?php else: ?>
    ... contenu par défaut de block2 ...
<?php endif; ?>

...
```

```
<?php if (isset($this->blocks['block3'])): ?>
    <?= $this->blocks['block3'] ?>
<?php else: ?>
    ... contenu par défaut de  block3 ...
<?php endif; ?>
...
```

3.7.4 Utilisation des composants view

Les composants `view` fournissent de nombreuses fonctionnalités relatives aux vues. Bien que vous puissiez créer des composants *view* en créant des instances de la classe `yii\base\View` ou de ses classes filles, dans la plupart des cas, vous utilisez principalement le composant d'application `view` . Vous pouvez configurer ce composant dans les configuration d'application, comme l'illustre l'exemple qui suit :

```
[
    // ...
    'components' => [
        'view' => [
            'class' => 'app\components\View',
        ],
        // ...
    ],
]
```

Les composants View fournissent les fonctionnalités utiles suivantes relatives aux vues, chacune décrite en détails dans une section séparée :

- gestion des thèmes : vous permet des développer et de changer les thèmes pour votre site Web.
- mise en cache de fragments : vous permet de mettre en cache un fragment de votre page Web.
- gestion des scripts client : prend en charge l'enregistrement et le rendu de code CSS et JavaScript.
- gestion de paquets de ressources : prend en charge l'enregistrement et le rendu de paquets de ressources.
- moteurs de modèle alternatif : vous permet d'utiliser d'autres moteur de modèles tels que Twig [15] et Smarty [16].

Vous pouvez aussi utiliser les fonctionnalités suivantes qui, bien que mineures, sont néanmoins utiles, pour développer vos pages Web.

Définition du titre des pages

Chaque page Web doit avoir un titre. Normalement la balise titre est affichée dans une disposition. Cependant, en pratique, le titre est souvent

15. https://twig.symfony.com/
16. https://www.smarty.net/

déterminé dans les vues de contenu plutôt que dans les dispositions. Pour résoudre ce problème,`yii\web\View` met à votre disposition la propriété `title` qui vous permet de passer l'information de titre de la vue de contenu à la disposition.

Pour utiliser cette fonctionnalité, dans chacune des vues de contenu, vous pouvez définir le titre de la page de la manière suivante : `php < ?php $this->title = 'Le titre de ma page'; ?> `

Ensuite dans la disposition, assurez-vous qui vous avez placé le code suivant dans la section `<head>` :

```
<title><?= Html::encode($this->title) ?></title>
```

Enregistrement des balises "meta"

Généralement, les pages Web, ont besoin de générer des balises "meta" variées dont ont besoin diverses parties. Comme le titre des pages, les balises "meta" apparaissent dans la section `<head>` et sont généralement générées dans les dispositions.

Si vous désirez spécifier quelles balises "meta" générer dans les vues de contenu, vous pouvez appeler `yii\web\View::registerMetaTag()` dans une vue de contenu comme illustrer ci-après :

```
<?php
$this->registerMetaTag(['name' => 'keywords', 'content' => 'yii, framework,
php']);
?>
```

Le code ci-dessus enregistre une balise "meta" "mot clé" dans le composant view. La balise "meta" enregistrée est rendue après que le rendu de la disposition est terminé. Le code HTML suivant est généré et inséré à l'endroit où vous appelez `yii\web\View::head()` dans la disposition :

```
<meta name="keywords" content="yii, framework, php">
```

Notez que si vous appelez `yii\web\View::registerMetaTag()` à de multiples reprises, elle enregistrera de multiples balises meta, que les balises soient les mêmes ou pas.

Pour garantir qu'il n'y a qu'une instance d'un type de balise meta, vous pouvez spécifier une clé en tant que deuxième paramètre lors de l'appel de la méthode. Par exemple, le code suivant, enregistre deux balises "meta" « description ». Cependant, seule la seconde sera rendue. F

```
$this->registerMetaTag(['name' => 'description', 'content' => 'This is my
cool website made with Yii!'], 'description');
$this->registerMetaTag(['name' => 'description', 'content' => 'This website
is about funny raccoons.'], 'description');
```

Enregistrement de balises liens

Comme les balises meta, les balises liens sont utiles dans de nombreux cas, comme la personnalisation de favicon, le pointage sur les flux RSS ou la délégation d'OpenID à un autre serveur. Vous pouvez travailler avec les balises liens comme avec les balises "meta" en utilisant `yii\web\View::registerLinkTag()`. Par exemple, dans une vue de contenu, vous pouvez enregistrer une balise lien de la manière suivante :

```
$this->registerLinkTag([
    'title' => 'Live News for Yii',
    'rel' => 'alternate',
    'type' => 'application/rss+xml',
    'href' => 'https://www.yiiframework.com/rss.xml/',
]);
```

Le code suivant produit le résultat suivant :

```
<link title="Live News for Yii" rel="alternate" type="application/rss+xml"
href="https://www.yiiframework.com/rss.xml/">
```

Comme avec `registerMetaTag()`, vous pouvez spécifier un clé lors de l'appel de `registerLinkTag()` pour éviter de générer des liens identiques.

3.7.5 Événements de vues

Les `composants View` déclenchent plusieurs événements durant le processus de rendu des vues. Vous pouvez répondre à ces événements pour injecter du contenu dans des vues ou traiter les résultats du rendu avant leur transmission à l'utilisateur final.

- `EVENT_BEFORE_RENDER` : déclenché au début du rendu d'un fichier dans un contrôleur. Les gestionnaires de cet événement peuvent définir `yii\base\ViewEvent::$isValid` à `false` (faux) pour arrêter le processus de rendu.
- `EVENT_AFTER_RENDER` : déclenché après le rendu d'un fichier par appel de `yii\base\View::afterRender()`. Les gestionnaires de cet événement peuvent obtenir le résultat du rendu via `yii\base\ViewEvent::$output` et peuvent modifier cette propriété pour modifier le résultat du rendu.
- `EVENT_BEGIN_PAGE` : déclenché par l'appel de `yii\base\View::beginPage()` dans une disposition.
- `EVENT_END_PAGE` : déclenché par l'appel de `yii\base\View::endPage()` dans une disposition.
- `EVENT_BEGIN_BODY` : déclenché par l'appel de `yii\web\View::beginBody()` dans une disposition.
- `EVENT_END_BODY` : déclenché par l'appel de `yii\web\View::endBody()` dans une disposition.

Par exemple, le code suivant injecte la date courante à la fin du corps de la page.

```
\Yii::$app->view->on(View::EVENT_END_BODY, function () {
    echo date('Y-m-d');
});
```

3.7.6 Rendu des pages statiques

Les pages statiques font références aux pages dont le contenu principal est essentiellement statique sans recours à des données dynamiques poussées par les contrôleurs.

Vous pouvez renvoyer des pages statiques en plaçant leur code dans des vues, et en utilisant un code similaire à ce qui suit dans un contrôleur :

```
public function actionAbout()
{
    return $this->render('about');
}
```

Si un site Web contient beaucoup de pages statiques, ce serait très ennuyeux de répéter un code similaire de nombreuses fois. Pour résoudre ce problème, vous pouvez introduire une action autonome appelée `yii\web\ViewAction` dans un contrôleur. Par exemple :

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'page' => [
                'class' => 'yii\web\ViewAction',
            ],
        ];
    }
}
```

Maintenant, si vous créez une vue nommée `about` dans le dossier `@app/views/site/pages`, vous pourrez afficher cette vue via l'URL suivante : `http://localhost/index.php?r=site%2Fpage&view=about`

Le paramètre `view` de la méthode `GET` dit à `yii\web\ViewAction` quelle est la vue requise. L'action recherche alors cette vue dans le dossier `@app/views/site/pages`. Vous pouvez configurer la propriété `yii\web\ViewAction::$viewPrefix` pour changer le dossier dans lequel la vue est recherchée.

3.7.7 Meilleures pratiques

Les vues sont chargées de présenter les modèles dans le format désiré par l'utilisateur final. En général :

- Elles devraient essentiellement contenir du code relatif à la présentation, tel que le code HTML, du code PHP simple pour parcourir, formater et rendre les données.
- Elles ne devraient pas contenir de code qui effectue des requêtes de base de données. Un tel code devrait être placé dans les modèles.
- Elles devraient éviter d'accéder directement aux données de la requête, telles que `$_GET`, `$_POST`. C'est le rôle des contrôleurs. Si les données de la requête sont nécessaires, elles devraient être poussées dans les vues par les contrôleurs.
- Elles peuvent lire les propriétés des modèles, mais ne devraient pas les modifier.

Pour rendre les vues plus gérables, évitez de créer des vues qui sont trop complexes ou qui contiennent trop de code redondant. Vous pouvez utiliser les techniques suivantes pour atteindre cet but :

- Utiliser des dispositions pour représenter les sections communes de présentation (p. ex. l'entête de page, le pied de page).
- Diviser une vue complexe en plusieurs vues plus réduites. Les vues plus réduites peuvent être rendue et assemblées dans une plus grande en utilisant les méthodes de rendu que nous avons décrites.
- Créer et utiliser des objets graphiques en tant que blocs de construction des vues.
- Créer et utiliser des classes d'aide pour transformer et formater les données dans les vues.

3.8 Filtres

Les filtres sont des objets qui sont exécutés avant et/ou après les actions de contrôleurs. Par exemple, un filtre de contrôle d'accès peut être exécuté avant les actions pour garantir qu'un utilisateur final particulier est autorisé à y accéder. Un filtre de compression de contenu peut être exécuté après les actions pour compresser la réponse avant de l'envoyer à l'utilisateur final.

Un filtre peut être constitué d"un pré-filtre (logique de filtrage appliquée *avant* les actions) et/ou un post-filtre (logique appliquée *après* les actions).

3.8.1 Utilisation des filtres

Pour l'essentiel, les filtres sont des sortes de comportements. Par conséquent, leur utilisation est identique à l' utilisation des comportements. Vous pouvez déclarer des filtres dans une classe de contrôleur en redéfinissant sa méthode `behaviors()` de la manière suivante :

```php
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index', 'view'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

Par défaut, les filtres déclarés dans une classe de contrôleur sont appliqués à *toutes* les action de ce contrôleur. Vous pouvez cependant, spécifier explicitement à quelles actions ils s'appliquent en configurant la propriété `only`. Dans l'exemple précédent, le filtre `HttpCache` s'applique uniquement aux actions `index` et `view`. Vous pouvez également configurer la propriété `except` pour empêcher quelques actions d'être filtrées.

En plus des contrôleurs, vous pouvez également déclarer des filtres dans un module ou dans une application. Lorsque vous faites cela, les filtres s'appliquent à *toutes* les actions de contrôleur qui appartiennent à ce module ou à cette application, sauf si vous configurez les propriétés des filtres `only` et `except` comme expliqué précédemment.

> **Note :** lorsque vous déclarez des filtres dans des modules ou des applications, vous devriez utiliser des routes plutôt que des identifiants d'action dans les propriétés `only` et `except`. Cela tient au fait qu'un identifiant d'action seul ne peut pas pleinement spécifier une action dans le cadre d'un module ou d'une application.

Lorsque plusieurs filtres sont configurés pour une même action, ils sont appliqués en respectant les règles et l'ordre qui suivent :

- Pré-filtrage
  - Les filtres déclarés dans l'application sont appliqués dans l'ordre dans lequel ils sont listés dans la méthode `behaviors()`.
  - Les filtres déclarés dans le module sont appliqués dans l'ordre dans lequel ils sont listés dans la méthode `behaviors()`.
  - Les filtres déclarés dans le contrôleur sont appliqués dans l'ordre dans lequel ils sont listés dans la méthode `behaviors()`.
  - Si l'un quelconque des filtres annule l'exécution de l'action, les filtres subséquents (à la fois de pré-filtrage et de post-fitrage) ne sont pas appliqués.
- L'action est exécutée si les filtres de pré-filtrage réussissent.
- Post-filtrage

— Les filtres déclarés dans le contrôleur sont appliqués dans l'ordre dans lequel ils sont listés dans la méthode `behaviors()`.
— Les filtres déclarés dans le module sont appliqués dans l'ordre dans lequel ils sont listés dans la méthode `behaviors()`.
— Les filtres déclarés dans l'application sont appliqués dans l'ordre dans lequel ils sont listés dans la méthode `behaviors()`.

3.8.2 Création de filtres

Pour créer un filtre d'action, vous devez étendre la classe `yii\base\ActionFilter` et redéfinir la méthode `beforeAction()` et/ou la méthode `afterAction()`. La première est exécutée avant l'exécution de l'action, tandis que la seconde est exécutée après l'exécution de l'action. Le valeur de retour de la méthode `beforeAction()` détermine si une action doit être exécutée ou pas. Si c'est `false` (faux), les filtres qui suivent sont ignorés et l'action n'est pas exécutée.

L'exemple qui suit montre un filtre qui enregistre dans un journal le temps d'exécution de l'action :

```
namespace app\components;

use Yii;
use yii\base\ActionFilter;

class ActionTimeFilter extends ActionFilter
{
    private $_startTime;

    public function beforeAction($action)
    {
        $this->_startTime = microtime(true);
        return parent::beforeAction($action);
    }

    public function afterAction($action, $result)
    {
        $time = microtime(true) - $this->_startTime;
        Yii::debug("Action '{$action->uniqueId}' spent $time second.");
        return parent::afterAction($action, $result);
    }
}
```

3.8.3 Filtres du noyau

Yii fournit un jeu de filtres couramment utilisés, que l'on trouve en premier lieu dans l'espace de noms `yii\filters`. Dans ce qui suit, nous introduisons brièvement ces filtres.

`AccessControl`

AccessControl (contrôle d'accès) fournit un contrôle d'accès simple basé sur un jeu de `règles`. En particulier, avant qu'une action ne soit exécutée, *AccessControl* examine les règles listées et trouve la première qui correspond aux variables du contexte courant (comme l'adresse IP, l'état de connexion de l'utilisateur, etc.). La règle qui correspond détermine si l'exécution de l'action requise doit être autorisée ou refusée. Si aucune des règles ne correspond, l'accès est refusé.

L'exemple suivant montre comment autoriser les utilisateurs authentifiés à accéder aux actions `create` et `update` tout en refusant l'accès à ces actions aux autres utilisateurs.

```
use yii\filters\AccessControl;

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::class,
            'only' => ['create', 'update'],
            'rules' => [
                // autoriser les utilisateurs authentifiés
                [
                    'allow' => true,
                    'roles' => ['@'],
                ],
                // tout autre chose est interdite d'accès par défaut
            ],
        ],
    ];
}
```

Pour plus de détails sur le contrôle d'accès en général, reportez-vous à la section Authorization.

Filtres de méthodes d'authentification

Les filtres de méthodes d'authentification sont utilisés pour authentifier un utilisateur qui utilise des méthodes d'authentification variées comme HTTP Basic Auth [17] ou OAuth 2 [18]. Les classes de filtre sont dans l'espace de noms `yii\filters\auth`.

L'exemple qui suit montre comment vous pouvez utiliser `yii\filters\auth\HttpBasicAuth` pour authentifier un utilisateur qui utilise un jeton d'accès basé sur la méthode *HTTP Basic Auth*. Notez qu'afin que cela fonctionne, votre `classe` doit implémenter l'interface `findIdentityByAccessToken()`.

17. https://en.wikipedia.org/wiki/Basic_access_authentication
18. https://oauth.net/2/

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    return [
        'basicAuth' => [
            'class' => HttpBasicAuth::class,
        ],
    ];
}
```

Les filtres de méthode d'authentification sont communément utilisés dans la mise en œuvre des API pleinement REST. Pour plus de détails, reportez-vous à la section Authentification REST.

ContentNegotiator

ContentNegotiator (négociateur de contenu) prend en charge la négociation des formats de réponse et la négociation de langue d'application. Il essaye de déterminer le format de la réponse et/ou la langue en examinant les paramètres de la méthode `GET` et ceux de l'entête HTTP `Accept`.

Dans l'exemple qui suit, le filtre *ContentNegotiator* est configuré pour prendre en charge JSON et XML en tant que formats de réponse, et anglais (États-Unis) et allemand en tant que langues.

```
use yii\filters\ContentNegotiator;
use yii\web\Response;

public function behaviors()
{
    return [
        [
            'class' => ContentNegotiator::class,
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ];
}
```

Les formats de réponse et les langues nécessitent souvent d'être déterminés bien plus tôt durant le cycle de vie de l'application. Pour cette raison, *ContentNegotiator* est conçu de manière à être également utilisé en tant que composant du processus d'amorçage. Par exemple, vous pouvez le configurer dans la configuration de l'application de la manière suivante :

```
use yii\filters\ContentNegotiator;
use yii\web\Response;

[
    'bootstrap' => [
        [
            'class' => ContentNegotiator::class,
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ],
];
```

> **Info :** dans le cas où le type de contenu et la langue préférés ne peuvent être déterminés à partir de la requête, le premier format et la première langue listés dans `formats` et`languages`, respectivement, sont utilisés.

HttpCache

HttpCache met en œuvre la mise en cache côté client en utilisant les entêtes HTTP `Last-Modified` (dernier modifié) et `Etag`. Par exemple :

```
use yii\filters\HttpCache;

public function behaviors()
{
    return [
        [
            'class' => HttpCache::class,
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

Reportez-vous à la section Mise en cache HTTP pour plus de détails sur l'utilisation de *HttpCache*.

PageCache

PageCache met en œuvre la mise en cache de pages entières côté serveur. Dans l'exemple qui suit, *PageCache0 est appliqué à l'action `index` pour

mettre la page entière en cache pendant un maximum de 60 secondes ou jusqu'à un changement du nombre d'entrées dans la table `post`. Il stocke également différentes versions de la page en fonction de la langue choisie.

```
use yii\filters\PageCache;
use yii\caching\DbDependency;

public function behaviors()
{
    return [
        'pageCache' => [
            'class' => PageCache::class,
            'only' => ['index'],
            'duration' => 60,
            'dependency' => [
                'class' => DbDependency::class,
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
            'variations' => [
                \Yii::$app->language,
            ]
        ],
    ];
}
```

Reportez-vous à la section Page Caching pour plus de détails sur l'utilisation de *PageCache*.

`RateLimiter`

RateLimiter met en œuvre un algorithme de limitation de débit basé sur l'algorithme leaky bucket [19]. On l'utilise en premier lieu dans la mise en œuvre des API pleinement REST. Reportez-vous à la section limitation de débit pour plus de détails sur l'utilisation de ce filtre.

`VerbFilter`

VerbFilter vérifie si les méthodes de requête HTTP sont autorisées par l'action requise. Si ce n'est pas le cas, une exception HTTP 405 est levée. Dans l'exemple suivant, *VerbFilter* est déclaré pour spécifier un jeu typique de méthodes de requête pour des actions CRUD — Create (créer), Read (lire), Update (mettre à jour), DELETE (supprimer).

```
use yii\filters\VerbFilter;

public function behaviors()
{
    return [
```

19. https://en.wikipedia.org/wiki/Leaky_bucket

```
        'verbs' => [
            'class' => VerbFilter::class,
            'actions' => [
                'index'  => ['get'],
                'view'   => ['get'],
                'create' => ['get', 'post'],
                'update' => ['get', 'put', 'post'],
                'delete' => ['post', 'delete'],
            ],
        ],
    ];
}
```

Cors

Cross-origin resource sharing CORS [20] est un mécanisme qui permet à des ressource (e.g. fonts, JavaScript, etc.) d'être requises d'un autre domaine en dehors du domaine dont la ressource est originaire. En particulier, les appels AJAX de Javascript peuvent utiliser le mécanisme *XMLHttpRequest*. Autrement, de telles requêtes "cross-domain" (inter domaines) seraient interdites par les navigateurs, à cause de la politique de sécurité dite d'origine identique (*same origin*). *CORS* définit une manière par laquelle le navigateur et le serveur interagissent pour déterminer si, oui ou non, la requête *cross-origin* (inter-site) est autorisée.

Le `filtre Cors` doit être défini avant les filtres d'authentification et/ou d'autorisation pour garantir que les entêtes CORS sont toujours envoyés.

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::class,
        ],
    ], parent::behaviors());
}
```

Consultez également la section sur les contrôleurs REST si vous voulez ajouter le filtre CORS à une classe `yii\rest\ActiveController` dans votre API.

Les filtrages Cors peuvent être peaufinés via la propriété `$cors`.

— `cors['Origin']` : un tableau utilisé pour définir les origines autorisées. Peut être `['*']` (tout le monde) ou `['https://www.myserver.net', 'https://www.myotherserver.com']`. Valeur par défaut `['*']`.

— `cors['Access-Control-Request-Method']` : un tableau des verbes autorisés tel que `['GET', 'OPTIONS', 'HEAD']`. Valeur par défaut `['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS']`.

20. https://developer.mozilla.org/fr/docs/Web/HTTP/CORS

— `cors['Access-Control-Request-Headers']` : un tableau des entêtes autorisés. Peut être`['*']` tous les entêtes ou certains spécifiquement `['X-Request-With']`. Valeur par défaut `['*']`.
— `cors['Access-Control-Allow-Credentials']` : définit si la requête courante peut être faite en utilisant des identifiants de connexion. Peut être `true` (vrai), `false` (faux) ou `null` (non défini). Valeur par défaut `null`.
— `cors['Access-Control-Max-Age']` : définit la durée de vie des requêtes de pré-vérification (*preflight requests*). Valeur par défaut `86400`.

Par exemple, autoriser CORS pour l'origine `https://www.myserver.net` avec les méthodes `GET`, `HEAD` et `OPTIONS` :

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::class,
            'cors' => [
                'Origin' => ['https://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD',
                'OPTIONS'],
            ],
        ],
    ], parent::behaviors());
}
```

Vous pouvez peaufiner les entêtes CORS en redéfinissant les paramètres par défaut action par action. Par exemple, ajouter les `Access-Control-Allow-Credentials` (autoriser les identifiants de contrôle d'accès) pour l'action`login` pourrait être réalisé comme ceci :

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::class,
            'cors' => [
                'Origin' => ['https://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD',
                'OPTIONS'],
            ],
            'actions' => [
                'login' => [
                    'Access-Control-Allow-Credentials' => true,
                ]
            ]
```

```
        ],
    ], parent::behaviors());
}
```

3.9 Objets graphiques

Les objets graphiques (*widgets*) sont des blocs de construction réutilisables dans des vues pour créer des éléments d'interface utilisateur complexes et configurables d'une manière orientée objet. Par exemple, un composant d'interface graphique de sélection de date peut générer un sélecteur de date original qui permet aux utilisateurs de sélectionner une date en tant qu'entrée. Tout ce que vous avez besoin de faire, c'est d'insérer le code dans une vue comme indiqué ci-dessous :

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget(['name' => 'date']) ?>
```

Il existe un grand nombre d'objets graphiques fournis avec Yii, tels que les`active form`, `menu`, jQuery UI widgets [21], Twitter Bootstrap widgets [22]. Dans ce qui suit, nous introduisons les connaissances de base sur les objets graphiques. Reportez-vous à la documentation de la classe dans l'API si vous désirez en apprendre davantage sur un objet graphique particulier.

3.9.1 Utilisation des objets graphiques

Les objets graphiques sont utilisés en premier lieu dans des vues. Vous pouvez appeler la méthode `yii\base\Widget::widget()` pour utiliser un objet graphique dans une vue. Cette méthode accepte un tableau de configuration pour initialiser l'objet graphique d'interface et retourne le résultat du rendu de cet objet. Par exemple, le code suivant insère un objet graphique de sélection de date qui est configuré dans la langue *russe* et conserve l'entrée dans l'attribut `from_date` du `$model`.

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget([
    'model' => $model,
    'attribute' => 'from_date',
    'language' => 'ru',
    'dateFormat' => 'php:Y-m-d',
]) ?>
```

21. https://www.yiiframework.com/extension/yiisoft/yii2-jui
22. https://www.yiiframework.com/extension/yiisoft/yii2-bootstrap

Quelques objets graphiques peuvent accepter un bloc de contenu qui doit être compris entre l'appel des méthodes `yii\base\Widget::begin()` et `yii\base\Widget::end()`. Par exemple, le code suivant utilise l'objet graphique `yii\widgets\ActiveForm` pour générer une ouverture de balise `<form>` à l'endroit de l'appel de `begin()` et une fermeture de la même balise à l'endroit de l'appel de `end()`. Tout ce qui se trouve entre les deux est rendu tel quel.

```
<?php
use yii\widgets\ActiveForm;
use yii\helpers\Html;
?>

<?php $form = ActiveForm::begin(['id' => 'login-form']); ?>

    <?= $form->field($model, 'username') ?>

    <?= $form->field($model, 'password')->passwordInput() ?>

    <div class="form-group">
        <?= Html::submitButton('Login') ?>
    </div>

<?php ActiveForm::end(); ?>
```

Notez que contrairement à la méthode `yii\base\Widget::widget()` qui retourne le résultat du rendu d'un objet graphique, la méthode `yii\base\Widget::begin()` retourne une instance de l'objet graphique que vous pouvez utiliser pour construire le contenu de l'objet d'interface.

> **Note :** quelques objets graphiques utilisent la mise en tampon de sortie[23] pour ajuster le contenu inclus quand la méthode `yii\base\Widget::end()` est appelée. Pour cette raison, l'appel des méthodes `yii\base\Widget::begin()` et `yii\base\Widget::end()` est attendu dans le même fichier de vue. Ne pas respecter cette règle peut conduire à des résultats inattendus.

Configuration des variables globales par défaut

Les variables globales par défaut pour un objet graphique peuvent être configurées via le conteneur d'injection de dépendances (*DI container*) :

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

Voir la section "Utilisation pratique " dans le Guide du conteneur d'injection de dépendances pour les détails.

23. https://www.php.net/manual/fr/book.outcontrol.php

3.9.2 Création d'objets graphiques

Pour créer un objet graphique, étendez la classe `yii\base\Widget` et redéfinissez sa méthode `yii\base\Widget::init()` et/ou sa méthode `yii\base\Widget::run()`. Ordinairement, la méthode `init()` devrait contenir le code qui initialise les propriétés de l'objet graphique, tandis que la méthode `run()` devrait contenir le code qui génère le résultat du rendu de cet objet graphique. Le résultat du rendu peut être "renvoyé en écho" directement ou retourné comme une chaîne de caractères par la méthode `run()`.

Dans l'exemple suivant, `HelloWidget` encode en HTML et affiche le contenu assigné à sa propriété `message`. Si la propriété n'est pas définie, il affiche "Hello World" par defaut.

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
{
    public $message;

    public function init()
    {
        parent::init();
        if ($this->message === null) {
            $this->message = 'Hello World';
        }
    }

    public function run()
    {
        return Html::encode($this->message);
    }
}
```

Pour utiliser cet objet graphique, contentez-vous d'insérer le code suivant dans une vue :

```
<?php
use app\components\HelloWidget;
?>
<?= HelloWidget::widget(['message' => 'Good morning']) ?>
```

Ce-dessous, nous présentons une variante de `HelloWidget` qui prend le contenu inséré entre les appels des méthodes `begin()` et `end()`, l'encode en HTML et l'affiche.

```
namespace app\components;

use yii\base\Widget;
```

```php
use yii\helpers\Html;

class HelloWidget extends Widget
{
    public function init()
    {
        parent::init();
        ob_start();
    }

    public function run()
    {
        $content = ob_get_clean();
        return Html::encode($content);
    }
}
```

Comme vous pouvez le voir, le tampon de sortie de PHP est démarré dans `init()` de manière à ce que toute sortie entre les appels de `init()` et de `run()` puisse être capturée, traitée et retournée dans `run()`.

> **Info :** lorsque vous appelez `yii\base\Widget::begin()`, une nouvelle instance de l'objet graphique est créé et sa méthode `init()` est appelée à la fin de la construction de l'objet. Lorsque vous appelez `yii\base\Widget::end()`, la méthode `run()` est appelée et sa valeur de retour est renvoyée en écho par `end()`.

Le code suivant montre comment utiliser cette nouvelle variante de `HelloWidget` :

```php
<?php
use app\components\HelloWidget;
?>
<?php HelloWidget::begin(); ?>

    content that may contain <tag>'s

<?php HelloWidget::end(); ?>
```

Parfois, un objet graphique peut avoir à rendre un gros bloc de contenu. Bien que vous puissiez incorporer le contenu dans la méthode `run()`, une meilleure approche consiste à le mettre dans une vue et à appeler la méthode `yii\base\Widget::render()` pour obtenir son rendu. Par exemple :

```php
public function run()
{
    return $this->render('hello');
}
```

Par défaut, les vues pour un objet graphique doivent être stockées dans le dossier `WidgetPath/views`, où `WidgetPath` représente le dossier contenant la classe de l'objet graphique. Par conséquent, l'exemple ci-dessus rend le fichier

de vue `@app/components/views/hello.php`, en supposant que la classe de l'objet graphique est située dans le dossier `@app/components`. Vous pouvez redéfinir la méthode `yii\base\Widget::getViewPath()` pour personnaliser le dossier qui contient les fichiers de vue des objets graphiques.

3.9.3 Meilleures pratiques

Les objets graphiques sont une manière orientée objets de réutiliser du code de vues.

Lors de la création d'objets graphiques, vous devriez continuer de suivre le modèle d'architecture MVC. En général, vous devriez conserver la logique dans les classes d'objets graphiques et la présentation dans les vues.

Les objets graphiques devraient également être conçus pour être auto-suffisants. Cela veut dire que, lors de l'utilisation d'un tel objet, vous devriez être en mesure de vous contenter de le placer dans une vue sans rien faire d'autre. Cela peut s'avérer délicat si un objet graphique requiert des ressources externes, comme du CSS, du JavaScript, des images, etc. Heureusement, Yii fournit une prise en charge des paquets de ressources que vous pouvez utiliser pour résoudre le problème.

Quand un objet graphique contient du code de vue seulement, il est très similaire à une vue. En fait, dans ce cas, la seule différence est que l'objet graphique est une classe distribuable, tandis qu'une vue est juste un simple script PHP que vous préférez conserver dans votre application.

3.10 Modules

Les modules sont des unités logicielles auto-suffisantes constituées de modèles, vues, contrôleurs et autres composants de prise en charge. L'utilisateur final peut accéder aux contrôleurs dans un module lorsqu'il est installé dans une application. Pour ces raisons, les modules sont souvent regardés comme de mini-applications. Les modules diffèrent des applications par le fait que les modules ne peuvent être déployés seuls et doivent résider dans une application.

3.10.1 Création de modules

Un module est organisé comme un dossier qui est appelé le `dossier de base (` du module. Dans ce dossier, se trouvent des sous-dossiers, tels que `controllers`, `models` et `views`, qui contiennent les contrôleurs, les modèles , les vues et d'autres parties de code, juste comme une application. L'exemple suivant présente le contenu d'un module :

```
forum/
    Module.php                  le fichier de classe du module
```

```
    controllers/                contient les fichiers de classe des
    contrôleurs
        DefaultController.php   le fichier de classe de contrôleur par
        défaut
    models/                     contient les fichiers de classe des modèles
    views/                      contient les fichiers de contrôleur, de vue
    et de disposition
        layouts/                contient les fichiers de diposition
        default/                contient les fichiers de vues pour le
        contrôleur par défaut
            index.php           le fichier de vue index
```

Classes de module

Chacun des modules doit avoir une classe unique de module qui étend `yii\base\Module`. La classe doit être située directement dans le `dossier de base` du module et doit être auto-chargeable. Quand un module est accédé, une instance unique de la classe de module correspondante est créée. Comme les instances d'application, les instances de module sont utilisées pour partager des données et des composants.

L'exemple suivant montre à quoi une classe de module peut ressembler :

```php
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->params['foo'] = 'bar';
        // ...  other initialization code ...
    }
}
```

La méthode `init()` contient un code volumineux pour initialiser les propriétés du module. Vous pouvez également les sauvegarder sous forme de configuration et charger cette configuration avec le code suivant dans la méthode `init()` :

```php
public function init()
{
    parent::init();
    // initialise le module à partir de la configuration chargée depuis
    config.php
    \Yii::configure($this, require __DIR__ . '/config.php');
}
```

où le fichier de configuration `config.php` peut avoir le contenu suivant, similaire à celui d'une configuration d'application.

```php
<?php
return [
    'components' => [
        // liste des configurations de composant
    ],
    'params' => [
        // liste des paramètres
    ],
];
```

Contrôleurs dans les modules

Lorsque vous créez des contrôleurs dans un module, une convention est de placer les classes de contrôleur dans le sous-espace de noms `controllers` dans l'espace de noms de la classe du module. Cela signifie également que les fichiers de classe des contrôleur doivent être placés dans le dossier `controllers` dans le `dossier de base` du module. Par exemple, pour créer un contrôleur `post` dans le module `forum` présenté dans la section précédente, vous devez déclarer la classe de contrôleur comme ceci :

```php
namespace app\modules\forum\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    // ...
}
```

Vous pouvez personnaliser l'espace de noms des classes de contrôleur en configurant la propriété `yii\base\Module::$controllerNamespace`. Dans le cas où certains contrôleurs sont en dehors de cet espace de noms, vous pouvez les rendre accessibles en configurant la propriété `yii\base\Module::$controllerMap` comme vous le feriez dans une application.

Vues dans les modules

Les vues dans les modules doivent être placées dans le dossier `views` du `dossier de base (` du module. Quant aux vues rendues par un contrôleur du module, elles doivent être placées dans le dossier `views/ControllerID`, où `ControllerID` fait référence à l'identifiant du contrôleur. Par exemple, si la classe du contrôleur est `PostController`, le dossier doit être `views/post` dans le `dossier de base` du module.

Un module peut spécifier une disposition qui s'applique aux vues rendues par les contrôleurs du module. La disposition doit être mise dans le dossier `views/layouts` par défaut, et vous devez configurer la propriété `yii\base\Module::$layout` pour qu'elle pointe sur le nom de la disposition. Si vous ne configurez pas la propriété `layout` c'est la disposition de l'application qui est utilisée à sa place.

Commande de console dans les modules

Votre module peut aussi déclarer des commandes, qui sont accessibles via le mode Console.

Afin que l'utilitaire de ligne de commande reconnaisse vos commandes, vous devez changer la propriété [[yii\base\Module : :controllerNamespace (espace de noms du contrôleur)]] lorsque Yii est exécuté en mode console, et le diriger sur votre espace de noms de commandes.

Une manière de réaliser cela est de tester le type d'instance de l'application Yii dans la méthode `init` du module :

```
public function init()
{
    parent::init();
    if (Yii::$app instanceof \yii\console\Application) {
        $this->controllerNamespace = 'app\modules\forum\commands';
    }
}
```

Vos commandes seront disponibles en ligne de commande en utilisant la route suivante :

```
yii <module_id>/<command>/<sub_command>
```

3.10.2 Utilisation des modules

Pour utiliser un module dans une application, il vous suffit de configurer l'application en listant le module dans la propriété `modules` de l'application. Le code qui suit dans la configuration de l'application permet l'utilisation du module `forum` :

```
[
    'modules' => [
        'forum' => [
            'class' => 'app\modules\forum\Module',
            // ... autres éléments de  configuration pour le module ...
        ],
    ],
]
```

La propriété `modules` accepte un tableau de configurations de module. Chaque clé du tableau représente un *identifiant de module* qui distingue ce module parmi les autres modules de l'application, et la valeur correspondante est une configuration pour la création du module.

Routes

Les routes sont utilisées pour accéder aux contrôleurs d'un module comme elles le sont pour accéder aux contrôleurs d'une application. Une route,

pour un contrôleur d'un module, doit commencer par l'identifiant du module, suivi de l'identifiant du contrôleur et de identifiant de l'action. Par exemple, si une application utilise un module nommé `forum`, alors la route `forum/post/index` représente l'action `index` du contrôleur `post` du module. Si la route ne contient que l'identifiant du module, alors la propriété `yii\base\Module::$defaultRoute`, dont la valeur par défaut est `default`, détermine quel contrôleur/action utiliser. Cela signifie que la route `forum` représente le contrôleur `default` dans le module `forum`.

Le gestionnaire d'URL du module doit être ajouté avant que la fonction `yii\web\UrlManager::parseRequest()` ne soit exécutée. Cela siginifie que le faire dans la fonction `init()` du module ne fonctionne pas parce que le module est initialisé après que les routes ont été résolues. Par conséquent, les règles doivent être ajoutées à l'étape d'amorçage. C'est également une bonne pratique d'empaqueter les règles d'URL du module dans `yii\web\GroupUrlRule`.

Dans le cas où un module est utilisé pour versionner une API, ses règles d'URL doivent être ajoutées directement dans la section `urlManager` de la configuration de l'application.

Accès aux modules

Dans un module, souvent, il arrive que vous ayez besoin d'une instance de la classe du module de façon à pouvoir accéder à l'identifiant du module, à ses paramètres, à ses composants, etc. Vous pouvez le faire en utilisant l'instruction suivante :

```
$module = MyModuleClass::getInstance();
```

dans laquelle `MyModuleClass` fait référence au nom de la classe du module qui vous intéresse. La méthode `getInstance()` retourne l'instance de la classe du module actuellement requis. Si le module n'est pas requis, la méthode retourne `null`. Notez que vous n'avez pas besoin de créer manuellement une nouvelle instance de la classe du module parce que celle-ci serait différente de celle créée par Yii en réponse à la requête.

> **Info :** lors du développement d'un module, vous ne devez pas supposer que le module va utiliser un identifiant fixe. Cela tient au fait qu'un module peut être associé à un identifiant arbitraire lorsqu'il est utilisé dans une application ou dans un autre module. Pour obtenir l'identifiant du module, vous devez utiliser l'approche ci-dessus pour obtenir d'abord une instance du module, puis obtenir l'identifiant via `$module->id`.

Vous pouvez aussi accéder à l'instance d'un module en utilisant les approches suivantes :

```
// obtenir le module fils dont l'identifiant est "forum"
$module = \Yii::$app->getModule('forum');

// obtenir le  module auquel le contrôleur actuellement requis appartient
$module = \Yii::$app->controller->module;
```

La première approche n'est utile que lorsque vous connaissez l'identifiant du module, tandis que la seconde est meilleure lorsque vous connaissez le contrôleur actuellement requis.

Une fois que vous disposez de l'instance du module, vous pouvez accéder aux paramètres et aux composants enregistrés avec le module. Par exemple :

```
$maxPostCount = $module->params['maxPostCount'];
```

Modules faisant partie du processus d'amorçage

Il se peut que certains modules doivent être exécutés pour chacune des requêtes. Le module `yii\debug\Module` en est un exemple. Pour que des modules soit exécutés pour chaque requête, vous devez les lister dans la propriété `bootstrap` de l'application.

Par exemple, la configuration d'application suivante garantit que le module `debug` est chargé à chaque requête :

```
[
    'bootstrap' => [
        'debug',
    ],

    'modules' => [
        'debug' => 'yii\debug\Module',
    ],
]
```

3.10.3 Modules imbriqués

Les modules peuvent être imbriqués sur un nombre illimité de niveaux. C'est à dire qu'un module pour contenir un autre module qui contient lui-même un autre module. Nous parlons alors de *module parent* pour le module englobant, et de *module enfant* pour le module contenu. Les modules enfants doivent être déclarés dans la propriété `modules` de leur module parent. Par exemple :

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();
```

```
        $this->modules = [
            'admin' => [
                // Vous devriez envisager l'utilisation d'un espace de noms
                plus court ici !
                'class' => 'app\modules\forum\modules\admin\Module',
            ],
        ];
    }
}
```

La route vers un contrôleur inclus dans un module doit inclure les identifiants de tous ses modules ancêtres. Par exemple, la route `forum/admin/dashboard/index` représente l'action `index` du contrôleur `dashboard` dans le module `admin` qui est un module enfant du module `forum`.

> **Info :** la méthode `getModule()` ne retourne que le module enfant appartenant directement à son parent. La propriété `yii\base\Application::$loadedModules` tient à jour une liste des modules chargés, y compris les enfant directs et les enfants des générations suivantes, indexée par le nom de classe.

3.10.4 Accès aux composants depuis l'intérieur des modules

Depuis la version 2.0.13, les modules prennent en charge la traversée des arbres. Cela permet aux développeurs de modules de faire référence à des composants (d'application) via le localisateur de services qui se trouve dans leur module. Cela signifie qu'il est préférable d'utiliser `$module->get('db')` plutôt que `Yii::$app->get('db')`. L'utilisateur d'un module est capable de spécifier un composant particulier pour une utilisation dans le module dans le cas où une configuration différente du composant est nécessaire.

Par exemple, considérons cette partie de la configuration d'une application :

```
'components' => [
    'db' => [
        'tablePrefix' => 'main_',
        'class' => Connection::class,
        'enableQueryCache' => false
    ],
],
'modules' => [
    'mymodule' => [
        'components' => [
            'db' => [
                'tablePrefix' => 'module_',
                'class' => Connection::class
            ],
        ],
    ],
],
```

Les tables de base de données de l'application seront préfixées par `main_`, tandis que les tables de tous les modules seront préfixées par `module_`. Notez cependant que la configuration ci-dessus n'est pas fusionnée ; le composant des modules par exemple aura le cache de requêtes activé puisque c'est la valeur par défaut.

3.10.5 Meilleures pratiques

L'utilisation des modules est préférable dans les grosses applications dont les fonctionnalités peuvent être réparties en plusieurs groupes, consistant chacun en un jeu de fonctionnalités liées d'assez près. Chacune de ces fonctionnalités peut être conçue comme un module développé et maintenu par un développeur ou une équipe spécifique.

Les modules sont aussi un bon moyen de réutiliser du code au niveau des groupes de fonctionnalités. Quelques fonctionnalité d'usage courant, telles que la gestion des utilisateurs, la gestion des commentaires, etc. peuvent être développées en tant que modules ce qui facilite leur réutilisation dans les projets suivants.

3.11 Ressources

Une ressource dans Yii est un fichier qui peut être référencé dans une page Web. Ça peut être un fichier CSS, un fichier JavaScript, une image, un fichier vidéo, etc. Les ressources sont situées dans un dossier accessible du Web et sont servies directement par les serveurs Web.

Il est souvent préférable de gérer les ressources par programmation. Par exemple, lorsque vous utilisez l'objet graphique `yii\jui\DatePicker` dans une page, il inclut automatiquement les fichiers CSS et JavaScript dont il a besoin, au lieu de vous demander de les inclure à la main.

De plus, lorsque vous mettez à jour l'objet graphique, il utilise une nouvelle version des fichiers de ressources. Dans ce tutoriel, nous décrivons les puissantes possibilités de la gestion des ressources de Yii.

3.11.1 Paquets de ressources

Yii gère les ressources sous forme de *paquets de ressources*. Un paquet de ressources est simplement une collection de ressources situées dans un dossier. Lorsque vous enregistrez un paquet de ressources dans une vue, cette vue inclut les fichiers CSS et JavaScript du paquet dans la page Web rendue.

3.11.2 Définition de paquets de ressources

Les paquets de ressources sont spécifiés comme des classes PHP qui étendent `yii\web\AssetBundle`. Le nom du paquet est simplement le nom

pleinement qualifié de la classe PHP correspondante (sans la barre oblique inversée de tête). Une classe de paquet de ressources doit être auto-chargeable. Généralement, elle spécifie où les ressources sont situées, quels fichiers CSS et JavaScript le paquet contient, et si le paquet dépend d'autres paquets de ressources.

Le code suivant définit le paquet de ressources principal utilisé par le modèle de projet *basic* :

```
<?php

namespace app\assets;

use yii\web\AssetBundle;

class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.css',
        ['css/print.css', 'media' => 'print'],
    ];
    public $js = [
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

La classe `AppAsset` ci-dessus spécifie que les fichiers de ressources sont situés dans le dossier `@webroot` qui correspond à l'URL `@web` ; le paquet contient un unique fichier CSS `css/site.css` et aucun fichier JavaScript ; le paquet dépend de deux autres paquets : `yii\web\YiiAsset` et `yii\bootstrap\BootstrapAsset`. Des explications plus détaillées sur les propriétés d'`yii\web\AssetBundle` sont disponibles dans les ressources suivantes :

— `sourcePath` (chemin des sources) : spécifie le dossier racine qui contient les fichiers de ressources dans ce paquet. Cette propriété doit être définie si le dossier racine n'est pas accessible du Web. Autrement, vous devez définir les propriétés `basePath` et `baseUrl`. Des alias de chemin sont utilisables ici.

— `basePath` (chemin de base) : spécifie un dossier accessible du Web qui contient les fichiers de ressources dans ce paquet. Lorsque vous spécifiez la propriété`sourcePath (chemin des sources)`, le gestionnaire de ressources publie les ressources de ce paquet dans un dossier accessible du Web et redéfinit cette propriété en conséquence. Vous devez définir cette propriété si vos fichiers de ressources sont déjà dans un dossier accessible du Web et n'ont pas besoin d'être publiés. Les alias de chemin sont utilisables ici.

- `baseUrl` (URL de base) : spécifie l'URL qui correspond au dossier `basePath`. Comme pour `basePath` (chemin de base), si vous spécifiez la propriété `sourcePath`, le gestionnaire de ressources publie les ressources et redéfinit cette propriété en conséquence. Les alias de chemin sont utilisables ici.
- `css` : un tableau listant les fichiers CSS contenu dans ce paquet de ressources. Notez que seul la barre oblique "/" doit être utilisée en tant que séparateur de dossier. Chaque fichier peut être spécifié en lui-même comme une chaîne de caractères ou dans un tableau avec les balises attributs et leur valeur.
- `js` : un tableau listant les fichiers JavaScript contenus dans ce paquet. Notez que seule la barre oblique de division "/" peut être utilisée en tant que séparateur de dossiers. Chaque fichier JavaScript peut être spécifié dans l'un des formats suivants :
  - Un chemin relatif représentant un fichier JavaScript local (p. ex. `js/main.js`). Le chemin réel du fichier peut être déterminé en préfixant le chemin relatif avec le `chemin de base`, et l'URL réelle du fichier peut être déterminée en préfixant le chemin relatif avec l'`URL de base`.
  - Une URL absolue représentant un fichier JavaScript externe. Par exemple , `https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js` ou `//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js`.
- `depends (dépendances)` : un tableau listant les paquets de ressources dont ce paquet dépend (brièvement expliqué).
- `jsOptions` : spécifie les options qui sont passées à la méthode `yii \web\View::registerJsFile()` lorsqu'elle est appelée pour enregistrer *chacun des* fichiers JavaScript de ce paquet.
- `cssOptions` : spécifie les options qui sont passées à la méthode `yii \web\View::registerCssFile()` lorsqu'elle est appelée pour enregistrer *chacun des* fichiers CSS de ce paquet.
- `publishOptions` : spécifie les options qui sont passées à la méthode `yii \web\AssetManager::publish()` lorsqu'elle est appelée pour publier les fichiers de ressources sources dans un dossier accessible du Web. Cela n'est utilisé que si vous spécifiez la propriété `sourcePath`.

Emplacement des ressources

En se basant sur leur emplacement, les ressources peuvent être classifiées comme suit :

- Les ressources sources : les fichiers de ressources qui sont situés avec du code source PHP et qui ne peuvent être accéder directement depuis le Web. Afin de pouvoir être utilisées dans une page, elles doivent être copiées dans un dossier accessible du Web et transformées en ressources publiées. Ce processus est appelé *publication des ressources* et il sera

décrit en détail bientôt.
— Les ressources publiées : les fichiers de ressources sont situés dans un dossier accessible du Web et peuvent par conséquent être accédés directement depuis le Web.
— Les ressources externes : les fichiers de ressources sont situés sur un serveur Web différent de celui qui héberge l'application Web.

Lors de la définition de classes de paquet de ressources, si vous spécifiez la propriété `sourcePath (chemin des sources)`, cela veut dire que les ressources listées en utilisant des chemins relatifs sont considérées comme des ressources sources. Si vous ne spécifiez pas cette propriété, cela signifie que ces ressources sont des ressources publiées (vous devez en conséquence spécifier [[yii\web\AssetBundle : :basePath (chemin de base)|basePath]] et `baseUrl (URL de base)` pour faire connaître à Yii l'emplacement où elles se trouvent).

Il est recommandé de placer les ressources appartenant à une application dans un dossier accessible du Web de manière à éviter une publication non nécessaire de ressources. C'est pourquoi `AppAsset` dans l'exemple précédent spécifie le `chemin de base` plutôt que le `chemin des sources`.

Quant aux extensions, comme leurs ressources sont situées avec le code source dans des dossiers non accessibles depuis le Web, vous devez spécifier la propriété `sourcePath` lorsque vous définissez des classes de paquet de ressources pour elles.

> **Note :** n'utilisez pas `@webroot/assets` en tant que `chemin des sources`. Ce dossier est utilisé par défaut par le `gestionnaire de ressources` pour sauvegarder les fichiers de ressources publiés depuis leur emplacement source. Tout contenu dans ce dossier est considéré temporaire et sujet à suppression.

Dépendances de ressources

Lorsque vous incluez plusieurs fichiers CSS ou JavaScript dans une page Web, ils doivent respecter un certain ordre pour éviter des problèmes de redéfinition. Par exemple, si vous utilisez l'objet graphique jQuery Ui dans une page Web, vous devez vous assurer que le fichier JavaScript jQuery est inclus avant le fichier JavaScript jQuery UI. Nous appelons un tel ordre : « dépendances entre ressources ».

Les dépendances entre ressources sont essentiellement spécifiées via la propriété `yii\web\AssetBundle::$depends`. Dans l'exemple `AppAsset`, le paquet de ressources dépend de deux autres paquets de ressources : `yii\web\YiiAsset` et `yii\bootstrap\BootstrapAsset`, ce qui veut dire que les fichiers CSS et JavaScript dans `AppAsset` sont inclus *après* les fichiers contenus dans ces deux paquets de ressources dont ils dépendent.

Les dépendances entre ressources sont transitives. Cela veut dire que si un paquet de ressources A dépend d'un paquet B qui lui-même dépend de C, A dépend de C également.

Options des ressources

Vous pouvez spécifier les propriétés `cssOptions` et `jsOptions` pour personnaliser la manière dont les fichiers CSS et JavaScript sont inclus dans une page. Les valeurs de ces propriétés sont passées aux méthodes `yii\web\View::registerCssFile()` et `yii\web\View::registerJsFile()`, respectivement, lorsqu'elles sont appelées par la vue pour inclure les fichiers CSS et JavaScript.

> **Note :** les options que vous définissez dans une classe de paquet de ressources s'appliquent à *chacun des* fichiers CSS/JavaScript du paquet. Si vous voulez utiliser des options différentes entre fichiers, vous devez utiliser le format indiqué `ci-dessus` ou créer des paquets de ressources séparés et utiliser un jeu d'options dans chacun des paquets.

Par exemple, pour inclure un fichier CSS sous condition que le navigateur soit IE9 ou inférieur, vous pouvez utiliser l'option suivante :

```
public $cssOptions = ['condition' => 'lte IE9'];
```

Avec cela, le fichier CSS du paquet pourra être inclus avec le code HTML suivant :

```
<!--[if lte IE9]>
<link rel="stylesheet" href="path/to/foo.css">
<![endif]-->
```

Pour envelopper le lien CSS généré dans une balise `<noscript>`, vous pouvez configurer `cssOptions` comme ceci :

```
public $cssOptions = ['noscript' => true];
```

Pour inclure un fichier JavaScript dans la section d'entête d'une page (par défaut les fichiers JavaScript sont inclus à la fin de la section body), utilisez l'option suivante :

```
public $jsOptions = ['position' => \yii\web\View::POS_HEAD];
```

Par défaut, lorsqu'un paquet de ressources est publié, tous les contenus dans le dossier spécifié par la propriété `yii\web\AssetBundle::$sourcePath` sont publiés. Vous pouvez personnaliser ce comportement en configurant la propriété `publishOptions`. Par exemple, pour publier seulement un ou quelques sous-dossiers du dossier spécifié par la propriété `yii\web\AssetBundle::$sourcePath`, vous pouvez procéder comme ceci dans la classe du paquet de ressources :

```php
<?php
namespace app\assets;

use yii\web\AssetBundle;

class FontAwesomeAsset extends AssetBundle
{
    public $sourcePath = '@bower/font-awesome';
    public $css = [
        'css/font-awesome.min.css',
    ];
    public $publishOptions = [
        'only' => [
            'fonts/',
            'css/',
        ]
    ];
}
```

L'exemple ci-dessus définit un paquet de ressources pour le paquet "fontawesome"[24]. En spécifiant l'option de publication `only`, seuls les sous-dossiers `fonts` et `css` sont publiés.

Installation des ressources Bower et NPM

La plupart des paquets JavaScript/CSS sont gérés par le gestionnaire de paquets Bower[25] et/ou le gestionnaire de paquets NPM[26]. Dans le monde PHP, nous disposons de Composer, qui gère les dépendances, mais il est possible de charger des paquets Bower et NPM comme des paquets PHP en utilisant `composer.json`.

Pour cela, nous devons configurer quelque peu notre composer. Il y a deux options possibles :

En utilisant le dépôt asset-packagist Cette façon de faire satisfera les exigences de la majorité des projets qui ont besoin de paquets Bower ou NPM.

> **Note :** depuis la version 2.0.13, les modèles de projet Basic et Advanced sont tous deux configuré pour utiliser asset-packagist par défaut, c'est pourquoi, vous pouvez sauter cette section.

Dans le fichier `composer.json` de votre projet, ajoutez les lignes suivantes :

```
"repositories": [
    {
```

24. https://fontawesome.com/
25. https://bower.io/
26. https://www.npmjs.com/

```
        "type": "composer",
        "url": "https://asset-packagist.org"
    }
]
```

Ajustez les aliases `@npm` et `@bower` dans la configuration de votre application :

```
$config = [
    ...
    'aliases' => [
        '@bower' => '@vendor/bower-asset',
        '@npm'   => '@vendor/npm-asset',
    ],
    ...
];
```

Visitez asset-packagist.org [27] pour savoir comment il fonctionne.

En utilisant le fxp/composer-asset-plugin Comparé à asset-packagist, composer-asset-plugin ne nécessite aucun changement dans la configuration de l'application. Au lieu de cela, il nécessite l'installation globale d'un greffon spécifique de Composer en exécutant la commande suivante :

```
composer global require "fxp/composer-asset-plugin:^1.4.1"
```

Cette commande installe composer asset plugin [28] globalement, ce qui permet de gérer les dépendances des paquets Bower et NPM via Composer. Après l'installation du greffon, tout projet de votre ordinateur prendra en charge les paquets Bower et NPM via `composer.json`.

Ajoutez les lignes suivantes au fichier `composer.json` de votre projet pour préciser les dossiers où seront installés les paquets, si vous voulez les publier en utilisant Yii :

```
"config": {
    "asset-installer-paths": {
        "npm-asset-library": "vendor/npm",
        "bower-asset-library": "vendor/bower"
    }
}
```

> **Note :** `fxp/composer-asset-plugin` ralentit significativement la commande `composer update` en comparaison avec asset-packagist.

Après avoir configuré Composer pour qu'il prenne en charge Bower et NPM :

27. https://asset-packagist.org
28. https://github.com/fxpio/composer-asset-plugin

1. Modifiez le fichier the `composer.json` de votre application ou extension et listez le paquet dans l'entrée `require`. Vous devez utiliser `bower-asset/PackageName` (pour les paquets Bower) ou `npm-asset/PackageName` (pour les paquets NPM) pour faire référence à la bibliothèque.
2. Exécutez `composer update`
3. Créez une classe de paquet de ressources et listez les fichiers JavaScript/CSS que vous envisagez d'utiliser dans votre application ou extension. Vous devez spécifier la propriété `sourcePath` comme `@bower/PackageName` ou `@npm/PackageName`. Cela parce que Composer installera le paquet Bower ou NPM dans le dossier correspondant à cet alias.

   Note : quelques paquets peuvent placer tous leurs fichiers distribués dans un sous-dossier. Si c'est le cas, vous devez spécifier le sous-dossier en tant que valeur de `sourcePath`. Par exemple, utilisez `yii\web\JqueryAsset` `@bower/jquery/dist` au lieu de `@bower/jquery`.

3.11.3 Utilisation des paquets de ressources

Pour utiliser un paquet de ressources, enregistrez-le dans une vue en appelant la méthode `yii\web\AssetBundle::register()`. Par exemple, dans un modèle de vue, vous pouvez enregistrer un paquet de ressources de la manière suivante :

```
use app\assets\AppAsset;
AppAsset::register($this);  // $this représente l'objet *view* (vue)
```

> **Info :** la méthode `yii\web\AssetBundle::register()` retourne un objet paquet de ressources contenant les informations sur les ressources publiées, telles que le `chemin de base` ou l'`URL de base`.

Si vous êtes en train d'enregistrer un paquet de ressources dans d'autres endroits, vous devez fournir l'objet *view* requis. Par exemple, pour enregistrer un paquet de ressources dans une classe d'objet graphique, vous pouvez obtenir l'objet *view* avec l'expression `$this->view`.

Lorsqu'un paquet de ressources est enregistré avec une vue, en arrière plan. Yii enregistre tous les paquets de ressources dont il dépend. Et si un paquet de ressources est situé dans un dossier inaccessible depuis le Web, il est publié dans un dossier accessible depuis le Web. Plus tard, lorsque la vue rend une page, elle génère les balises `<link>` et `<script>` pour les fichiers CSS et JavaScript listés dans le paquet de ressources enregistré. L'ordre des ces balises est déterminé par les dépendances entre paquets enregistrés et l'ordre des ressources listées dans les propriétés `yii\web\AssetBundle::$css` et `yii\web\AssetBundle::$js`.

Paquets de ressources dynamiques

Une classe PHP ordinaire de paquet de ressources peut comporter sa propre logique et peut ajuster ses paramètres internes dynamiquement. Par exemple : il se peut que vous utilisiez une bibliothèque JavaScript sophistiquée qui des ressources d'internationalisation dans des fichiers séparés pour chacune des langues. En conséquence de quoi, vous devez ajouter certains fichiers '.js' particuliers à votre page pour la fonction de traduction de la bibliothèque fonctionne. Cela peut être fait en redéfinissant la méthode `yii\web\AssetBundle::init()` :

```
namespace app\assets;

use yii\web\AssetBundle;
use Yii;

class SophisticatedAssetBundle extends AssetBundle
{
    public $sourcePath = '/path/to/sophisticated/src';
    public $js = [
        'sophisticated.js' // fichier toujours utilisé
    ];

    public function init()
    {
        parent::init();
        $this->js[] = 'i18n/' . Yii::$app->language . '.js'; // fichier
        dynamique ajouté
    }
}
```

Un paquet de ressources particuliers peut aussi être ajusté via son instance retourné par `yii\web\AssetBundle::register()`. Par exemple :

```
use app\assets\SophisticatedAssetBundle;
use Yii;

$bundle = SophisticatedAssetBundle::register(Yii::$app->view);
$bundle->js[] = 'i18n/' . Yii::$app->language . '.js'; // fichier dynamique
ajouté
```

> Note : bien que l'ajustement dynamique des paquets de ressources soit pris e charge, c'est une **mauvaise** pratique qui peut conduire à des effets de bord inattendus et qui devrait être évité si possible.

Personnalisation des paquets de ressources

Yii gère les paquets de ressources à l'aide d'un composant d'application nommé `assetManager` (gestionnaire de ressources) qui est mis œuvre par `yii\web\AssetManager`. En configurant la propriété `yii\web\AssetManager`

`::$bundles`, il est possible de personnaliser le comportement d'un paquet de ressources. Par exemple, le paquet de ressources par défaut `yii\web\JqueryAsset` utilise le fichier `jquery.js` du paquet Bower installé. Pour améliorer la disponibilité et la performance, vous désirez peut-être utiliser une version hébergée par Google. Vous pouvez le faire en configurant `assetManager` dans la configuration de l'application comme ceci :

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => [
                    'sourcePath' => null,   // ne pas publier le paquet
                    'js' => [
                        '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',
                    ]
                ],
            ],
        ],
    ],
];
```

Vous pouvez configurer de multiples paquets de ressources de manière similaire via `yii\web\AssetManager::$bundles`. Les clés du tableau doivent être les nom des classes (sans la barre oblique inversée de tête) des paquets de ressources, et les valeurs du tableau doivent être les tableaux de configuration correspondants.

> **Conseil :** vous pouvez choisir quelles ressources utiliser dans un paquet en fonction d'une condition. L'exemple suivant montre comment utiliser `jquery.js` dans l'environnement de développement et `jquery.min.js` autrement :
>
> ```
> 'yii\web\JqueryAsset' => [
>     'js' => [
>         YII_ENV_DEV ? 'jquery.js' : 'jquery.min.js'
>     ]
> ],
> ```

Vous pouvez désactiver un ou plusieurs paquets de ressources en associant `false` (faux) aux noms des paquets de ressources que vous voulez désactiver. Lorsque vous enregistrez un paquet de ressources dans une vue, aucun des paquets dont il dépend n'est enregistré, et la vue, elle non plus, n'inclut aucune des ressources du paquet dans la page qu'elle rend. Par exemple, pour désactiver `yii\web\JqueryAsset`, vous pouvez utiliser la configuration suivante :

```
return [
    // ...
```

```
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => false,
            ],
        ],
    ],
];
```

Vous pouvez aussi désactiver *tous* les paquets de ressources en définissant `yii\web\AssetManager::$bundles` à la valeur `false`.

Mise en correspondance des ressources

Parfois, vous désirez « corriger » des chemins de fichiers de ressources incorrects ou incompatibles utilisés par plusieurs paquets de ressources. Par exemple, un paquet A utilise `jquery.min.js` version 1.11.1, et un paquet B utilise `jquery.js` version 2.1.1. Bien que vous puissiez corriger le problème en personnalisant chacun des paquets, une façon plus facile est d'utiliser la fonctionnalité *mise en correspondance des ressources* pour mettre en correspondance les ressources incorrectes avec celles désirées. Pour le faire, configurez la propriété [[yii\web\AssetManager : :assetMap (table de mise en correspondance des ressources)]] comme indiqué ci-après :

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'assetMap' => [
                'jquery.js' =>
                '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',
            ],
        ],
    ],
];
```

Les clés de la `table de mise en correspondance des ressources` sont les noms des ressources que vous voulez corriger, et les valeurs sont les chemins des ressources désirées. Lorsque vous enregistrez un paquet de ressources dans une vue, chacune des ressources relatives dans ses tableaux `css` et `js` sont examinées dans cette table. Si une des clés est trouvée comme étant la dernière partie d'un chemin de fichier de ressources (qui est préfixé par le [[yii\web\AssetBundle : :chemin des sources si disponible)]], la valeur correspondante remplace la ressource et est enregistrée avec la vue. For exemple, le fichier de ressources `my/path/to/jquery.js` correspond à la clé `jquery.js`.

> **Note :** seules les ressources spécifiées en utilisant des chemins relatifs peuvent faire l'objet d'une mise en correspondance. Les chemins de ressources cibles doivent être soit des URL absolues, soit des chemins relatifs à `yii\web\AssetManager::$basePath`.

Publication des ressources

Comme mentionné plus haut, si un paquet de ressources est situé dans un dossier non accessible depuis le Web, ses ressources sont copiées dans un dossier Web lorsque le paquet est enregistré dans une vue. Ce processus est appelé *publication des ressources* et est accompli automatiquement par le `gestionnaire de ressources`.

Par défaut, les ressources sont publiées dans le dossier `@webroot/assets` qui correspond à l'URL `@web/assets`. Vous pouvez personnaliser cet emplacement en configurant les propriétés `basePath` et `baseUrl`.

Au lieu de publier les ressources en copiant les fichiers, vous pouvez envisager d'utiliser des liens symboliques, si votre système d'exploitation et votre serveur Web le permettent. Cette fonctionnalité peut être activée en définissant la propriété `linkAssets` à `true` (vrai).

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'linkAssets' => true,
        ],
    ],
];
```

Avec la configuration ci-dessus, le gestionnaire de ressources crée un lien symbolique vers le chemin des sources d'un paquet de ressources lors de sa publication. Cela est plus rapide que la copie de fichiers et peut également garantir que les ressources publiées sont toujours à jour.

Fonctionnalité d'affranchissement du cache

Pour les application Web tournant en mode production, une pratique courante consiste à activer la mise en cache HTTP pour les ressources statiques. Un inconvénient de cette pratique est que si vous modifiez une ressource et la republiez en production, le navigateur peut toujours utiliser l'ancienne version à cause de la mise en cache HTTP. Pour s'affranchir de cet inconvénient, vous pouvez utiliser la fonctionnalité d'affranchissement du cache qui a été introduite dans la version 2.0.3 en configurant le gestionnaire de ressources `yii\web\AssetManager` comme suit :

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'appendTimestamp' => true,
        ],
    ],
];
```

Ce faisant, l'horodatage de la dernière modification du fichier est ajoutée en fin d'URL de la ressource publiée. Par exemple, l'URL vers `yii.js` ressemble à `/assets/5515a87c/yii.js?v=1423448645"`, où `v` représente l'horodatage de la dernière modification du fichier `yii.js`. Désormais, si vous modifiez une ressource, son URL change également ce qui force le navigateur à aller chercher la dernière version de la ressource.

3.11.4 Paquets de ressources couramment utilisés

Le code du noyau de Yii a défini de nombreux paquets de ressources. Parmi eux, les paquets suivants sont couramment utilisés et peuvent être référencés dans le code de votre application ou de votre extension.

- `yii\web\YiiAsset` : ce paquet comprend essentiellement le fichier `yii.js` qui met en œuvre un mécanisme d'organisation du code JavaScript en modules. Il fournit également une prise en charge spéciale des attributs `data-method` et `data-confirm` et autres fonctionnalités utiles.
- `yii\web\JqueryAsset` : ce paquet comprend le fichier `jquery.js` du paquet Bower de jQuery.
- `yii\bootstrap\BootstrapAsset` : ce paquet inclut le fichier CSS du framework Twitter Bootstrap.
- `yii\bootstrap\BootstrapPluginAsset` : ce paquet inclut le fichier JavaScript du framework Twitter Bootstrap pour la prise en charge des greffons JavaScript de Bootstrap.
- `yii\jui\JuiAsset` : ce paquet inclut les fichiers CSS et JavaScript de la bibliothèque jQuery UI.

Si votre code dépend de jQuery, jQuery UI ou Bootstrap, vous devriez utiliser les paquets de ressources prédéfinis plutôt que de créer vos propres versions. Si les réglages par défaut des ces paquets de ressources prédéfinis ne répondent pas à vos besoins, vous pouvez les personnaliser comme expliqué à la sous-section Personnalisation des paquets de ressources.

3.11.5 Conversion de ressources

Au lieu d'écrire directement leur code CSS et/ou JavaScript, les développeurs l'écrivent souvent dans une syntaxe étendue et utilisent des outils spéciaux pour le convertir en CSS/JavaScript. Par exemple, pour le code CSS vous pouvez utiliser LESS [29] ou SCSS [30] ; et pour JavaScript, vous pouvez utiliser TypeScript [31].

Vous pouvez lister les fichiers de ressources écrits dans une syntaxe étendue dans les propriétés `css` et `js` d'un paquet de ressources.

29. https://lesscss.org/
30. https://sass-lang.com/
31. https://www.typescriptlang.org/

```
class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.less',
    ];
    public $js = [
        'js/site.ts',
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

Lorsque vous enregistrez une tel paquet de ressources dans une vue, le `gestionnaire de ressources` exécute automatiquement l'outil de pré-traitement pour convertir les ressources, écrites dans une syntaxe reconnue, en CSS/JavaScript. Lorsque la vue rend finalement la page, elle inclut les fichiers CSS/JavaScript dans la page, au lieu des ressources originales écrites dans la syntaxe étendue.

Yii utilise l'extension du nom de fichier pour identifier dans quelle syntaxe une ressource est écrite. Par défaut, il reconnaît les syntaxes et les extensions de nom suivants :

— LESS [32] : `.less`
— SCSS [33] : `.scss`
— Stylus [34] : `.styl`
— CoffeeScript [35] : `.coffee`
— TypeScript [36] : `.ts`

Yii se fie aux outils de pré-traitement installés pour convertir les ressources. Par exemple, pour utiliser LESS [37], vous devriez utiliser la commande de pré-traitement `lessc`.

Vous pouvez personnaliser les commandes de pré-traitement et la syntaxe étendue prise en charge en configurant `yii\web\AssetManager::$converter` comme ci-après :

```
return [
    'components' => [
        'assetManager' => [
            'converter' => [
                'class' => 'yii\web\AssetConverter',
                'commands' => [
                    'less' => ['css', 'lessc {from} {to} --no-color'],
```

32. https://lesscss.org/
33. https://sass-lang.com/
34. https://stylus-lang.com/
35. https://coffeescript.org/
36. https://www.typescriptlang.org/
37. https://lesscss.org/

```
                'ts' => ['js', 'tsc --out {to} {from}'],
            ],
        ],
    ],
],
];
```

Dans la syntaxe précédente, nous spécifions les syntaxes étendues prises en charge via la propriété `yii\web\AssetConverter::$commands`. Les clés du tableau sont les extensions de nom de fichier (sans le point de tête), et les valeurs sont les extensions des fichiers de ressources résultants ainsi que les commandes pour effectuer les conversions. Les valeurs à remplacer `{from}` et `{to}` dans les commandes doivent être remplacées par les chemins de fichiers de ressources sources et les chemins de fichiers de ressources cibles.

> **Info :** il y a d'autres manières de travailler avec les ressources en syntaxe étendue, en plus de celle décrite ci-dessus. Par exemple, vous pouvez utiliser des outils de compilation comme grunt [38] pour surveiller et convertir automatiquement des ressources écrites en syntaxe étendue. Dans ce cas, vous devez lister les fichiers CSS/JavaScript résultants dans des paquets de ressources plutôt que les fichiers originaux.

3.11.6 Combinaison et compression de ressources

Une page Web peut inclure plusieurs fichiers CSS et/ou JavaScript. Pour réduire le nombre de requêtes HTTP et la taille des fichiers téléchargés, une pratique courante est de combiner et compresser ces fichiers CSS/JavaScript multiples en un ou très peu de fichiers, et d'inclure ces fichiers compressés dans les pages Web à la place des fichiers originaux.

> **Info :** la combinaison et la compression de ressources sont généralement nécessaires lorsqu'une application est dans le mode production. En mode développement, l'utilisation des fichiers CSS/JavaScript originaux est souvent plus pratique pour des raisons de débogage plus facile.

Dans ce qui est présenté ci-dessous, nous introduisons une approche pour combiner et compresser les fichiers de ressources sans avoir besoin de modifier le code existant.

1. Identifier tous les paquets de ressources dans l'application que vous envisagez de combiner et de compresser.
2. Diviser ces paquets en un ou quelques groupes. Notez que chaque paquet ne peut appartenir qu'à un seul groupe.

38. https://gruntjs.com/

3. Combiner/compresser les fichiers CSS de chacun des groupes en un fichier unique. Faire de même avec les fichiers JavaScript.
4. Définir un nouveau paquet de ressources pour chacun des groupes :
   — Définir les propriétés `css` et `js` comme étant les fichiers CSS et JavaScript combinés, respectivement.
   — Personnaliser les paquets de ressources dans chacun des groupes en définissant leurs propriétés `css` et `js` comme étant vides, et en définissant leur propriété `depends` comme étant le nouveau paquet de ressources créé pour le groupe.

En utilisant cette approche, lorsque vous enregistrez un paquet de ressources dans une vue, cela engendre un enregistrement automatique du nouveau paquet de ressources pour le groupe auquel le paquet original appartient. Et, en conséquence, les fichiers de ressources combinés/compressés sont inclus dans la page à la place des fichiers originaux.

Un exemple

Examinons ensemble un exemple pour expliquer plus précisément l'approche ci-dessus.

Supposons que votre application possède deux pages X et Y. La page X utilise les paquets de ressources A, B et C, tandis que la page Y utilise les paquets des ressources B, C et D.

Vous avez deux possibilités pour diviser ces paquets de ressources. La première consiste à utiliser un groupe unique pour y inclure tous les paquets de ressources, la seconde est de mettre A dans un groupe X, D dans un groupe Y et (B,C) dans un groupe S. Laquelle des deux est la meilleure ? Cela dépend. La première possibilité offre l'avantage que les deux pages partagent les mêmes fichiers CSS et JavaScript combinés, ce qui rend la mise en cache HTTP plus efficace. Cependant, comme le groupe unique contient tous les paquets, la taille des fichiers combinés CSS et JavaScript est plus importante et accroît donc le temps de transmission initial. Par souci de simplification, dans cet exemple, nous utiliserons la première possibilité, c'est à dire, un groupe unique contenant tous les paquets.

> **Info :** la division des paquets de ressources en groupes, n'est pas une tâche triviale. Cela requiert généralement une analyse du trafic réel des données des différentes ressources sur différentes pages. Au début, vous pouvez démarrer avec un groupe unique par souci de simplification.

Utilisez les outils existants (p. ex. Closure Compiler [39], YUI Compressor](https://github.com/yui/yuicompressor/)) pour combiner et compresser les fichiers CSS et JavaScript dans tous les paquets. Notez que les fichiers doivent

39. https://developers.google.com/closure/compiler/

être combinés dans l'ordre qui permet de satisfaire toutes les dépendances entre paquets. Par exemple, si le paquet A dépend du paquet B, qui dépend lui-même du paquet C et du paquet D, alors vous devez lister les fichiers de ressources en commençant par C et D, suivi de B et, pour finir, A.

Après avoir combiné et compressé, nous obtenons un fichier CSS et un fichier JavaScript. Supposons qu'ils s'appellent `all-xyz.css` et `all-xyz.js`, où `xyz` est un horodatage ou une valeur de hachage qui est utilisé pour rendre le nom de fichier unique afin d'éviter les problèmes de mise en cache HTTP.

Nous en sommes au dernier stade maintenant. Configurez le `gestionnaire de ressources` dans la configuration de l'application comme indiqué ci-dessous :

```
return [
    'components' => [
        'assetManager' => [
            'bundles' => [
                'all' => [
                    'class' => 'yii\web\AssetBundle',
                    'basePath' => '@webroot/assets',
                    'baseUrl' => '@web/assets',
                    'css' => ['all-xyz.css'],
                    'js' => ['all-xyz.js'],
                ],
                'A' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'B' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'C' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'D' => ['css' => [], 'js' => [], 'depends' => ['all']],
            ],
        ],
    ],
];
```

Comme c'est expliqué dans la sous-section Personnalisation des paquets de ressources, la configuration ci-dessus modifie le comportement par défaut des chacun des paquets. En particulier, les paquets A, B, C et D ne possèdent plus aucun fichier de ressources. Ils dépendent tous du paquet `all` qui contient les fichiers combinés `all-xyz.css` et `all-xyz.js`. Par conséquent, pour la page X, au lieu d'inclure les fichiers sources originaux des paquets A, B et C, seuls ces deux fichiers combinés sont inclus ; la même chose se passe par la page Y.

Il y a un truc final pour rendre l'approche ci-dessus plus lisse. Au lieu de modifier directement le fichier de configuration de l'application, vous pouvez mettre le tableau de personnalisation dans un fichier séparé et l'inclure dans la configuration de l'application en fonction d'une condition. Par exemple :

```
return [
    'components' => [
        'assetManager' => [
```

```
            'bundles' => require __DIR__ . '/' . (YII_ENV_PROD ?
            'assets-prod.php' : 'assets-dev.php'),
        ],
    ],
];
```

Cela veut dire que le tableau de configuration du paquet de ressources est sauvegardé dans `assets-prod.php` pour le mode production, et `assets-dev.php` pour les autres modes.

Utilisation de la commande

Yii fournit une commande de console nommée `asset` pour automatiser l'approche que nous venons juste de décrire.

Pour utiliser cette commande, vous devez d'abord créer un fichier de configuration pour décrire quels paquets de ressources seront combinés et comment ils seront regroupés. Vous pouvez utiliser la sous-commande `asset/template` pour créer d'abord un modèle, puis le modifier pour l'adapter à vos besoins.

```
yii asset/template assets.php
```

La commande génère un fichier `assets.php` dans le dossier courant. Le contenu de ce fichier ressemble à ce qui suit :

```
<?php
/**
 * Fichier de configuration pour la commande de console "yii asset".
 * Notez que dans l'environnement console, quelques alias de chemin comme
'@webroot' et '@web' peuvent ne pas exister.
 * Pensez à définir ces alias de chemin manquants.
 */
return [
    // Ajuste la commande/fonction de rappel pour la compression des
    fichiers JavaScript :
    'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_file
    {to}',
    // Ajuste la commande/fonction de rappel pour la compression des
    fichiers   CSS :
    'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o
    {to}',
    // La liste des paquets de ressources à compresser :
    'bundles' => [
        // 'yii\web\YiiAsset',
        // 'yii\web\JqueryAsset',
    ],
    // Paquets de ressources par la sortie de compression :
    'targets' => [
        'all' => [
            'class' => 'yii\web\AssetBundle',
            'basePath' => '@webroot/assets',
            'baseUrl' => '@web/assets',
```

```
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
        ],
    ],
    // Configuration du gestionnaire de ressources :
    'assetManager' => [
    ],
];
```

Vous devez modifier ce fichier et spécifier quels paquets vous envisagez de combiner dans l'option `bundles`. Dans l'option `targets` vous devez spécifier comment les paquets sont divisés en groupes. Vous pouvez spécifier un ou plusieurs groupes, comme nous l'avons déjà dit.

> **Note :** comme les alias `@webroot` et `@web` ne sont pas disponibles dans l'application console, vous devez les définir explicitement dans la configuration.

Les fichiers JavaScript sont combinés, compressés et écrits dans `js/all-{hash}.js` où {hash} est une valeur à remplacer par la valeur de hachage du fichier résultant.

Les options `jsCompressor` et `cssCompressor` spécifient les commandes de console ou les fonctions de rappel PHP pour effectuer la combinaison/compression des fichiers JavaScript et CSS. Par défaut, Yii utilise Closure Compiler [40] pour combiner les fichiers JavaScript et YUI Compressor [41] pour combiner les fichiers CSS. Vous devez installer ces outils à la main ou ajuster ces options pour utiliser vos outils favoris.

Avec le fichier de configuration, vous pouvez exécuter la commande `asset` pour combiner et compresser les fichiers de ressources et générer un nouveau fichier de configuration de paquet de ressources `assets-prod.php` :

```
yii asset assets.php config/assets-prod.php
```

Le fichier de configuration peut être inclus dans la configuration de l'application comme décrit dans la dernière sous-section .

> **Info :** l'utilisation de la commande `asset` n'est pas la seule option pour automatiser la combinaison et la compression des ressources. Vous pouvez utiliser l'excellent outil d'exécution de tâches grunt [42] pour arriver au même résultat.

Regroupement des paquets de ressources

Dans la dernière sous-section présentée, nous avons expliqué comment combiner tous les paquets de ressources en un seul de manière à minimiser

40. https://developers.google.com/closure/compiler/
41. https://github.com/yui/yuicompressor/
42. https://gruntjs.com/

les requêtes HTTP pour les fichiers de ressources utilisés par l'application. Ce n'est pas toujours une pratique souhaitable. Par exemple, imaginez que votre application dispose d'une interface utilisateur (*frontend*) et d'une interface d'administration (*backend*), lesquelles utilisent un jeu différent de fichiers CSS et JavaScript. Dans un tel cas, combiner les paquets de ressources des deux interfaces en un seul n'a pas beaucoup de sens, parce que les paquets de ressources pour l'interface utilisateur ne sont pas utilisés par l'interface d'administration, et parce que cela conduit à un gâchis de bande passante du réseau d'envoyer les ressources de l'interface d'administration lorsqu'une page du l'interface utilisateur est demandée.

Pour résoudre ce problème, vous pouvez diviser les paquets de ressources en groupes et combiner les paquets de ressources de chacun des groupes. La configuration suivante montre comment vous pouvez grouper les paquets de ressources : `php return [

```
...
// Specifie les paquets de sortie par groupe :
'targets' => [
    'allShared' => [
        'js' => 'js/all-shared-{hash}.js',
        'css' => 'css/all-shared-{hash}.css',
        'depends' => [
            // Include all assets shared between 'backend' and 'frontend'
            'yii\web\YiiAsset',
            'app\assets\SharedAsset',
        ],
    ],
    'allBackEnd' => [
        'js' => 'js/all-{hash}.js',
        'css' => 'css/all-{hash}.css',
        'depends' => [
            // Include only 'backend' assets:
            'app\assets\AdminAsset'
        ],
    ],
    'allFrontEnd' => [
        'js' => 'js/all-{hash}.js',
        'css' => 'css/all-{hash}.css',
        'depends' => [], // Include all remaining assets
    ],
],
...

]; `
```

Comme vous le voyez, les paquets de ressources sont divisés en trois groupes : `allShared`, `allBackEnd` et `allFrontEnd`. Ils dépendent tous d'un jeu approprié de paquets de ressources. Par exemple, `allBackEnd` dépend de `app\assets\AdminAsset`. En exécutant la commande `asset` avec cette configuration, les paquets de ressources sont combinés en respectant les spécifications ci-dessus.

> **Info :** vous pouvez laisser la configuration de `depends` vide pour l'un des paquets cible. Ce faisant, ce paquet de ressources dépendra de tous les paquets de ressources dont aucun autre paquet de ressources ne dépend.

3.12 Extensions

Les extensions sont des paquets logiciels distribuables, spécialement conçus pour être utilisés dans des applications, et qui procurent des fonctionnalités prêtes à l'emploi. Par exemple, l'extension yiisoft/yii2-debug [43] ajoute une barre de débogage très pratique au pied de chaque page dans votre application pour vous aider à comprendre plus aisément comment les pages sont générées. Vous pouvez utiliser des extensions pour accélérer votre processus de développement. Vous pouvez aussi empaqueter votre code sous forme d'extensions pour partager votre travail avec d'autres personnes.

> **Info :** nous utilisons le terme "extension" pour faire référence à des paquets logiciels spécifiques à Yii. Quant aux paquets à but plus général, qui peuvent être utilisés en dehors de Yii, nous y faisons référence en utilisant les termes « paquet » ou « bibliothèque ».

3.12.1 Utilisation des extensions

Pour utiliser une extension, vous devez d'abord l'installer. La plupart des extensions sont distribuées en tant que paquets Composer [44] qui peuvent être installés en suivant les deux étapes suivantes :

1. Modifier le fichier `composer.json` de votre application et spécifier quelles extensions (paquets Composer) vous désirez installer.
2. Exécuter la commande `composer install` pour installer les extensions spécifiées.

Notez que devez installer Composer [45] si vous ne l'avez pas déjà fait.

Par défaut, Composer installe les paquets enregistrés sur Packagist [46] — le plus grand dépôt pour les paquets Composer Open Source. Vous pouvez rechercher des extensions sur Packagist. Vous pouvez aussi créer votre propre dépôt [47] et configurer Composer pour l'utiliser. Ceci est utile si vous développez des extensions privées que vous ne voulez partager que dans vos propres projets seulement.

43. https://github.com/yiisoft/yii2-debug
44. https://getcomposer.org/
45. https://getcomposer.org/
46. https://packagist.org/
47. https://getcomposer.org/doc/05-repositories.md#repository

Les extensions installées par Composer sont stockées dans le dossier `BasePath/vendor`, où `BasePath` fait référence au chemin de base de l'application. Comme Composer est un gestionnaire de dépendances, quand il installe un paquet, il installe aussi automatiquement tous les paquets dont le paquet dépend.

Par exemple, pour installer l'extension `yiisoft/yii2-imagine`, modifier votre fichier `composer.json` comme indiqué ci-après :

```
{
    // ...

    "require": {
        // ... autres dépendances

        "yiisoft/yii2-imagine": "~2.0.0"
    }
}
```

Après l'installation, vous devriez apercevoir le dossier `yiisoft/yii2-imagine` dans le dossier `BasePath/vendor`. Vous devriez également apercevoir un autre dossier `imagine/imagine` contenant les paquets dont l'extension dépend et qui ont été installés.

> **Info :** l'extension `yiisoft/yii2-imagine` est une extension du noyau développée et maintenue par l'équipe de développement de Yii. Toutes les extensions du noyau sont hébergées sur Packagist [48] et nommées selon le format `yiisoft/yii2-xyz`, où `xyz` varie selon l'extension.

Vous pouvez désormais utiliser les extensions installées comme si elles faisaient partie de votre application. L'exemple suivant montre comment vous pouvez utiliser la classe `yii\imagine\Image` que l'extension `yiisoft/yii2-imagine` fournit :

```
use Yii;
use yii\imagine\Image;

// generate a thumbnail image
Image::thumbnail('@webroot/img/test-image.jpg', 120, 120)
    ->save(Yii::getAlias('@runtime/thumb-test-image.jpg'), ['quality' =>
    50]);
```

> **Info :** les classes d'extension sont chargées automatiquement par la classe de chargement automatique de Yii (*autoloader*).

48. https://packagist.org/

Installation manuelle d'extensions

Dans quelques cas rares, vous désirez installer quelques, ou toutes les, extensions manuellement, plutôt que de vous en remettre à Composer. Pour le faire, vous devez :

1. Télécharger les archives des extensions et les décompresser dans le dossier `vendor`.
2. Installer la classe *autoloader* procurée par les extensions, si elles en possèdent.
3. Télécharger et installer toutes les extensions dont vos extensions dépendent selon les instructions.

Si une extension ne possède pas de classe *autoloader* mais obéit à la norme PSR-4 [49], vous pouvez utiliser la classe *autoloader* procurée par Yii pour charger automatiquement les classes d'extension. Tout ce que vous avez à faire, c'est de déclarer un alias racine pour le dossier racine de l'extension. Par exemple, en supposant que vous avez installé une extension dans le dossier `vendor/mycompany/myext`, et que les classes d'extension sont sous l'espace de noms `myext`, alors vous pouvez inclure le code suivant dans la configuration de votre application :

```
[
    'aliases' => [
        '@myext' => '@vendor/mycompany/myext',
    ],
]
```

3.12.2 Création d'extensions

Vous pouvez envisager de créer une extension lorsque vous ressentez l'envie de partager votre code avec d'autres personnes. Une extension pour contenir n'importe quel code à votre goût, comme une classe d'aide, un objet graphique, un module, etc.

Il est recommandé de créer une extension sous la forme d'un paquet Composer [50] de façon à ce qu'elle puisse être installée facilement par d'autres utilisateurs, comme nous l'avons expliqué dans la sous-section précédente.

Ci-dessous, nous présentons les étapes de base à suivre pour créer une extension en tant que paquet Composer.

1. Créer un projet pour votre extension et l'héberger dans un dépôt VCS, tel que github.com [51]. Le travail de développement et de maintenance pour cette extension doit être fait sur ce dépôt.

49. https://www.php-fig.org/psr/psr-4/
50. https://getcomposer.org/
51. https://github.com

2. Dans le dossier racine du projet, créez un fichier nommé `composer.json` comme le réclame Composer. Reportez-vous à la sous-section suivante pour plus de détails.
3. Enregistrez votre extension dans un dépôt Composer tel que Packagist [52], afin que les autres utilisateurs puissent la trouver et l'installer avec Composer.

Tout paquet Composer doit disposer d'un fichier `composer.json` dans son dossier racine. Ce fichier contient les méta-données à propos du paquet. Vous pouvez trouver une spécification complète de ce fichier dans le manuel de Composer [53]. L'exemple suivant montre le fichier `composer.json` de l'extension `yiisoft/yii2-imagine` :

```
{
    // package name (nom du paquet)
    "name": "yiisoft/yii2-imagine",

    // package type (type du paquet)
    "type": "yii2-extension",

    "description": "l'intégration d'Imagine pour le framework Yii ",
    "keywords": ["yii2", "imagine", "image", "helper"],
    "license": "BSD-3-Clause",
    "support": {
        "issues":
        "https://github.com/yiisoft/yii2/issues?labels=ext%3Aimagine",
        "forum": "https://forum.yiiframework.com/",
        "wiki": "https://www.yiiframework.com/wiki/",
        "irc": "ircs://irc.libera.chat:6697/yii",
        "source": "https://github.com/yiisoft/yii2"
    },
    "authors": [
        {
            "name": "Antonio Ramirez",
            "email": "amigo.cobos@gmail.com"
        }
    ],

    // dépendances du paquet
    "require": {
        "yiisoft/yii2": "~2.0.0",
        "imagine/imagine": "v0.5.0"
    },

    // class autoloading specs
```

52. https://packagist.org/
53. https://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup

```
    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

Nommage des paquets Chaque paquet Composer doit avoir un nom de paquet qui le distingue des autres paquets. Le format d'un nom de paquet est `vendorName/projectName`. Par exemple, dans le nom de paquet `yiisoft/yii2-imagine`, le nom de vendeur et le nom du projet sont, respectivement, `yiisoft` et`yii2-imagine`.

N'utilisez PAS `yiisoft` comme nom de vendeur car il est réservé pour le noyau de Yii.

Nous recommandons que vous préfixiez votre nom de projet par `yii2-` pour les paquets qui sont des extensions de Yii, par exemple,`myname/yii2-mywidget`. Cela permet aux utilisateurs de distinguer plus facilement les extensions de Yii 2.

Types de paquet Il est important de spécifier le type de paquet de votre extension comme `yii2-extension` afin que le paquet puisse être reconnu comme une extension de Yii lors de son installation.

Losqu'un utilisateur exécute `composer install` pour installer une extension, le fichier `vendor/yiisoft/extensions.php` est automatiquement mis à jour pour inclure les informations sur la nouvelle extension. Grâce à ce fichier, les application Yii peuvent connaître quelles extensions sont installées (l'information est accessible via `yii\base\Application::$extensions`).

Dépendances Bien sûr, votre extension dépend de Yii. C'est pourquoi, vous devez lister (`yiisoft/yii2`) dans l'entrée `require` dans `composer.json`. Si votre extension dépend aussi d'autres extensions ou bibliothèques de tierces parties, vous devez les lister également. Assurez-vous que vous de lister également les contraintes de versions appropriées (p. ex. `1.*`, `@stable`) pour chacun des paquets dont votre extension dépend. Utilisez des dépendances stables lorsque votre extension est publiée dans une version stable.

La plupart des paquets JavaScript/CSS sont gérés par Bower [54] et/ou NPM [55], plutôt que par Composer. Yii utilise le [greffon *assets* de Composer(https://github.com/fxpio/composer-asset-plugin) pour activer la gestion de ce genre de paquets par Composer. Si votre extension dépend d'un paquet Bower, vous pouvez simplement lister la dépendance dans `composer.json` comme ceci :

```
{
    // paquets dépendances
```

54. https://bower.io/
55. https://www.npmjs.com/

```
    "require": {
        "bower-asset/jquery": ">=1.11.*"
    }
}
```

Le code ci-dessus établit que l'extension dépend de paquet Bower `jquery`. En général, vous pouvez utiliser le nom `bower-asset/PackageName` — où `PackageName` est le nom du paquet — pour faire référence à un paquet Bower dans `composer.json`, et utiliser `npm-asset/PackageName` pour faire référence à un paquet NPM. Quand Composer installe un paquet Bower ou NPM, par défaut, le contenu du paquet est installé dans le dossier `@vendor/bower/PackageName` ou `@vendor/npm/Packages`, respectivement. On peut aussi faire référence à ces dossier en utilisant les alias plus courts `@bower/PackageName` et `@npm/PackageName`.

Pour plus de détails sur la gestion des ressources, reportez-vous à la section sur les Ressources.

Chargement automatique des classes Afin que vos classes soient chargées automatiquement par la classe *autoloader* de Yii ou celle de Composer, vous devez spécifier l'entrée `autoload` dans le fichier `composer.json`, comme précisé ci-après :

```
{
    // ....

    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

Vous pouvez lister un ou plusieurs espaces de noms racines et leur chemin de fichier correspondant.

Lorsque l'extension est installée dans une application, Yii crée un alias pour chacun des espaces de noms racines. Cet alias fait référence au dossier correspondant à l'espace de noms. Par exemple, la déclaration `autoload` ci-dessus correspond à un alias nommé `@yii/imagine`.

Pratiques recommandées

Parce que les extensions sont prévues pour être utilisées par d'autres personnes, vous avez souvent besoin de faire un effort supplémentaire pendant le développement. Ci-dessous nous introduisons quelques pratiques courantes et recommandées pour créer des extensions de haute qualité.

Espaces de noms Pour éviter les collisions de noms et rendre le chargement des classes de votre extension automatique, vous devez utiliser des espaces de noms et nommer les classes de votre extension en respectant la norme PSR-4 [56] ou la norme PSR-0 [57].

Vos noms de classe doivent commencer par `vendorName\extensionName`, où `extensionName` est similaire au nom du projet dans le nom du paquet sauf qu'il doit contenir le préfixe `yii2-`. Par exemple, pour l'extension `yiisoft/yii2-imagine`, nous utilisons l'espace de noms `yii\imagine` pour ses classes.

N'utilisez pas `yii`, `yii2` ou `yiisoft` en tant que nom de vendeur. Ces noms sont réservés au code du noyau de Yii.

Classes d'amorçage Parfois, vous désirez que votre extension exécute un certain code durant le processus d'amorçage d'une application. Par exemple, votre extension peut vouloir répondre à l'événement `beginRequest` pour ajuster quelques réglages d'environnement. Bien que vous puissiez donner des instructions aux utilisateurs de l'extension pour qu'ils attachent explicitement votre gestionnaire d'événement dans l'extension à l'événement `beginRequest`, c'est mieux de le faire automatiquement.

Pour ce faire, vous pouvez créer une classe dite *classe du processus d'amorçage* en implémentant l'interface `yii\base\BootstrapInterface`. Par exemple :

```
namespace myname\mywidget;

use yii\base\BootstrapInterface;
use yii\base\Application;

class MyBootstrapClass implements BootstrapInterface
{
    public function bootstrap($app)
    {
        $app->on(Application::EVENT_BEFORE_REQUEST, function () {
            // do something here
        });
    }
}
```

ensuite, listez cette classe dans le fichier `composer.json` de votre extension de cette manière :

```
{
    // ...

    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

56. https://www.php-fig.org/psr/psr-4/
57. https://www.php-fig.org/psr/psr-0/

Lorsque l'extension est installée dans l'application, Yii instancie automatiquement la classe d'amorçage et appelle sa méthode `bootstrap()` durant le processus de démarrage pour chacune des requêtes.

Travail avec des bases de données Votre extension peut avoir besoin d'accéder à des bases de données. Ne partez pas du principe que les applications qui utilisent votre extension utilisent toujours `Yii::$db` en tant que connexion à la base de données. Déclarez plutôt une propriété `db` pour les classes qui requièrent un accès à une base de données. Cette propriété permettra aux utilisateurs de votre extension de personnaliser la connexion qu'ils souhaitent que votre extension utilise. Pour un exemple, reportez-vous à la classe `yii\caching\DbCache` et voyez comment elle déclare et utilise la propriété`db`.

Si votre extension a besoin de créer des tables de base de données spécifiques, ou de faire des changements dans le schéma de la base de données, vous devez :

— fournir des migrations pour manipuler le schéma de base de données, plutôt que d'utiliser des fichiers SQL ;
— essayer de rendre les migrations applicables à différents systèmes de gestion de bases de données ;
— éviter d'utiliser Active Record dans les migrations.

Utilisation des ressources Si votre extension est un objet graphique ou un module, il est probable qu'elle ait besoin de quelques ressources pour fonctionner. Par exemple, un module peut afficher quelques pages qui contiennent des images, du code JavaScript et/ou CSS. Comme les fichiers d'une extension sont tous dans le même dossier, qui n'est pas accessible depuis le Web lorsque l'extension est installée dans une application, vous avez deux possibilités pour rendre ces ressources accessibles depuis le Web.

— demander aux utilisateurs de l'extension de copier les ressources manuellement dans un dossier spécifique accessible depuis le Web ;
— déclarer un paquet de ressources et compter sur le mécanisme de publication automatique des ressources pour copier les fichiers listés dans le paquet de ressources dans un dossier accessible depuis le Web.

Nous recommandons la deuxième approche de façon à ce que votre extension puisse être plus facilement utilisée par d'autres personnes. Reportez-vous à la section Ressources pour plus de détails sur la manière de travailler avec des ressources en général.

Internationalisation et Localisation Votre extension peut être utilisée par des applications prenant en charge différentes langues ! Par conséquent, si votre extension affiche des contenus pour l'utilisateur final, vous devez essayer de traiter à la fois internationalisation et localisation. Plus spécialement :

— Si l'extension affiche des messages pour l'utilisateur final, les messages doivent être enveloppés dans la méthode `Yii::t()` afin de pouvoir être traduits. Les messages à l'attention des développeurs (comme les messages d'exceptions internes) n'ont pas besoin d'être traduits. -Si l'extension affiche des nombres, des dates, etc., ils doivent être formatés en utilisant `yii\i18n\Formatter` avec les règles de formatage appropriées.

Pour plus de détails, reportez-vous à la section Internationalisation.

Tests Vous souhaitez que votre extension s'exécute sans créer de problème à ses utilisateurs. Pour atteindre ce but vous devez la tester avant de la publier.

Il est recommandé que créiez des cas de test variés pour tester votre extension plutôt que de vous fier à des tests manuels. À chaque fois que vous vous apprêterez à publier une nouvelle version de votre extension, vous n'aurez plus qu'à exécuter ces cas de test pour garantir que tout est en ordre. Yii fournit une prise en charge des tests qui peut vous aider à écrire facilement des unités de test, des tests d'acceptation et des tests de fonctionnalités. Pour plus de détails, reportez-vous à la section Tests.

Numérotation des versions Vous devriez donner à chacune des versions publiées de votre extension un numéro (p. ex. `1.0.1`). Nous recommandons de suivre la pratique de la numérotation sémantique des versions [58] lors de la détermination d'un numéro de version.

Publication Pour permettre aux autres personnes de connaître votre extension, vous devez la publier. Si c'est la première fois que vous publiez l'extension, vous devez l'enregistrer sur un dépôt Composer tel que Packagist [59]. Ensuite, tout ce que vous avez à faire, c'est de créer une balise de version (p. ex. `v1.0.1`) sur le dépôt VCS de votre extension et de notifier au dépôt Composer la nouvelle version. Les gens seront capables de trouver votre nouvelle version et, soit de l'installer, soit de la mettre à jour via le dépôt Composer.

Dans les versions de votre extension, en plus des fichiers de code, vous devez envisager d'inclure ce qui suit par aider les gens à connaître votre extension et à l'utiliser :

— Un ficher *readme* (lisez-moi) dans le dossier racine du paquet : il doit décrire ce que fait votre extension, comment l'installer et l'utiliser. Nous vous recommandons de l'écrire dans le format Markdown [60] et de nommer ce fichier `readme.md`.

58. https://semver.org
59. https://packagist.org/
60. https://daringfireball.net/projects/markdown/

— Un fichier *changelog* (journal des modifications) dans le dossier racine du paquet : il liste les changements apportés dans chacune des versions. Ce fichier peut être écrit dans le format Markdown et nommé `changelog.md`.
— Un fichier *upgrade* (mise à jour) dans le dossier racine du paquet : il donne les instructions sur la manière de mettre l'extension à jour en partant d'une version précédente. Ce fichier peut être écrit dans le format Markdown et nommé `upgrade.md`.
— Tutorials, demos, screenshots, etc. : ces derniers sont nécessaires si votre extension fournit de nombreuses fonctionnalités qui ne peuvent être couvertes dans le fichier readme.
— Une documentation de l'API : votre code doit être bien documenté pour permettre aux autres personnes de le lire plus facilement et de le comprendre. Vous pouvez faire référence au fichier de la classe BaseObject [61] pour savoir comment documenter votre code.

  Info : les commentaires de votre code peuvent être écrits dans le format Markdown. L'extension `yiisoft/yii2-apidoc` vous fournit un outil pour générer une documentation d'API agréable et basée sur les commentaires de votre code.

  Info : bien que cela ne soit pas une exigence, nous suggérons que votre extension respecte un certain style de codage. Vous pouvez vous reporter au document style du codage du noyau du framework [62].

3.12.3 Extensions du noyau

Yii fournit les extensions du noyau suivantes (ou "les extensions officielles"[63]) qui sont développées et maintenues par l'équipe de développement de Yii. Elles sont toutes enregistrées surPackagist [64] et peuvent être facilement installées comme décrit dans la sous-section Utilisation des extensions.

— yiisoft/yii2-apidoc [65] : fournit un générateur d'API extensible et de haute performance. Elle est aussi utilisée pour générer l'API du noyau du framework.
— yiisoft/yii2-authclient [66] : fournit un jeu de clients d'authentification courants tels que Facebook OAuth2 client, GitHub OAuth2 client.
— yiisoft/yii2-bootstrap [67] : fournit un jeu d'objets graphiques qui encap-

61. https://github.com/yiisoft/yii2/blob/master/framework/base/BaseObject.php
62. https://github.com/yiisoft/yii2/blob/master/docs/internals/core-code-style.md
63. https://www.yiiframework.com/extensions/official
64. https://packagist.org/
65. https://www.yiiframework.com/extension/yiisoft/yii2-apidoc
66. https://www.yiiframework.com/extension/yiisoft/yii2-authclient
67. https://www.yiiframework.com/extension/yiisoft/yii2-bootstrap

sulent les composants et les greffons de Bootstrap [68].

— yiisoft/yii2-debug [69] : fournit la prise en charge du débogage des applications Yii. Lorsque cette extension est utilisée, une barre de débogage apparaît au pied de chacune des pages. Cette extension fournit aussi un jeu de pages autonomes pour afficher des informations de débogage plus détaillées.
— yiisoft/yii2-elasticsearch [70] : fournit la prise en charge d'Elasticsearch [71]. Elle inclut un moteur de requêtes/recherches de base et met en œuvre le motif Active Record qui permet de stocker des enregistrement actifs dans Elasticsearch.
— yiisoft/yii2-faker [72] : fournit la prise en charge de Faker [73] pour générer des données factices pour vous.
— yiisoft/yii2-gii [74] : fournit un générateur de code basé sur le Web qui est hautement extensible et peut être utilisé pour générer rapidement des modèles, des formulaires, des modules, des requêtes CRUD, etc.
— yiisoft/yii2-httpclient [75] : provides an HTTP client.
— yiisoft/yii2-imagine [76] : fournit des fonctionnalités couramment utilisées de manipulation d'images basées sur Imagine [77].
— yiisoft/yii2-jui [78] : fournit un jeu d'objets graphiques qui encapsulent les interactions et les objets graphiques de JQuery UI [79].
— yiisoft/yii2-mongodb [80] : fournit la prise en charge de MongoDB [81]. Elle inclut des fonctionnalités telles que les requêtes de base, les enregistrements actifs, les migrations, la mise en cache, la génération de code, etc.
— yiisoft/yii2-queue [82] : fournit la prise en charge pour exécuter des tâches en asynchrone via des queues. Il prend en charge les queues en se basant sur, DB, Redis, RabbitMQ, AMQP, Beanstalk et Gearman.
— yiisoft/yii2-redis [83] : fournit la prise en charge de redis [84]. Elle inclut des fonctionnalités telles que les requêtes de base, les enregistrements

68. https://getbootstrap.com/
69. https://www.yiiframework.com/extension/yiisoft/yii2-debug
70. https://www.yiiframework.com/extension/yiisoft/yii2-elasticsearch
71. https://www.elastic.co/
72. https://www.yiiframework.com/extension/yiisoft/yii2-faker
73. https://www.yiiframework.com/extension/fzaninotto/Faker
74. https://www.yiiframework.com/extension/yiisoft/yii2-gii
75. https://www.yiiframework.com/extension/yiisoft/yii2-httpclient
76. https://github.com/yiisoft/yii2-imagine
77. https://www.yiiframework.com/extension/yiisoft/yii2-imagine
78. https://www.yiiframework.com/extension/yiisoft/yii2-jui
79. https://jqueryui.com/
80. https://www.yiiframework.com/extension/yiisoft/yii2-mongodb
81. https://www.mongodb.com/
82. https://www.yiiframework.com/extension/yiisoft/yii2-queue
83. https://github.com/yiisoft/yii2-redis
84. https://redis.io/

actifs, la mise en cache, etc.
— yiisoft/yii2-shell [85] : fournit un interprète de commandes (shell) basé sur psysh [86].
— yiisoft/yii2-smarty [87] : fournit un moteur de modèles basé sur Smarty [88].
— yiisoft/yii2-sphinx [89] : fournit la prise en charge de Sphinx [90]. Elle inclut des fonctionnalités telles que les requêtes de base, les enregistrements actifs, la génération de code, etc.
— yiisoft/yii2-swiftmailer [91] : fournit les fonctionnalités d'envoi de courriels basées sur swiftmailer [92].
— yiisoft/yii2-twig [93] : fournit un moteur de modèles basé sur Twig [94].

Les extensions officielles suivantes sont valables pour les versions Yii 2.1 et plus récentes. Vous n'avez pas besoin de les installer car elles sont incluse dans le cœur du framework.

— yiisoft/yii2-captcha [95] : fournit un CAPTCHA.
— yiisoft/yii2-jquery [96] : fournit une prise en charge de jQuery [97].
— yiisoft/yii2-maskedinput [98] : fournit un composant graphique de saisie masqué basé sur jQuery Input Mask plugin [99].
— yiisoft/yii2-mssql [100] : fournit la prise en charge de MSSQL [101].
— yiisoft/yii2-oracle [102] : fournit la prise en charge de Oracle [103].
— yiisoft/yii2-rest [104] : fournit le support pour l'API REST.

85. https://www.yiiframework.com/extension/yiisoft/yii2-shell
86. https://psysh.org/
87. https://www.yiiframework.com/extension/yiisoft/yii2-smarty
88. https://www.smarty.net/
89. https://github.com/yiisoft/yii2-sphinx
90. https://www.yiiframework.com/extension/yiisoft/yii2-sphinx
91. https://www.yiiframework.com/extension/yiisoft/yii2-swiftmailer
92. https://swiftmailer.symfony.com/
93. https://www.yiiframework.com/extension/yiisoft/yii2-twig
94. https://twig.symfony.com/
95. https://www.yiiframework.com/extension/yiisoft/yii2-captcha
96. https://www.yiiframework.com/extension/yiisoft/yii2-jquery
97. https://jquery.com/
98. https://www.yiiframework.com/extension/yiisoft/yii2-maskedinput
99. https://robinherbots.github.io/Inputmask/
100. https://www.yiiframework.com/extension/yiisoft/yii2-mssql
101. https://www.microsoft.com/sql-server/
102. https://www.yiiframework.com/extension/yiisoft/yii2-oracle
103. https://www.oracle.com/
104. https://www.yiiframework.com/extension/yiisoft/yii2-rest

Chapitre 4

Gérer les Requêtes

4.1 Amorçage

L'amorçage fait référence au processus de préparation de l'environnement avant qu'une application ne démarre, pour résoudre et traiter une requête d'entrée. L'amorçage se fait en deux endroits : le script d'entrée et l'application.

Dans le script d'entrée, les classes de chargement automatique (*autoloaders*) pour différentes bibliothèques sont enregistrées. Cela inclut la classe de chargement automatique de Composer via son fichier `autoload.php` et la classe de chargement automatique de Yii via son fichier de classe `Yii`. Ensuite, le script d'entrée charge la configuration de l'application et crée une instance d'application.

Dans le constructeur de l'application, le travail d'amorçage suivant est effectué :

1. La méthode `preInit()` est appelée. Elle configure quelques propriétés de haute priorité de l'application, comme `le chemin de base (`.
2. Le `gestionnaire d'erreurs` est enregistré.
3. Les propriétés qui utilisent la configuration de l'application sont initialisées.
4. La méthode `init()` est appelée. À son tour elle appelle la méthode `bootstrap()` pour exécuter les composants d'amorçage.
   - Le fichier de manifeste des extensions `vendor/yiisoft/extensions.php` est inclus.
   - Lescomposants d'amorçage déclarés par les extensions sont créés et exécutés
   - Les [composants d'application(structure-application-components.md) et/ou les modules déclarés dans la propriété bootstrap de l'application sont créés et exécutés.

Comme le travail d'amorçage doit être fait avant *chacune* des requêtes, il est très important de conserver ce processus aussi léger et optimisé que possible.

Évitez d'enregistrer trop de composants d'amorçage. Un composant d'amorçage est seulement nécessaire s'il doit participer à tout le cycle de vie de la prise en charge des requêtes. Par exemple,si un module a besoin d'enregistrer des règles d'analyse additionnelles, il doit être listé dans la propriété bootstrap afin que les nouvelles règles d'URL prennent effet avant qu'elles ne soient utilisées pour résoudre des requêtes.

Dans le mode production, activez un cache bytecode, tel que PHP OPcache [1] ou APC [2], pour minimiser le temps nécessaire à l'inclusion et à l'analyse des fichiers PHP.

Quelques applications volumineuses ont des configurations d'application très complexes qui sont divisées en fichiers de configuration plus petits. Si c'est le cas, envisagez de mettre tout le tableau de configuration en cache et de le charger directement à partir cache avant la création de l'instance d'application dans le script d'entrée.

4.2 Routage et création d'URL

Lorsqu'une application Yii commence à traiter une URL objet d'une requête, sa première étape consiste à analyser cette URL pour la résoudre en une route. La route est ensuite utilisée pour instancier l'action de contrôleur correspondante pour la prise en charge de la requête. Ce processus est appelé *routage*.

Le processus inverse du routage, qui consiste à créer une URL à partir d'une route et des paramètres associés de la requête,
est appelé *création d'URL*. Lorsque l'URL créée est ensuite requise, le processus de routage est capable de la résoudre en la route originale avec les paramètres de requête.

L'élément central en charge du routage et de la création d'URL est le `gestionnaire d'URL`, qui est enregistré en tant que composant d'application sous le nom `urlManager`. Le `gestionnaire d'URL` fournit la méthode `parseRequest()` pour analyser une requête entrante et la résoudre en une route et les paramètres de requête associés, et la méthode `createUrl()` pour créer une URL en partant d'une route avec ses paramètres de requête associés.

En configurant le composant `urlManager` dans la configuration de l'application, vous pouvez laisser votre application reconnaître les formats d'URL arbitraires sans modifier le code existant de votre application. Par exemple, vous pouvez utiliser le code suivant pour créer une URL pour l'action `post/view` :

1. https://www.php.net/manual/fr/book.opcache.php
2. https://www.php.net/manual/fr/book.apcu.php

```php
use yii\helpers\Url;

// Url::to() appelle UrlManager::createUrl() pour créer une URL
$url = Url::to(['post/view', 'id' => 100]);
```

Selon la configuration de `urlManager`, l'URL créée peut ressembler à l'une des URL suivantes (ou autre formats). Et si l'URL est requise plus tard, elle sera toujours analysée pour revenir à la route originale et aux valeurs des paramètres de la requête.

```
/index.php?r=post%2Fview&id=100
/index.php/post/100
/posts/100
```

4.2.1 Formats d'URL

Le `gestionnaire d'URL` prend en charge deux formats d'URL :
— le format d'URL par défaut,
— le format d'URL élégantes.

Le format d'URL par défaut utilise un `paramètre de requête` nommé `r` qui représente la route et les paramètres de requête normaux associés à la route. Par exemple, l'URL `/index.php?r=post/view&id=100` represente la route `post/view` et le paramètre de requête `id` dont la valeur est 100. Le format d'URL par défaut ne requiert aucune configuration du [[yii\web\UrlManager|gestionnaire d'URL] et fonctionne dans toutes les configurations de serveur Web.

Le format d'URL élégantes utilise le chemin additionnel qui suit le nom du script d'entrée pour représenter la route et les paramètres de requête associés. Par exemple, le chemin additionnel dans l'URL `/index.php/post/100` est `/post/100` qui, avec une `règle d'URL` appropriée, peut représenter la route `post/view` et le paramètre des requête `id` avec une valeur de 100 . Pour utiliser le format d'URL élégantes, vous devez définir un jeu de `règles d'URL` en cohérence avec les exigences réelles sur la présentation d'une URL.

Vous pouvez passer d'un format d'URL à l'autre en inversant la propriété `enablePrettyUrl` du `gestionnaire d'URL` sans changer quoi que ce soit au code de votre application.

4.2.2 Routage

Le routage se fait en deux étapes :
— La requête entrante est analysée et résolue en une route et les paramètres de requête associés.
— L'action de contrôleur correspondant à la route analysée est créée pour prendre la requête en charge.

Lors de l'utilisation du format d'URL par défaut, la résolution d'une requête en route est aussi simple que d'obtenir le paramètre nommé `r` de la méthode `GET`.

Lors de l'utilisation du format d'URL élégantes, le [[yii\web\UrlManager|gestionnaire d'URL] examine les `règles d'URL` enregistrées pour trouver une règle qui correspond et résoudre la requête en une route. Si une telle règle n'est pas trouvée, une exception `yii\web\NotFoundHttpException` est levée.

Une fois que la requête est résolue en une route, il est temps de créer l'action de contrôleur identifiée par la route. La route est éclatée en de multiples parties par des barres obliques de division. Par exemple, `site/index` est éclatée en `site` et `index`. Chacune des parties est considérée comme un identifiant qui peut faire référence à un module, un contrôleur ou une action. En partant de la première partie dans la route, l'application entreprend les étapes suivantes pour créer un module (s'il en existe un), un contrôleur et une action :

1. Définit l'application comme étant le module courant.
2. Vérifie si la `table de mise en correspondance des contrôleurs` du module courant contient l'identifiant courant. Si c'est le cas, un objet *controller* est créé en respectant la configuration du contrôleur trouvé dans la table de mise en correspondance, et on passe à l'étape 5 pour prendre en compte le reste de la route.
3. Vérifie si l'identifiant fait référence à un module listé dans la propriété `modules` du module courant. Si c'est le cas, un module est créé en respectant la configuration trouvée dans la liste des modules et on passe à l'étape 2 pour prendre en compte le reste de la route dans le contexte du nouveau module.
4. Traite l'identifiant comme un identifiant de contrôleur, crée un objet *controller* et passe à l'étape suivante avec le reste de la route.
5. Le contrôleur recherche l'identifiant courant dans sa `table de mise en correspondance des actions`. S'il le trouve, il crée une action respectant la configuration trouvée dans la table de mise en correspondance. Autrement, le contrôleur essaye de créer une action en ligne dont le nom de méthode correspond à l' identifiant d'action courant.

Si une erreur se produit dans l'une des étapes décrites ci-dessus, une exception `yii\web\NotFoundHttpException` est levée, indiquant l'échec du processus de routage.

Route par défaut

Quand une requête est analysée et résolue en une route vide, la route dite *route par défaut* est utilisée à sa place. Par défaut, la route par défaut est `site/index`, qui fait référence à l'action `index` du contrôleur `site`. Vous pouvez la personnaliser en configurant la propriété `defaultRoute` de l'application dans la configuration de l'application comme indiqué ci-dessous :

```
[
    // ...
    'defaultRoute' => 'main/index',
];
```

De façon similaire à la route par défaut de l'application, il existe aussi une route par défaut pour les modules. Ainsi s'il existe un module `user` (utilisateur) et que la requête est résolue en la route `user`, la propriété `defaultRoute` du module est utilisée pour déterminer le contrôleur. Par défaut, le nom du contrôleur est `default`. Si aucune action n'est spécifiée dans la propriété `defaultRoute`, la propriété `defaultAction` du contrôleur est utilisée pour déterminer l'action. Dans cet exemple, la route complète serait `user/default/index`.

La route

Parfois, vous désirez mettre votre application Web en mode maintenance temporairement et afficher la même page d'information pour toutes les requêtes. Il y a plusieurs moyens de faire cela. L'une des manières les plus simples est de configurer la propriété `yii\web\Application::$catchAll` dans la configuration de l'application comme indiqué ci-dessous :

```
[
    // ...
    'catchAll' => ['site/offline'],
];
```

Avec la configuration ci-dessus, l'action `site/offline` est utilisée pour prendre toutes les requêtes entrantes en charge.

La propriété `catchAll` accepte un tableau dont le premier élément spécifie une route et le reste des éléments des couples clé-valeur pour les paramètres liés à l'action.

> **Info :** le [panneau de débogage]](https://github.com/yiisoft/yii2-debug/blob/master/docs/guide/README.md) de l'environnement de développement ne fonctionne pas lorsque cette propriété est activée.

4.2.3 Création d'URL

Yii fournit une méthode d'aide `yii\helpers\Url::to()` pour créer différentes sortes d'URL à partir de routes données et de leurs paramètres de requête associés. Par exemple :

```
use yii\helpers\Url;

// crée une URL d'une route: /index.php?r=post%2Findex
echo Url::to(['post/index']);
```

```
// crée une URL d'une route avec paramètres :
/index.php?r=post%2Fview&id=100
echo Url::to(['post/view', 'id' => 100]);

// crée une  URL avec ancre : /index.php?r=post%2Fview&id=100#content
echo Url::to(['post/view', 'id' => 100, '#' => 'content']);

// crée une URL absolue : https://www.example.com/index.php?r=post%2Findex
echo Url::to(['post/index'], true);

// crée une URL absolue en utilisant le schéma https :
https://www.example.com/index.php?r=post%2Findex
echo Url::to(['post/index'], 'https');
```

Notez que dans l'exemple ci-dessus, nous supposons que le format d'URL est le format par défaut. Si le format d'URL élégantes est activé, les URL créées sont différentes et respectent les `règles d'URL` en cours d'utilisation.

La route passée à la méthode `yii\helpers\Url::to()` est sensible au contexte. Elle peut être soit *relative*, soit *absolue* et normalisée en respect des règles suivantes :

— Si la route est une chaîne vide, la `route` couramment requise est utilisée ;
— Si la route ne contient aucune barre oblique de division, elle est considérée comme un identifiant d'action du contrôleur courant et est préfixée par la valeur de l'identifiant `uniqueId` du contrôleur courant ;
— Si la route n'a pas de barre oblique de division en tête, elle est considérée comme une route relative au module courant et préfixée par la valeur de l'identifiant `uniqueId` du module courant.

À partir de la version 2.0.2, vous pouvez spécifier une route en terme d'alias. Si c'est le cas, l'alias est d'abord converti en la route réelle qui est ensuite transformée en route absolue dans le respect des règles précédentes.

Par exemple, en supposant que le module courant est `admin` et que le contrôleur courant est `post`,

```
use yii\helpers\Url;

// route couramment requise : /index.php?r=admin%2Fpost%2Findex
echo Url::to(['']);

// une route relative avec un identifiant d'action seulement :
/index.php?r=admin%2Fpost%2Findex
echo Url::to(['index']);

// une route relative : /index.php?r=admin%2Fpost%2Findex
echo Url::to(['post/index']);

// une route absoulue : /index.php?r=post%2Findex
echo Url::to(['/post/index']);
```

```
// /index.php?r=post%2Findex     suppose que l'alias "@posts" est défini
comme  "/post/index"
echo Url::to(['@posts']);
```

La méthode `yii\helpers\Url::to()` est mise en œuvre en appelant les méthodes `createUrl()` et `createAbsoluteUrl()` du `gestionnaire d'URL`. Dans les quelques sous-sections suivantes, nous expliquons comment configurer le `gestionnaire d'URL` pour personnaliser le format des URL créées.

La méthode `yii\helpers\Url::to()` prend aussi en charge la création d'URL qui n'ont **pas** de relation avec des routes particulières. Au lieu de passer un tableau comme premier paramètre, vous devez, dans ce cas, passer une chaîne de caractères. Par exemple :

```
use yii\helpers\Url;

// URL couramment requise : /index.php?r=admin%2Fpost%2Findex
echo Url::to();

// un alias d'URL: https://example.com
Yii::setAlias('@example', 'https://example.com/');
echo Url::to('@example');

// une URL absolue : https://example.com/images/logo.gif
echo Url::to('/images/logo.gif', true);
```

En plus de la méthode `to()`, la classe d'aide `yii\helpers\Url` fournit aussi plusieurs méthode pratiques de création d'URL. Par exemple :

```
use yii\helpers\Url;

// URL de page d'accueil: /index.php?r=site%2Findex
echo Url::home();

// URL de base, utile si l'application est déployée dans un sous-dossier du
dossier Web racine
echo Url::base();

// l'URL canonique de l'URL couramment requise
// voir https://fr.wikipedia.org/wiki/%C3%89l%C3%A9ment_de_lien_canonique
echo Url::canonical();

// mémorise l'URL couramment requise et la retrouve dans les requêtes
subséquentes
Url::remember();
echo Url::previous();
```

4.2.4 Utilisation des URL élégantes

Pour utiliser les URL élégantes, configurez le composant `urlManager` dans la configuration de l'application comme indiqué ci-dessous :

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => false,
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

La propriété `enablePrettyUrl` est obligatoire car elle active/désactive le format d'URL élégantes. Le reste des propriétés est facultatif. Néanmoins, leur configuration montrée plus haut est couramment utilisée.

— `showScriptName` : cette propriété détermine si le script d'entrée doit être inclus dans l'URL créée. Par exemple, au lieu de créer une URL `/index.php/post/100`, en définissant cette propriété à `false`, l'URL `/post/100` est générée.

— `enableStrictParsing` : cette propriété détermine si l'analyse stricte est activée . Si c'est le cas, l'URL entrante doit correspondre à au moins une des **`règles`** afin d'être traitée comme une requête valide, sinon une exception `yii\web\NotFoundHttpException` est levée. Si l'analyse stricte est désactivée, lorsqu'aucune
`règle` ne correspond à l'URL requise, la partie chemin de l'URL est considérée comme étant la route requise.

— `rules` : cette propriété contient une liste de règles spécifiant comme analyser et créer des URL. C'est la propriété principale avec laquelle vous devez travailler afin de créer des URL dont le format satisfait les exigences particulières de votre application.

> **Note :** afin de cacher le nom du script d'entrée dans l'URL créée, en plus de définir la propriété `showScriptName` à `false`, vous pouvez aussi configurer votre serveur Web de manière à ce qu'il puisse identifier correctement quel script PHP doit être exécuté lorsqu'une URL requise n'en précise aucun explicitement. Si vous utilisez un serveur Apache ou nginx, vous pouvez vous reporter à la configuration recommandée décrite dans la section Installation.

Règles d'URL

Une règle d'URL est une classe mettant en œuvre l'interface `yii\web\UrlRuleInterface`, généralement une instance de la classe `yii\web\UrlRule`. Chaque règle d'URL consiste en un motif utilisé pour être mis en correspondance avec la partie chemin de l'URL, une route, et quelques paramètres

de requête. Une règle d'URL peut être utilisée pour analyser une requête si son motif correspond à l'URL requise. Une règle d'URL peut être utilisée pour créer une URL si sa route et le nom de ses paramètres de requête correspondent à ceux qui sont fournis.

Quand le format d'URL élégantes est activé, le `gestionnaire d'URL` utilise les règles d'URL déclarées dans sa propriété `rules` pour analyser les requêtes entrantes et créer des URL. En particulier, pour analyser une requête entrante, le `gestionnaire d'URL` examine les règles dans l'ordre de leur déclaration et cherche la *première* règle qui correspond à l'URL requise. La règle correspondante est ensuite utilisée pour analyser l'URL et la résoudre en une route et ses paramètres de requête associés. De façon similaire, pour créer une URL, le `gestionnaire d'URL` cherche la première règle qui correspond à la route donnée et aux paramètres et l'utilise pour créer l'URL.

Vous pouvez configurer la propriété `yii\web\UrlManager::$rules` sous forme de tableau dont les clés sont les `motifs` et les valeurs, les `routes` correspondantes. Chacune des paires motif-route construit une règle d'URL. Par exemple, la configuration des `règles` suivante déclare deux règles d'URL. La première correspond à l'URL `posts` et la met en correspondance avec la route `post/index`. La seconde correspond à une URL qui correspond à l'expression régulière `post/(\d+)` et la met en correspondance avec la route `post/view` et le paramètre nommé `id`.

```
[
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

> **Info :** le motif dans une règle est utilisé pour correspondre à la partie chemin d'une URL.
> Par exemple, la partie chemin de `/index.php/post/100?source=ad` est `post/100` (les barres obliques de division de début et de fin sont ignorées) et correspond au motif `post/(\d+)`.

En plus de déclarer des règles d'URL sous forme de paires motif-route, vous pouvez aussi les déclarer sous forme de tableaux de configuration. Chacun des tableaux de configuration est utilisé pour configurer un simple objet règle d'URL. C'est souvent nécessaire lorsque vous voulez configurer d'autres propriétés d'une règle d'URL. Par exemple :

```
'rules' => [
    // ...autres règles d'URL...
    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

Par défaut, si vous ne spécifiez pas l'option `class` pour une configuration de règle, elle prend la valeur par défaut `yii\web\UrlRule` qui est la valeur par défaut définie dans `yii\web\UrlManager::$ruleConfig`.

Paramètres nommés

Une règle d'URL peut être associée à quelques paramètres de requête nommés qui sont spécifiés dans le motif et respectent le format `<ParamName:RegExp>`, dans lequel `ParamName` spécifie le nom du paramètre et `RegExp` est une expression régulière facultative utilisée pour établir la correspondance avec une valeur de paramètre. Si `RegExp` n'est pas spécifié, cela signifie que la valeur du paramètre doit être une chaîne de caractères sans aucune barre oblique de division.

> **Note :** vous pouvez seulement spécifier des expressions régulières pour les paramètres. La partie restante du motif est considérée être du texte simple.

Lorsqu'une règle est utilisée pour analyser une URL, elle remplit les paramètres associés avec les valeurs des parties de l'URL qui leur correspondent, et ces paramètres sont rendus disponibles dans `$_GET` et plus tard dans le composant d'application `request`. Lorsque la règle est utilisée pour créer une URL, elle prend les valeurs des paramètres fournis et les insère à l'endroit où ces paramètres sont déclarés.

Prenons quelques exemples pour illustrer comment les paramètres nommés fonctionnent. Supposons que nous ayons déclaré les règles d'URL suivantes :

```
[
    'posts/<year:\d{4}>/<category>' => 'post/index',
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

Lorsque les règles sont utilisées pour analyser des URL :

- `/index.php/posts` est analysée et résolue en la route `post/index` en utilisant la deuxième règle ;
- `/index.php/posts/2014/php` est analysée et résolue en la route `post/index`, le paramètre `year` dont la valeur est 2014 et le paramètre `category` dont la valeur est `php` en utilisant la première règle ;
- `/index.php/post/100` est analysée et résolue en la route `post/view` et le paramètre `id` dont la valeur est 100 en utilisant la troisième règle ;
- `/index.php/posts/php` provoque la levée d'une exception `yii\web\NotFoundHttpException` quand la propriété `yii\web\UrlManager::$enableStrictParsing` est définie à `true`, parce qu'elle ne correspond à aucun des motifs. Si `yii\web\UrlManager::$enableStrictParsing` est définie à `false` (la valeur par défaut), la partie chemin `posts/php` est retournée en tant que

route. Cela provoque l'exécution de l'action correspondante si elle existe, ou lève une exception `yii\web\NotFoundHttpException` autrement.

Et quand les règles sont utilisées pour créer des URL :

- `Url::to(['post/index'])` crée `/index.php/posts` en utilisant la deuxième règle ;
- `Url::to(['post/index', 'year' => 2014, 'category' => 'php'])` crée `/index.php/posts/2014/php` en utilisant la première règle ;
- `Url::to(['post/view', 'id' => 100])` crée `/index.php/post/100` en utilisant la troisième règle ;
- `Url::to(['post/view', 'id' => 100, 'source' => 'ad'])` crée `/index.php/post/100?source=ad` en utilisant la troisième règle. Comme le paramètre `source` n'est pas spécifié dans la règle, il est ajouté en tant que paramètre de requête à l'URL créée.
- `Url::to(['post/index', 'category' => 'php'])` crée `/index.php/post/index?category=php` en utilisant aucune des règles. Notez que, aucune des règles n'étant utilisée, l'URL est créée en ajoutant simplement la route en tant que partie chemin et tous les paramètres en tant que partie de la chaîne de requête.

Paramétrage des routes

Vous pouvez inclure les noms des paramètres dans la route d'une règle d'URL. Cela permet à une règle d'URL d'être utilisée pour correspondre à de multiples routes. Par exemple, les règles suivantes incluent les paramètres `controller` et `action` dans les routes.

```
'rules' => [
    '<controller:(post|comment)>/create' => '<controller>/create',
    '<controller:(post|comment)>/<id:\d+>/<action:(update|delete)>' =>
    '<controller>/<action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/view',
    '<controller:(post|comment)>s' => '<controller>/index',
]
```

Pour analyser l'URL `/index.php/comment/100/update`, la deuxième règle s'applique et définit le paramètre `controller`
comme étant `comment` et le paramètre `action` comme étant `create`. La route `<controller>/<action>` est par conséquent résolue comme `comment/update`.

De façon similaire, pour créer une URL à partir de la route `comment/index`, la dernière règle s'applique, ce qui donne l'URL `/index.php/comments`.

> **Info :** en paramétrant les routes, il est possible de réduire grandement le nombre de règles d'URL, ce qui peut accroître significativement la performance du `gestionnaire d'URL`.

Valeur par défaut des paramètres

Par défaut, tous les paramètres déclarés dans une règle sont requis. Si une URL requise ne contient pas un paramètre particulier, ou si une URL est créée sans un paramètre particulier, la règle ne s'applique pas. Pour rendre certains paramètres facultatifs, vous pouvez configurer la propriété `defaults` de la règle. Les paramètres listés dans cette propriété sont facultatifs et prennent les valeurs spécifiées lorsqu'elles ne sont pas fournies.

Dans la déclaration suivante d'une règle, les paramètres `page` et `tag` sont tous les deux facultatifs et prennent la valeur 1 et vide, respectivement quand ils ne sont pas fournis.

```
[
    // ...autres règles...
    [
        'pattern' => 'posts/<page:\d+>/<tag>',
        'route' => 'post/index',
        'defaults' => ['page' => 1, 'tag' => ''],
    ],
]
```

La règle ci-dessus peut être utilisée pour analyser ou créer l'une quelconque des URL suivantes :

- `/index.php/posts` : `page` est 1, `tag` est ''.
- `/index.php/posts/2` : `page` est 2, `tag` est ''.
- `/index.php/posts/2/news` : `page` est 2, `tag` est `'news'`.
- `/index.php/posts/news` : `page` est 1, `tag` est `'news'`.

Sans les paramètres facultatifs, vous devriez créer quatre règles pour arriver au même résultat.

> **Note :** si `pattern` (motif) ne contient que des paramètres facultatifs et des barres obliques de division, le premier paramètre peut être omis seulement si tous les autres paramètres le sont.

Règles avec des noms de serveur

Il est possible d'inclure des noms de serveur Web dans le motif d'une règle d'URL. Cela est principalement utilisé lorsque votre application doit se comporter différemment selon le nom du serveur Web. Par exemple, les règles suivantes analysent et résolvent l'URL `https://admin.example.com/login` en la route `admin/user/login` et `https://www.example.com/login` en la route `site/login`.

```
[
    'https://admin.example.com/login' => 'admin/user/login',
    'https://www.example.com/login' => 'site/login',
]
```

Vous pouvez aussi inclure des paramètres dans les noms de serveurs pour en extraire de l'information dynamique. Par exemple, la règle suivante analyse et résout l'URL `https://en.example.com/posts` en la route `post/index` et le paramètre `language=en`.

```
[
    'http://<language:\w+>.example.com/posts' => 'post/index',
]
```

Depuis la version 2.0.11, vous pouvez également utiliser des motifs relatifs au protocole qui marchent à la fois pour `http` et `https`. La syntaxe est la même que ci-dessus mais en sautant la partie `http`, p. ex. `'//www.example.com/login' => 'site/login'`.

> **Note :** les règles avec des noms de serveur ne doivent **pas** comprendre le sous-dossier du script d'entrée dans leur motif. Par exemple, si l'application est sous `https://www.example.com/sandbox/blog`, alors vous devez utiliser le motif `https://www.example.com/posts` au lieu de `https://www.example.com/sandbox/blog/posts`. Cela permet à votre application d'être déployée sous n'importe quel dossier sans avoir à changer son code. Yii détecte automatiquement l'URL de base de l'application.

Suffixes d'URL

Vous désirez peut-être ajouter des suffixes aux URL pour des raisons variées. Par exemple, vous pouvez ajouter `.html` aux URL de manière à ce qu'elles ressemblent à des URL de pages HTML statiques. Vous pouvez aussi y ajouter `.json` pour indiquer le type de contenu attendu pour la réponse. Vous pouvez faire cela en configurant la propriété `yii\web\UrlManager::$suffix` dans la configuration de l'application comme ceci :

```
[
    // ...
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            // ...
            'suffix' => '.html',
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

La configuration ci-dessus permet au `gestionnaire d'URL` de reconnaître les URL requises et aussi de créer des URL avec le suffixe `.html`.

> **Conseil :** vous pouvez définir `/` en tant que suffixe des URL de manière à ce que tous les URL se terminent par la barre oblique de division.
>
> **Note :** lorsque vous configurez un suffixe d'URL, si une URL requise ne contient pas ce suffixe, elle est considérée comme une URL non reconnue. Cela est une pratique recommandée pour l'optimisation des moteurs de recherche (SE0 – Search Engine Optimization).

Parfois vous désirez utiliser des suffixes différents pour différentes URL. Cela peut être fait en configurant la propriété `suffix` des règles d'URL individuelles. Lorsqu'une URL a cette propriété définie, elle écrase la valeur définie au niveau du `gestionnaire d'URL`. Par exemple, la configuration suivante contient une règle d'URL personnalisée qui utilise `.json` en tant que suffixe à la place du suffixe défini globalement `.html`.

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            // ...
            'suffix' => '.html',
            'rules' => [
                // ...
                [
                    'pattern' => 'posts',
                    'route' => 'post/index',
                    'suffix' => '.json',
                ],
            ],
        ],
    ],
]
```

Méthodes HTTP

En mettant en œuvre des API pleinement REST, il est couramment nécessaire que la même URL puisse être résolue en différentes routes selon la méthode HTTP utilisée par la requête. Cela peut être fait facilement en préfixant les motifs des règles avec les méthodes HTTP prises en charge. Si une règle prend en charge plusieurs méthodes HTTP, il faut séparer les noms de méthode par une virgule. Par exemple, les règles suivantes ont le même motif `post/<id:\d+>` mais des méthodes HTTP différentes. Une requête de `PUT post/100` est résolue en la route `post/update`, tandis que la requête de `GET post/100` en la route `post/view`.

```
'rules' => [
    'PUT,POST post/<id:\d+>' => 'post/update',
```

```
    'DELETE post/<id:\d+>' => 'post/delete',
    'post/<id:\d+>' => 'post/view',
]
```

> **Note :** si une règle d'URL contient des méthodes HTTP dans son motif, la règle n'est utilisée qu'à des fins d'analyse sauf si `GET` fait partie des verbes spécifiés. Elle est ignorée quand le `gestionnaire d'URL` est sollicité pour créer une URL.
>
> **Conseil :** pour simplifier le routage des API pleinement REST, Yii fournit la classe spéciale de règle d'URL `yii\rest\UrlRule` qui est très efficace et prend en charge quelques fonctionnalités originales comme la pluralisation automatique des identifiants de contrôleur. Pour plus de détails, reportez-vous à la section Routage dans le chapitre API pleinement REST.

Ajout dynamique de règles

Des règles d'URL peuvent être ajoutées dynamiquement au `gestionnaire d'URL`. Cela est souvent nécessaire pour les modules distribuables qui veulent gérer leurs propres règles d'URL. Pour que les règles ajoutées dynamiquement prennent effet dans de processus de routage, vous devez les ajouter dans l'étape d'amorçage. Pour les modules, cela signifie qu'ils doivent implémenter l'interface `yii\base\BootstrapInterface` et ajouter les règles dans leur méthode `bootstrap()` comme l'exemple suivant le montre :

```
public function bootstrap($app)
{
    $app->getUrlManager()->addRules([
        // rule declarations here
    ], false);
}
```

Notez que vous devez également lister ces modules dans la propriété `yii\web\Application::bootstrap()` afin qu'ils puissent participer au processus d'amorçage.

Création des classes règles

En dépit du fait que la classe par défaut `yii\web\UrlRule` est suffisamment flexible pour la majorité des projets, il y a des situations dans lesquelles vous devez créer votre propres classes de règle. Par exemple, dans un site Web de vendeur de voitures, vous désirerez peut-être prendre en charge des formats d'URL du type `/Manufacturer/Model`, où `Manufacturer` et `Model` doivent correspondre à quelques données stockées dans une base de données. La classe de règle par défaut ne fonctionne pas dans ce cas car elle s'appuie sur des motifs déclarés de manière statique.

Vous pouvez créer les classes de règle d'URL suivantes pour résoudre ce problème :

```
<?php

namespace app\components;

use yii\web\UrlRuleInterface;
use yii\base\BaseObject;

class CarUrlRule extends BaseObject implements UrlRuleInterface
{
    public function createUrl($manager, $route, $params)
    {
        if ($route === 'car/index') {
            if (isset($params['manufacturer'], $params['model'])) {
                return $params['manufacturer'] . '/' . $params['model'];
            } elseif (isset($params['manufacturer'])) {
                return $params['manufacturer'];
            }
        }
        return false;  // this rule does not apply
    }

    public function parseRequest($manager, $request)
    {
        $pathInfo = $request->getPathInfo();
        if (preg_match('%^(\w+)(/(\w+))?$%', $pathInfo, $matches)) {
            // vérifie  $matches[1] et $matches[3] pour voir si
            // elles correspondent à un  manufacturer et à un model dans la
            base de données
            // si oui, définit $params['manufacturer'] et/ou
            $params['model']
            // et retourne  ['car/index', $params]
        }
        return false;  // cette règle ne s'applique pas
    }
}
```

Et utilisez la nouvelle classe de règle dans la configuration de `yii\web\UrlManager::$rules` :

```
[
    // ...other rules...

    [
        'class' => 'app\components\CarUrlRule',
        // ...configure d'autres propriétés...
    ],
]
```

4.2.5 Normalisation d'URL

Depuis la version 2.0.10, le `gestionnaire d'URL` peut être configuré pour utiliser le `normalisateur d'URL` pour prendre en compte les variations de la même URL, p. ex. avec et sans la barre oblique de division de fin.

Parce que, techniquement, `https://example.com/path` et `https://example.com/path/` sont des URL différentes, servir le même contenu pour chacune d'elles peut dégrader le classement SEO. Par défaut, le normalisateur fusionne les barres obliques de division consécutives, ajoute ou retire des barres de division de fin selon que le suffixe comporte une barre de division de fin ou pas, et redirige vers la version normalisée de l'URL en utilisant la redirection permanente[3]. Le normalisateur peut être configuré globalement pour le gestionnaire d'URL ou individuellement pour chacune des règles — par défaut, chacune des règles utilise le normalisateur du gestionnaire d'URL. Vous pouvez définir `UrlRule::$normalizer` à `false` pour désactiver la normalisation pour une règle d'URL particulière.

Ce qui suit est un exemple de configuration pour le `normalisateur d'URL` :

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'showScriptName' => false,
    'enableStrictParsing' => true,
    'suffix' => '.html',
    'normalizer' => [
        'class' => 'yii\web\UrlNormalizer',
        // utilise la redirection temporaire au lieu de la redirection
        permanente pour le débogage
        'action' => UrlNormalizer::ACTION_REDIRECT_TEMPORARY,
    ],
    'rules' => [
        // ...autres règles...
        [
            'pattern' => 'posts',
            'route' => 'post/index',
            'suffix' => '/',
            'normalizer' => false, // désactive le normalisateur pour cette
            règle
        ],
        [
            'pattern' => 'tags',
            'route' => 'tag/index',
            'normalizer' => [
                // ne fusionne pas les barres obliques de division
                consécutives pour cette règle
                'collapseSlashes' => false,
            ],
        ],
```

3. https://fr.wikipedia.org/wiki/HTTP_301

```
    ],
]
```

> **Note :** par défaut `UrlManager::$normalizer` est désactivé. Vous devez le configure explicitement pour activer la normalisation d'URL.

4.2.6 Considérations de performance

Lors du développement d'une application Web complexe, il est important d'optimiser les règles d'URL afin que l'analyse des requêtes et la création d'URL prennent moins de temps.

En utilisant les routes paramétrées, vous pouvez réduire le nombre de règles d'URL, ce qui accroît significativement la performance.

Lors de l'analyse d'URL ou de la création d'URL, le `gestionnaire d'URL` examine les règles d'URL dans l'ordre de leur déclaration. En conséquence, vous devez envisager d'ajuster cet ordre afin que les règles les plus spécifiques et/ou utilisées couramment soient placées avant les règles les moins utilisées.

Si quelques règles d'URL partagent le même préfixe dans leur motif ou dans leur route, vous pouvez envisager d'utiliser `yii\web\GroupUrlRule` pour qu'elles puissent être examinées plus efficacement par le `gestionnaire d'URL` en tant que groupe. Cela est souvent le cas quand votre application est composée de modules, chacun ayant son propre jeu de règles d'URL avec l'identifiant de module comme préfixe commun.

4.3 Requêtes

Les requêtes faites à l'application sont représentées en terme d'objets `yii\web\Request` qui fournissent des informations telles que les paramètres de requête, les entêtes HTTP, les cookies, etc. Pour une requête donnée, vous avez accès au composant d'application`request` qui, par défaut, est une instance de `yii\web\Request`. Dans cette section, nous décrivons comment utiliser ce composant dans vos applications.

4.3.1 Paramètres de requête

Pour obtenir les paramètres de requête, vous pouvez appeler les méthodes `get()` et `post()` du composant `request` component. Elles retournent les valeurs de `$_GET` et `$_POST`, respectivement. Pas exemple :

```
$request = Yii::$app->request;

$get = $request->get();
// équivalent à : $get = $_GET;

$id = $request->get('id');
```

```php
// équivalent à : $id = isset($_GET['id']) ? $_GET['id'] : null;

$id = $request->get('id', 1);
// équivalent à : $id = isset($_GET['id']) ? $_GET['id'] : 1;

$post = $request->post();
// équivalent à : $post = $_POST;

$name = $request->post('name');
// equivalent to: $name = isset($_POST['name']) ? $_POST['name'] : null;

$name = $request->post('name', '');
// équivalent à : $name = isset($_POST['name']) ? $_POST['name'] : '';
```

> **Info :** plutôt que d'accéder directement à `$_GET` et `$_POST` pour récupérer les paramètres de requête, il est recommandé de les obtenir via le composant `request` comme indiqué ci-dessus. Cela rend l'écriture des tests plus facile parce que vous pouvez créer un simulacre de composant 'request' avec des données de requête factices.

Lorsque vous mettez en œuvre des API pleinement REST, vous avez souvent besoin de récupérer les paramètres qui sont soumis via les méthodes de requête PUT, PATCH ou autre . Vous pouvez obtenir ces paramètres en appelant la méthode `yii\web\Request::getBodyParam()`. par exemple :

```php
$request = Yii::$app->request;

// retourne tous les paramètres
$params = $request->bodyParams;

// retourne le paramètre  "id"
$param = $request->getBodyParam('id');
```

> **Info :** à la différence des paramètres de `GET`, les paramètres soumis via `POST`, `PUT`, `PATCH` etc. sont envoyés dans le corps de la requête. Le composant `request` analyse ces paramètres lorsque vous y accédez via les méthodes décrites ci-dessus. Vous pouvez personnaliser la manière dont ces paramètres sont analysés en configurant la propriété `yii\web\Request::$parsers`.

4.3.2 Méthodes de requête

Vous pouvez obtenir la méthode HTTP utilisée par la requête courante via l'expression `Yii::$app->request->method`. Un jeu entier de propriétés booléennes est également fourni pour que vous puissiez déterminer le type de la méthode courante. Par exemple :

```php
$request = Yii::$app->request;
```

```php
if ($request->isAjax) { /* la méthode de requête est requête AJAX */ }
if ($request->isGet)  { /* la méthode de requête est requête GET */ }
if ($request->isPost) { /* la méthode de requête est requête POST */ }
if ($request->isPut)  { /* la méthode de requête est requête PUT */ }
```

4.3.3 URL de requête

Le composant `request` fournit plusieurs manières d'inspecter l'URL couramment requise.

En supposant que l'URL requise soit `https://example.com/admin/index.php/product?id=100`, vous pouvez obtenir différentes parties de cette URL comme c'est résumé ci-dessous :

- `url` : retourne`/admin/index.php/product?id=100`, qui est l'URL sans la partie hôte.
- `absoluteUrl` : retourne `https://example.com/admin/index.php/product?id=100`, qui est l'URL complète y compris la partie hôte.
- `hostInfo` : retourne `https://example.com`, qui est la partie hôte de l'URL.
- `pathInfo` : retourne `/product`, qui est la partie après le script d'entrée et avant le point d'interrogation (chaîne de requête).
- `queryString` : retourne `id=100`, qui est la partie après le point d'interrogation.
- `baseUrl` : retourne `/admin`, qui est la partie après l'hôte et avant le nom du script d'entrée.
- `scriptUrl` : retourne `/admin/index.php`, qui set l'URL sans le chemin et la chaîne de requête.
- `serverName` : retourne `example.com`, qui est le nom d'hôte dans l'URL.
- `serverPort` : retourne 80, qui est le numéro de port utilisé par le serveur Web.

4.3.4 Entêtes HTTP

Vous pouvez obtenir les entêtes HTTP via la `collection d'entêtes` qui est retournée par la propriété `yii\web\Request::$headers`. Par exemple :

```php
// $headers est un objet   yii\web\HeaderCollection
$headers = Yii::$app->request->headers;

// retourne la valeur de l'entête  Accept
$accept = $headers->get('Accept');

if ($headers->has('User-Agent')) { /* il existe un entête User-Agent  */ }
```

Le composant `request` fournit aussi la prise en charge de l'accès rapide à quelques entêtes couramment utilisés. Cela inclut :

- `userAgent` : retourne la valeur de l'entête `User-Agent`.
- `contentType` : retourne la valeur de l'entête `Content-Type` qui indique le type MIME des données dans le corps de la requête.

— `acceptableContentTypes` : retourne les types MIME acceptés par l'utilisateur. Les types retournés sont classés par ordre de score de qualité. Les types avec les plus hauts scores sont retournés en premier.
— `acceptableLanguages` : retourne les langues acceptées par l'utilisateur. Les langues retournées sont classées par niveau de préférence. Le premier élément représente la langue préférée. Si votre application prend en charge plusieurs langues et que vous voulez afficher des pages dans la langue préférée de l'utilisateur, vous pouvez utiliser la méthode de négociation de la langue `yii\web\Request::getPreferredLanguage()`. Cette méthode accepte une liste des langues prises en charge par votre application, la compare avec les [[yii\web\Request : :acceptableLanguages (langues acceptées)|acceptableLanguages]], et retourne la langue la plus appropriée.

Conseil : vous pouvez également utiliser le filtre `ContentNegotiator` pour déterminer dynamiquement quel type de contenu et quelle langue utiliser dans la réponse. Le filtre met en œuvre la négociation de contenu en plus des propriétés et méthodes décrites ci-dessus.

4.3.5 Informations sur le client

Vous pouvez obtenir le nom d'hôte et l'adresse IP de la machine cliente via `userHost` et `userIP`, respectivement. Par exemple :

```
$userHost = Yii::$app->request->userHost;
$userIP = Yii::$app->request->userIP;

## Mandataires de confiance et entêtes <span id="trusted-proxies"></span>

Dans la section précédente, vous avez vu comment obtenir des informations
sur l'utilisateur comme le nom d'hôte et l'adresse IP.
Cela fonctionne sans aucune configuration complémentaire dans une
configuration normale dans laquelle une unique serveur Web est utilisé pour
servir le site.
Cependant, si votre application s'exécute derrière un mandataire inverse,
vous devez compléter la configuration pour retrouver ces informations car le
client direct est désormais le mandataire
et l'adresse IP de l'utilisateur est passée à l'application Yii par une
entête établie par le mandataire.


Vous ne devez pas faire confiance aveuglément aux entêtes fournies par un
mandataire sauf si vous faites explicitement confiance à ce mandataire.
Depuis sa version 2.0.13,  Yii prend en charge la configuration des
mandataires de confiance via les propriétés
[[yii\web\Request::trustedHosts|trustedHosts]],
[[yii\web\Request::secureHeaders|secureHeaders]],
[[yii\web\Request::ipHeaders|ipHeaders]] and
[[yii\web\Request::secureProtocolHeaders|secureProtocolHeaders]]
du composant `request`.
```

```
Ce qui suit est la configuration d'une requête pour une application qui
s'exécute derrière une tableau de mandataires inverses
situés dans le réseau IP `10.0.2.0/24` IP network:

```php
'request' => [
    // ...
    'trustedHosts' => [
        '10.0.2.0/24',
    ],
],
```

L'adresse IP est envoyée par défaut par le mandataire dans l'entête `X-Forwarded-For` , et le protocole (`http` ou `https`) est envoyé dans `X-Forwarded-Proto`.

Dans le cas où vos mandataires utilisent différentes entêtes, vous pouvez utiliser la configuration de la requête pour les ajuster, p. ex. :

```php
'request' => [
    // ...
    'trustedHosts' => [
        '10.0.2.0/24' => [
            'X-ProxyUser-Ip',
            'Front-End-Https',
        ],
    ],
    'secureHeaders' => [
        'X-Forwarded-For',
        'X-Forwarded-Host',
        'X-Forwarded-Proto',
        'X-Proxy-User-Ip',
        'Front-End-Https',
    ],
    'ipHeaders' => [
        'X-Proxy-User-Ip',
    ],
    'secureProtocolHeaders' => [
        'Front-End-Https' => ['on']
    ],
],
```

Avec la configuration précédente, toutes les entêtes listées dans `secureHeaders` sont filtrées de la requête à l'exception des entêtes `X-ProxyUser-Ip` et `Front-End-Https` pour le cas où la requête est élaborée par le mandataire. Dans un tel cas, le précédent est utilisé pour retrouver l'adresse IP de l'utilisateur comme configuré dans `ipHeaders` et le dernier est utilisé pour déterminer le résultat de `yii\web\Request::getIsSecureConnection()`.
```

4.4 Réponses

Quand une application a terminé la prise en charge d'une requête, elle génère un objet `response` et l'envoie à l'utilisateur final. L'objet `response` contient des informations telles que le code d'état HTTP, les entêtes HTTP et le corps. Le but ultime du développement d'applications Web est essentiellement du construire de tels objets `response` pour des requêtes variées.

Dans la plupart des cas, vous devez travailler avec le composant d'application `response` qui, par défaut, est une instance de `yii\web\Response`. Néanmoins, Yii vous permet également de créer vos propres objets `response` et de les envoyer à l'utilisateur final comme nous l'expliquons dans ce qui suit.

Dans cette section, nous décrivons comment composer et envoyer des réponses à l'utilisateur final.

4.4.1 Code d'état

Une des premières choses que vous devez faire lorsque vous construisez une réponse est de déclarer si la requête a été correctement prise en charge ou pas. Cela se fait en définissant la propriété
`code d'état` qui peut prendre un des codes d'état HTTP [4] valides. Par exemple, pour indiquer que la requête a été prise en charge avec succès, vous pouvez définir le code à 200, comme ceci :

```
Yii::$app->response->statusCode = 200;
```

Néanmoins, dans la plupart des cas, vous n'avez pas besoin de définir ce code explicitement. Cela tient au fait que la valeur par défaut de `yii\web\Response::$statusCode` est 200. Et, si vous voulez indiquer que la prise en charge de la requête a échoué vous pouvez lever une exception appropriée comme ceci :

```
throw new \yii\web\NotFoundHttpException;
```

Lorsque le gestionnaire d'erreurs intercepte l'exception, il extraie le code d'état de l'exception et l'assigne à la réponse. Concernant l'exception `yii\web\NotFoundHttpException` ci-dessus, elle est associée au code d'état HTTP 404. Les exception HTTP suivantes sont prédéfinies dans Yii :

- `yii\web\BadRequestHttpException` : code d'état 400.
- `yii\web\ConflictHttpException` : code d'état 409.
- `yii\web\ForbiddenHttpException` : code d'état 403.
- `yii\web\GoneHttpException` : code d'état 410.
- `yii\web\MethodNotAllowedHttpException` : code d'état 405.
- `yii\web\NotAcceptableHttpException` : code d'état 406.

4. https://tools.ietf.org/html/rfc2616#section-10

— `yii\web\NotFoundHttpException` : code d'état 404.
— `yii\web\ServerErrorHttpException` : code d'état 500.
— `yii\web\TooManyRequestsHttpException` : code d'état 429.
— `yii\web\UnauthorizedHttpException` : code d'état 401.
— `yii\web\UnsupportedMediaTypeHttpException` : code d'état 415.

Si l'exception que vous voulez lever ne fait pas partie de cette liste, vous pouvez en créer une en étendant la classe `yii\web\HttpException`, ou en en levant une à laquelle vous passez directement le code d'état. Par exemple :

```
throw new \yii\web\HttpException(402);
```

4.4.2 Entêtes HTTP

Vous pouvez envoyer les entêtes HTTP en manipulant la `collection d'entêtes` dans le composant `response`. Par exemple :

```
$headers = Yii::$app->response->headers;

// ajoute un entête  Pragma . L'entête Pragma existant n'est PAS écrasé.
$headers->add('Pragma', 'no-cache');

// définit un entête Pragma. Tout entête Pragma existant est supprimé.
$headers->set('Pragma', 'no-cache');

// retire un (des) entêtes Pragma et retourne les valeurs de l'entête Pragma
retiré dans un tableau
$values = $headers->remove('Pragma');
```

> **Info :** les noms d'entête ne sont pas sensibles à la casse. Les nouveaux entêtes enregistrés ne sont pas envoyés à l'utilisateur tant que la méthode `yii\web\Response::send()` n'est pas appelée.

4.4.3 Corps de la réponse

La plupart des réponses doivent avoir un corps qui transporte le contenu que vous voulez montrer à l'utilisateur final.

Si vous disposez déjà d'une chaîne de caractères formatée pour le corps, vous pouvez l'assigner à la propriété `yii\web\Response::$content` de la réponse. Par exemple :

```
Yii::$app->response->content = 'hello world!';
```

Si vos données doivent être formatées avant l'envoi à l'utilisateur final, vous devez définir les propriétés `format` et `data`. La propriété `format` spécifie dans quel format les `données` doivent être formatées. Par exemple :

```
$response = Yii::$app->response;
$response->format = \yii\web\Response::FORMAT_JSON;
$response->data = ['message' => 'hello world'];
```

De base, Yii prend en charge les formats suivants, chacun mis en œuvre par une classe `formatter`. Vous pouvez personnaliser les formateurs ou en ajouter de nouveaux en configurant la propriété `yii\web\Response::$formatters`.

- `HTML` : mise en œuvre par `yii\web\HtmlResponseFormatter`.
- `XML` : mise en œuvre par `yii\web\XmlResponseFormatter`.
- `JSON` : mise en œuvre par `yii\web\JsonResponseFormatter`.
- `JSONP` : mise en œuvre par `yii\web\JsonResponseFormatter`.
- `RAW` : utilisez ce format si vous voulez envoyer la réponse directement sans lui appliquer aucun formatage.

Bien que le corps de la réponse puisse être défini explicitement comme montré ci-dessus, dans la plupart des cas, vous pouvez le définir implicitement en utilisant la valeur retournée par les méthodes d'action. Un cas d'usage courant ressemble à ceci :

```
public function actionIndex()
{
    return $this->render('index');
}
```

L'action `index` ci-dessus retourne le résultat du rendu de la vue `index`. La valeur de retour est interceptée par le composant `response`, formatée et envoyée à l'utilisateur final.

Parce que le format par défaut de la réponse est `HTML`, vous devez seulement retourner un chaîne de caractères dans une méthode d'action. Si vous utilisez un format de réponse différent, vous devez le définir avant de retourner les donnés. Par exemple :

```
public function actionInfo()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
    return [
        'message' => 'hello world',
        'code' => 100,
    ];
}
```

Comme mentionné plus haut, en plus d'utiliser le composant d'application `response`, vous pouvez également créer vos propres objets `response` et les envoyer à l'utilisateur final. Vous pouvez faire cela en retournant un tel objet dans une méthode d'action, comme le montre l'exemple suivant :

```
public function actionInfo()
{
    return \Yii::createObject([
        'class' => 'yii\web\Response',
        'format' => \yii\web\Response::FORMAT_JSON,
        'data' => [
            'message' => 'hello world',
```

```
            'code' => 100,
        ],
    ]);
}
```

> Note : si vous êtes en train de créer vos propres objets `response`, vous ne pourrez par bénéficier des configurations que vous avez établies pour le composant `response` dans la configuration de l'application. Vous pouvez néanmoins, utiliser l'injection de dépendances pour appliquer une configuration commune à vos nouveaux objets `response`.

4.4.4 Redirection du navigateur

La redirection du navigateur s'appuie sur l'envoi d'un entête HTTP `Location`. Comme cette fonctionnalité est couramment utilisée, Yii fournit une prise en charge spéciale pour cela.

Vous pouvez rediriger le navigateur sur une URL en appelant la méthode `yii\web\Response::redirect()`. Cette méthode définit l'entête `Location` approprié avec l'URL donnée et retourne l'objet `response` lui-même. Dans une méthode d'action vous pouvez appeler sa version abrégée `yii\web\Controller::redirect()`. Par exemple :

```
public function actionOld()
{
    return $this->redirect('https://example.com/new', 301);
}
```

Dans le code précédent, la méthode d'action retourne le résultat de la méthode `redirect()`. Comme expliqué ci-dessus, l'objet `response` retourné par une méthode d'action est utilisé en tant que réponse à envoyer à l'utilisateur final.

Dans des endroits autres que les méthodes d'action, vous devez appeler la méthode `yii\web\Response::redirect()` directement, suivi d'un appel chaîné à la méthode `yii\web\Response::send()` pour garantir qu'aucun contenu supplémentaire ne sera ajouté à la réponse.

```
\Yii::$app->response->redirect('https://example.com/new', 301)->send();
```

> **Info :** par défaut la méthode `yii\web\Response::redirect()` définit le code d'état à 302 pour indiquer au navigateur que la ressource requise est *temporairement* située sous un URI différent. Vous pouvez passer un code 301 pour dire au navigateur que la ressource a été déplacée *de manière permanente*.

Lorsque la requête courante est une requête AJAX, l'envoi d'un entête `Location` ne provoque pas automatiquement une redirection du navigateur. Pour pallier ce problème, la méthode `yii\web\Response::redirect()` définit un

entête `X-Redirect` avec l'URL de redirection comme valeur. Du côté client, vous pouvez écrire un code JavaScript pour lire l'entête et rediriger le navigateur sur l'URL transmise.

> **Info :** Yii est fourni avec un fichier JavaScript `yii.js` qui fournit un jeu d'utilitaires JavaScript, y compris l'utilitaire de redirection basé sur l'entête `X-Redirect`. Par conséquent, si vous utilisez ce fichier JavaScript (en enregistrant le paquet de ressources `yii\web\YiiAsset`), vous n'avez rien à écrire pour prendre en charge la redirection AJAX. De l'information complémentaire sur `yii.js` est disponible à la section Scripts client.

4.4.5 Envoi de fichiers

Comme la redirection du navigateur, l'envoi de fichiers est une autre fonctionnalité qui s'appuie sur les entêtes HTTP spécifiques. Yii fournit un jeu de méthodes pour prendre en charge différents besoins d'envoi de fichiers. Elles assurent toutes la prise en charge de la plage d'entêtes HTTP.

- `yii\web\Response::sendFile()` : envoie un fichier existant à un client.
- `yii\web\Response::sendContentAsFile()` : envoie un chaîne de caractères en tant que fichier à un client.
- `yii\web\Response::sendStreamAsFile()` : envoie un flux de fichier existant en tant que fichier à un client.

Ces méthodes ont la même signature avec l'objet `response` comme valeur de retour. Si le fichier à envoyer est trop gros, vous devez envisager d'utiliser `yii\web\Response::sendStreamAsFile()` parce qu'elle fait un usage plus efficace de la mémoire. L'exemple qui suit montre comment envoyer un fichier dans une action de contrôleur.

```
public function actionDownload()
{
    return \Yii::$app->response->sendFile('path/to/file.txt');
}
```

Si vous appelez la méthode d'envoi de fichiers dans des endroits autres qu'une méthode d'action, vous devez aussi appeler la méthode `yii\web\Response::send()` immédiatement après pour garantir qu'aucun contenu supplémentaire ne sera ajouté à la réponse.

```
\Yii::$app->response->sendFile('path/to/file.txt')->send();
```

Quelques serveurs Web assurent une prise en charge spéciale de l'envoi de fichiers appelée *X-Sendfile*. L'idée est de rediriger la requête d'un fichier sur le serveur Web qui sert directement le fichier. En conséquence, l'application Web peut terminer plus rapidement tandis que le serveur Web est en train d'envoyer le fichier. Pour utiliser cette fonctionnalité, vous pouvez appeler la méthode `yii\web\Response::xSendFile()`. La liste suivante résume,

comment activer la fonctionnalité `X-Sendfile` pour quelques serveurs Web populaires :

— Apache : X-Sendfile [5]
— Lighttpd v1.4 : X-LIGHTTPD-send-file [6]
— Lighttpd v1.5 : X-Sendfile [7]
— Nginx : X-Accel-Redirect [8]
— Cherokee : X-Sendfile and X-Accel-Redirect [9]

4.4.6 Envoi de la réponse

Le contenu d'une réponse n'est pas envoyé à l'utilisateur tant que la méthode `yii\web\Response::send()` n'est pas appelée. Par défaut, cette méthode est appelée automatiquement à la fin de `yii\base\Application::run()`. Vous pouvez néanmoins appeler cette méthode explicitement pour forcer l'envoi de la réponse immédiatement.

La méthode `yii\web\Response::send()` entreprend les étapes suivantes pour envoyer la réponse :

1. Elle déclenche l'événement `yii\web\Response::EVENT_BEFORE_SEND`.
2. Elle appelle `yii\web\Response::prepare()` pour formater `les données de la réponse` du `contenu de la réponse`.
3. Elle déclenche l'événement `yii\web\Response::EVENT_AFTER_PREPARE`.
4. Elle appelle la méthode `yii\web\Response::sendHeaders()` pour envoyer les entêtes HTTP enregistrés.
5. Elle appelle la méthode `yii\web\Response::sendContent()` pour envoyer le corps de la réponse.
6. Elle déclenche l'événement `yii\web\Response::EVENT_AFTER_SEND`.

Après que la méthode `yii\web\Response::send()` est appelée une fois, tout appel suivant de cette méthode est ignoré. Cela signifie qu'une fois la réponse expédiée, vous ne pouvez lui ajouter aucun contenu.

Comme vous pouvez le voir, la méthode `yii\web\Response::send()` déclenche plusieurs événements utiles. En répondant à ces événements, il est possible d'ajuster ou d'enjoliver la réponse.

---

5. https://tn123.org/mod_xsendfile
6. https://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
7. https://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
8. https://www.nginx.com/resources/wiki/start/topics/examples/xsendfile/
9. https://www.cherokee-project.com/doc/other_goodies.html#x-sendfile

4.5 Sessions et témoins de connexion

Les sessions et les témoins de connexion permettent à des données d'être conservées à travers des requêtes multiples. Avec le langage PHP simple, vous pouvez y accéder via les variables globales `$_SESSION` et `$_COOKIE`, respectivement. Yii encapsule les sessions et les témoins de connexion sous forme d'objets et, par conséquent, vous permet d'y accéder d'une manière orientée objet avec des améliorations utiles.

4.5.1 Sessions

Comme pour les requêtes et les réponses, vous pouvez accéder aux sessions via le composant d'application `session` qui, par défaut, est une instance de la classe `yii\web\Session`.

Ouverture et fermeture d'une session

Pour ouvrir et fermer une session, vous pouvez procéder comme suit :

```
$session = Yii::$app->session;

// vérifie si une session est déjà ouverte
if ($session->isActive) ...

// ouvre une session
$session->open();

// ferme une session
$session->close();

// détruit toutes les données enregistrées dans une session.
$session->destroy();
```

Vous pouvez appeler les méthodes `open()` et `close()` plusieurs fois sans causer d'erreur ; en interne les méthodes commencent par vérifier si la session n'est pas déjà ouverte.

Accès aux données de session

Pour accéder aux données stockées dans une session, vous pouvez procéder comme indiqué ci-après :

```
$session = Yii::$app->session;

// obtient une variable de session. Les utilisations suivantes sont
équivalentes :
$language = $session->get('language');
$language = $session['language'];
$language = isset($_SESSION['language']) ? $_SESSION['language'] : null;
```

```php
// définit une variable de session variable. Les utilisations suivantes sont
équivalentes :
$session->set('language', 'en-US');
$session['language'] = 'en-US';
$_SESSION['language'] = 'en-US';

// supprime une variable session. Les utilisations suivantes sont
équivalentes :
$session->remove('language');
unset($session['language']);
unset($_SESSION['language']);

// vérifie si une session possède la variable 'language'. Les utilisations
suivantes sont équivalentes :
if ($session->has('language')) ...
if (isset($session['language'])) ...
if (isset($_SESSION['language'])) ...

// boucle sur toutes les sessions. Les utilisations suivantes sont
équivalentes :
foreach ($session as $name => $value) ...
foreach ($_SESSION as $name => $value) ...
```

> **Info :** lorsque vous accédez aux données d'une session via le composant `session`, une session est automatiquement ouverte si elle ne l'a pas déjà été. Cela est différent de l'accès aux données via la variable globale `$_SESSION`, qui réclame un appel préalable explicite de `session_start()`.

Lorsque vous travaillez avec les données de session qui sont des tableaux, le composant `session` possède une limitation qui vous empêche de modifier directement un des élément de ces tableaux. Par exemple :

```php
$session = Yii::$app->session;

// le code suivant ne fonctionne PAS
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// le code suivant fonctionne :
$session['captcha'] = [
    'number' => 5,
    'lifetime' => 3600,
];

// le code suivant fonctionne également :
echo $session['captcha']['lifetime'];
```

Vous pouvez utiliser une des solutions de contournement suivantes pour résoudre ce problème :

```php
$session = Yii::$app->session;

// utiliser directement  $_SESSION (assurez-vous que
Yii::$app->session->open() a été appelée)
$_SESSION['captcha']['number'] = 5;
$_SESSION['captcha']['lifetime'] = 3600;

// obtenir le tableau complet d'abord, le modifier et le sauvegarder
$captcha = $session['captcha'];
$captcha['number'] = 5;
$captcha['lifetime'] = 3600;
$session['captcha'] = $captcha;

// utiliser un  ArrayObject au lieu d'un tableau
$session['captcha'] = new \ArrayObject;
...
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// stocker les données du tableau par une clé avec un préfixe commun
$session['captcha.number'] = 5;
$session['captcha.lifetime'] = 3600;
```

Pour une meilleure performance et une meilleure lisibilité du code, nous recommandons la dernière solution de contournement. Elle consiste, au lieu de stocker un tableau comme une donnée de session unique, à stocker chacun des éléments du tableau comme une variable de session qui partage le même préfixe de clé avec le reste des éléments de ce tableau.

Stockage de session personnalisé

La classe par défaut `yii\web\Session` stocke les données de session sous forme de fichiers sur le serveur. Yii fournit également des classes de session qui mettent en œuvre des procédés de stockage différents. En voici la liste :

- `yii\web\DbSession` : stocke les données de session dans une base de données.
- `yii\web\CacheSession` : stocke les données de session dans un cache avec l'aide d'un composant cache configuré.
- `yii\redis\Session` : stocke les données de session en utilisant le médium de stockage redis [10] as the storage medium.
- `yii\mongodb\Session` : stocke les données de session dans une base de données de documents MongoDB [11].

Toutes ces classes de session prennent en charge le même jeu de méthodes d'API. En conséquence, vous pouvez changer de support de stockage sans avoir à modifier le code de votre application qui utilise ces sessions.

10. https://redis.io/
11. https://www.mongodb.com/

> **Note :** si vous voulez accéder aux données de session via `$_SESSION` quand vous êtes en train d'utiliser une session à stockage personnalisé, vous devez vous assurer que cette session a été préalablement démarrée via `yii\web\Session::open()`. Cela est dû au fait que les gestionnaires de stockage des sessions personnalisées sont enregistrés à l'intérieur de cette méthode.
>
> Note : si vous utilisez un stockage de session personnalisé, vous devez configurer le collecteur de déchets de session explicitement. Quelques installations de PHP (p. ex. Debian) utilisent une probabilité de collecteur de déchets de 0 et nettoient les fichiers de session hors ligne dans une tâche de cron. Ce processus ne s'applique pas à votre stockage personnalisé, c'est pourquoi vous devez configurer `yii\web\Session::$GCProbability` pour utiliser une valeur non nulle.

Pour savoir comment configurer et utiliser ces classes de composant, reportez-vous à leur documentation d'API. Ci-dessous, nous présentons un exemple de configuration de `yii\web\DbSession` dans la configuration de l'application pour utiliser une base de données en tant que support de stockage d'une session :

```
return [
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',
            // 'db' => 'mydb',  // l'identifiant du composant d'application
            de la connexion à la base de données. Valeur par défaut : 'db'.
            // 'sessionTable' => 'my_session', // nom de la table 'session'
            . Valeur par défaut : 'session'.
        ],
    ],
];
```

Vous devez aussi créer la base de données suivante pour stocker les données de session :

```
CREATE TABLE session
(
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

où 'BLOB' fait référence au type « grand objet binaire » (binary large objet — BLOB) de votre système de gestion de base de données (DBMS) préféré. Ci-dessous, vous trouverez les types de BLOB qui peuvent être utilisés par quelques DBMS populaires :

— MySQL : LONGBLOB
— PostgreSQL : BYTEA
— MSSQL : BLOB

Note : en fonction des réglages de `session.hash_function` dans votre fichier php.ini, vous devez peut-être ajuster la longueur de la colonne `id`. Par exemple, si `session.hash_function=sha256`, vous devez utiliser une longueur de 64 au lieu de 40.

Cela peut être accompli d'une façon alternative avec la migration suivante :

```
<?php

use yii\db\Migration;

class m170529_050554_create_table_session extends Migration
{
    public function up()
    {
        $this->createTable('{{%session}}', [
            'id' => $this->char(64)->notNull(),
            'expire' => $this->integer(),
            'data' => $this->binary()
        ]);
        $this->addPrimaryKey('pk-id', '{{%session}}', 'id');
    }

    public function down()
    {
        $this->dropTable('{{%session}}');
    }
}
```

Donnés flash

Les données flash sont une sorte de données de session spéciale qui, une fois définies dans une requête, ne restent disponibles que durant la requête suivante et sont détruites automatiquement ensuite. Les données flash sont le plus communément utilisées pour mettre en œuvre des messages qui doivent être présentés une seule fois, comme les messages de confirmation affichés après une soumission réussie de formulaire.

Vous pouvez définir des données flash et y accéder via le composant d'application `session`. Par exemple :

```
$session = Yii::$app->session;

// Request #1
// définit un message flash nommé "commentDeleted"
$session->setFlash('commentDeleted', 'Vous avez réussi la suppression de
votre commentaire.');
```

```php
// Request #2
// affiche le message  flash nommé "commentDeleted"
echo $session->getFlash('commentDeleted');

// Request #3
// $result est faux puisque le message flash a été automatiquement supprimé
$result = $session->hasFlash('commentDeleted');
```

Comme les données de session ordinaires, vous pouvez stocker des données arbitraires sous forme de données flash.

Vous pouvez appeler `yii\web\Session::setFlash()`, cela écrase toute donnée flash préexistante qui a le même nom. Pour ajouter une nouvelle donnée flash à un message existant, vous pouvez utiliser `yii\web\Session::addFlash()` à la place. Par exemple :

```php
$session = Yii::$app->session;

// Request #1
// ajoute un message flash nommé  "alerts"
$session->addFlash('alerts', 'Vous avez réussi la suppression de votre
commentaire');
$session->addFlash('alerts', 'Vous avez réussi l'ajout d'un ami.');
$session->addFlash('alerts', 'Vous êtes promu.');

// Request #2
// $alerts est un tableau de messages flash nommé "alerts"
$alerts = $session->getFlash('alerts');
```

> **Note :** évitez d'utiliser `yii\web\Session::setFlash()` en même temps que `yii\web\Session::addFlash()` pour des données flash de même nom. C'est parce que la deuxième méthode transforme automatiquement les données flash en tableau pour pouvoir y ajouter des données. En conséquence, quand vous appelez `yii\web\Session::getFlash()`, vous pouvez parfois recevoir un tableau ou une chaîne de caractères selon l'ordre dans lequel ces méthodes ont été appelées.
>
> **Conseil :** pour afficher des messages Flash vous pouvez utiliser l'objet graphique `yii\bootstrap\Alert` de la manière suivante :
>
> ```php
> echo Alert::widget([
>    'options' => ['class' => 'alert-info'],
>    'body' => Yii::$app->session->getFlash('postDeleted'),
> ]);
> ```

4.5.2 Témoins de connexion

Yii représente chacun des témoins de connexion sous forme d'objet de classe `yii\web\Cookie`. Les objets `yii\web\Request` et `yii\web\Response` contiennent une collection de témoins de connexion via la propriété nommée

`cookies`. La collection de témoins de connexion dans le premier de ces objets est celle soumise dans une requête, tandis que celle du deuxième objet représente les témoins de connexion envoyés à l'utilisateur.

La partie de l'application qui traite la requête et la réponse directement est le contrôleur. Par conséquent, les témoins de connexion doivent être lus et envoyés dans le contrôleur.

Lecture des témoins de connexion

Vous pouvez obtenir les témoins de connexion de la requête courante en utilisant le code suivant :

```
// obtient la collection de témoins de connexion (yii\web\CookieCollection)
du composant "request"
$cookies = Yii::$app->request->cookies;

// obtient la valeur du témoin de connexion  "language". Si le témoin de
connexion n'existe pas, retourne  "en" par défaut.
$language = $cookies->getValue('language', 'en');

// une façon alternative d'obtenir la valeur du témoin de connexion
"language"
if (($cookie = $cookies->get('language')) !== null) {
    $language = $cookie->value;
}

// vous pouvez aussi utiliser  $cookies comme un tableau
if (isset($cookies['language'])) {
    $language = $cookies['language']->value;
}

// vérifie si un témoin de connexion "language" existe
if ($cookies->has('language')) ...
if (isset($cookies['language'])) ...
```

Envoi de témoins de connexion

Vous pouvez envoyer des témoins de connexion à l'utilisateur final avec le code suivant :

```
// obtient la collection de témoins de connexion (yii\web\CookieCollection)
du composant "response"
$cookies = Yii::$app->response->cookies;

// ajoute un témoin de connexion à la réponse à envoyer
$cookies->add(new \yii\web\Cookie([
    'name' => 'language',
    'value' => 'zh-CN',
]));

// supprime un cookie
```

```php
$cookies->remove('language');
// équivalent à
unset($cookies['language']);
```

En plus des propriétés `name (nom)`, `value (valeur)` montrées dans les exemples ci-dessus, la classe `yii\web\Cookie` définit également d'autres propriétés pour représenter complètement toutes les informations de témoin de connexion disponibles, comme les propriétés `domain (domaine)`, `expire (date d'expiration)`. Vous pouvez configurer ces propriété selon vos besoins pour préparer un témoin de connexion et ensuite l'ajouter à la collection de témoins de connexion de la réponse.

> **Note :** pour une meilleure sécurité, la valeur par défaut de la propriété `yii\web\Cookie::$httpOnly` est définie à `true`. Cela permet de limiter le risque qu'un script client n'accède à un témoin de connexion protégé (si le navigateur le prend en charge). Reportez-vous à l'article de wiki httpOnly [12] pour plus de détails.

Validation des témoins de connexion

Lorsque vous lisez ou envoyez des témoins de connexion via les composants `request` et `response` comme expliqué dans les sous-sections qui précèdent, vous appréciez la sécurité additionnelle de validation des témoins de connexion qui protège vos témoins de connexion de la modification côté client. Cela est réalisé en signant chacun des témoins de connexion avec une valeur de hachage qui permet à l'application de dire si un témoin de connexion a été modifié ou pas du côté client. Si c'est le cas, le témoin de connexion n'est PLUS accessible via la `collection de témoins de connexion` du composant `request`.

> **Note :** la validation des témoins de connexion ne protège que contre les effets de la modification des valeurs de témoins de connexion. Néanmoins, si un témoin de connexion ne peut être validé, vous pouvez continuer à y accéder via la variable globale `$_COOKIE`. Ceci est dû au fait que les bibliothèques de tierces parties peuvent manipuler les témoins de connexion d'une façon qui leur est propre, sans forcément impliquer la validation des témoins de connexion.

La validation des témoins de connexion est activée par défaut. Vous pouvez la désactiver en définissant la propriété `yii\web\Request::$enableCookieValidation` à `false` (faux) mais nous vous recommandons fortement de ne pas le faire.

> **Note :** les témoins de connexion qui sont lus/écrits directement via `$_COOKIE` et `setcookie()` ne seront PAS validés.

12. https://owasp.org/www-community/HttpOnly

Quand vous utilisez la validation des témoins de connexion, vous devez spécifier une [[yii\web\Request : :cookieValidationKey |clé de validation des témoins de connexion]] gui sera utilisée pour générer la valeur de hachage dont nous avons parlé plus haut. Vous pouvez faire ça en configurant le composant `request` dans la configuration de l'application configuration comme indiqué ci-après :

```
return [
    'components' => [
        'request' => [
            'cookieValidationKey' => 'entrez une clé secrète ici',
        ],
    ],
];
```

> **Info :** la `clé de validation des témoins de connexion (cookieValidationKey)` est un élément critique de la sécurité de votre application. Elle ne devrait être connue que des personnes à qui vous faites confiance. Ne le stockez pas dans le système de gestion des version.

Error : not existing file : runtime-url-handling.md

4.6 Gestion des erreurs

Yii inclut un `gestionnaire d'erreur` pré-construit qui rend la gestion des erreurs bien plus agréable qu'auparavant. En particulier, le gestionnaire d'erreurs de Yii possède les fonctionnalités suivantes pour améliorer la gestion des erreurs.

— Toutes les erreurs PHP non fatales (p. ex. avertissements, notifications) sont converties en exceptions susceptibles d'être interceptées.
— Les exceptions et les erreurs fatales sont affichées avec les informations détaillées de la pile des appels et les lignes de code source en mode *debug*.
— Prise en charge de l'utilisation d'une action de contrôleur dédiée à l'affichage des erreurs.
— Prise en charge de différents formats de réponse d'erreur.

Le `gestionnaire d'erreur` est activé par défaut. Vous pouvez le désactiver en définissant la constante `YII_ENABLE_ERROR_HANDLER` à `false` (faux) dans le script d'entrée de votre application.

4.6.1 Utilisation du gestionnaire d'erreurs

Le `gestionnaire d'erreurs` est enregistré en tant que composant d'application nommé `errorHandler`. Vous pouvez le configurer dans la configuration de l'application comme indiqué ci-dessous :

```
return [
    'components' => [
        'errorHandler' => [
            'maxSourceLines' => 20,
        ],
    ],
];
```

Avec la configuration qui précède, le nombre de lignes de code source à afficher dans les pages d'exception est limité à 20.

Comme cela a déjà été dit, le gestionnaire d'erreur transforme toutes les erreurs PHP non fatales en exception susceptibles d'être interceptées. Cela signifie que vous pouvez utiliser le code suivant pour vous servir de cette gestion d'erreurs :

```
use Yii;
use yii\base\ErrorException;

try {
    10/0;
} catch (ErrorException $e) {
    Yii::warning("Division by zero.");
}

// l'exécution continue...
```

Si vous désirez afficher une page d'erreur disant à l'utilisateur que sa requête est invalide ou inattendue, vous pouvez simplement lever une `exception HTTP`, comme l'exception `yii\web\NotFoundHttpException`. Le gestionnaire d'erreurs définit alors correctement le code d'état HTTP de la réponse et utilise une vue d'erreur appropriée pour afficher le message d'erreur.

```
use yii\web\NotFoundHttpException;

throw new NotFoundHttpException();
```

4.6.2 Personnalisation de l'affichage des erreurs

Le `gestionnaire d'erreurs` ajuste l'affichage de l'erreur en tenant compte de la valeur de la constante `YII_DEBUG`. Quand `YII_DEBUG` est égale à `true` (vrai) (ce qui signifie que le mode *debug* est activé), le gestionnaire d'erreurs affiche les exceptions avec le détail de la pile des appels et les lignes de code apportant de l'aide au débogage. Quand `YII_DEBUG` est égale à `false` (faux), seule le message d'erreur est affiché pour ne pas révéler des informations sensibles sur l'application.

> **Info :** si une exception est un descendant de la classe `yii\base \UserException`, aucune pile des appels n'est affichée, et ceci indépendamment de la valeur `YII_DEBUG`. Cela tient au fait que de telles exceptions résultent d'erreurs commises par l'utilisateur et que les développeurs n'ont rien à corriger.

Par défaut, le `gestionnaire d'erreurs` affiche les erreurs en utilisant deux vues :

- `@yii/views/errorHandler/error.php` : utilisée lorsque les erreurs doivent être affichées SANS les informations sur la pile des appels. Quand `YII_DEBUG` est égale à `false`, c'est la seule vue d'erreur à afficher.
- `@yii/views/errorHandler/exception.php` : utilisée lorsque les erreurs doivent être affichées AVEC les informations sur la pile des appels.

Vous pouvez configurer les propriétés `errorView` et `exceptionView` du gestionnaire d'erreur pour utiliser vos propres vues afin de personnaliser l'affichage des erreurs.

Utilisation des actions d'erreurs

Une meilleure manière de personnaliser l'affichage des erreurs est d'utiliser des actions d'erreur dédiées. Pour cela, commencez par configurer la propriété `errorAction` du composant `errorHandler` comme indiqué ci-après :

```
return [
    'components' => [
        'errorHandler' => [
            'errorAction' => 'site/error',
```

```
        ],
    ]
];
```

La propriété `errorAction` accepte une route vers une action. La configuration ci-dessus établit que lorsqu'une erreur doit être affichée sans information de la pile des appels, l'action `site/error` doit être exécutée.

Vous pouvez créer une action `site/error` comme ceci :

```
namespace app\controllers;

use Yii;
use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'error' => [
                'class' => 'yii\web\ErrorAction',
            ],
        ];
    }
}
```

Le code ci-dessus définit l'action `error` en utilisant la classe `yii\web\ErrorAction` qui rend une erreur en utilisant une vue nommée `error`.

En plus d'utiliser `yii\web\ErrorAction`, vous pouvez aussi définir l'action `error` en utilisant une méthode d'action similaire à la suivante :

```
public function actionError()
{
    $exception = Yii::$app->errorHandler->exception;
    if ($exception !== null) {
        return $this->render('error', ['exception' => $exception]);
    }
}
```

Vous devez maintenant créer un fichier de vue `views/site/error.php`. Dans ce fichier de vue, vous pouvez accéder aux variables suivantes si l'action d'erreur est définie en tant que `yii\web\ErrorAction` :

— `name` : le nom de l'erreur ;
— `message` : le message d'erreur ;
— `exception` : l'objet exception via lequel vous pouvez retrouver encore plus d'informations utiles telles que le code d'état HTTP, le code d'erreur, la pile des appels de l'erreur, etc.

Info : si vous êtes en train d'utiliser le modèle de projet *basic* ou le modèle de projet avancé [13], l'action d'erreur est la vue d'erreur sont déjà définies pour vous.

Note : si vous avez besoin de rediriger dans un gestionnaire d'erreur, faites-le de la manière suivante :

```
Yii::$app->getResponse()->redirect($url)->send();
return;
```

Personnalisation du format de la réponse d'erreur

Le gestionnaire d'erreurs affiche les erreurs en respectant le réglage de format de la réponse. Si le `format de la réponse` est `html`, il utilise la vue d'erreur ou d'exception pour afficher les erreurs, comme c'est expliqué dans la sous-section précédente. Pour les autres formats de réponse, le gestionnaire d'erreurs assigne la représentation de l'erreur sous forme de tableau à la propriété `yii\web\Response::$data` qui est ensuite convertie dans le format désiré. Par exemple, si le format de la réponse est `json`, vous pourriez voir une réponse similaire à la suivante :

```
HTTP/1.1 404 Not Found
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
    "code": 0,
    "status": 404
}
```

Vous pouvez personnaliser le format de réponse d'erreur en répondant à l'événement `beforeSend` du composant `response` dans la configuration de l'application :

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null) {
                    $response->data = [
                        'success' => $response->isSuccessful,
```

13. https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md

```
                    'data' => $response->data,
                ];
                $response->statusCode = 200;
            }
        },
    ],
],
];
```

Le code précédent formate la réponse d'erreur comme suit :

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "success": false,
    "data": {
        "name": "Not Found Exception",
        "message": "The requested resource was not found.",
        "code": 0,
        "status": 404
    }
}
```

4.7 Enregistrements des messages

Yii fournit une puissante base structurée (framework) d'enregistrement des messages qui est très personnalisable et extensible. En utilisant cette base structurée, vous pouvez facilement enregistrer des types variés de messages, les filtrer et les rassembler dans différentes cibles comme les bases de données et les courriels.

L'utilisation de la base structurée d'enregistrement des messages de Yii nécessite de suivre les étapes suivantes :

— Enregistrer les messages à différents endroits de votre code ;
— Configurer cibles d'enregistrement dans la configuration de l'application pour filtrer et exporter les messages enregistrés ;
— Examiner les messages enregistrés, filtrés et exportés par les différentes cibles (p. ex. débogueur de Yii).

Dans cette section, nous décrivons principalement les deux premières étapes.

4.7.1 Messages enregistrés

Enregistrer des messages est aussi simple que d'appeler une des méthodes suivantes :

- `Yii::debug()` : enregistre un message pour garder une trace de comment un morceau de code fonctionne. Cela est utilisé principalement en développement.
- `Yii::info()` : enregistre un message qui contient quelques informations utiles.
- `Yii::warning()` : enregistre un message d'avertissement qui indique que quelque chose d'inattendu s'est produit.
- `Yii::error()` : enregistre une erreur fatale qui doit être analysée dès que possible.

Ces méthodes enregistrent les messages à différents niveaux de sévérité et dans différentes catégories. Elles partagent la même signature `function ($message, $category = 'application')`, où `$message` représente le message à enregistrer, tandis que `$category` est la catégorie de ce message. Le code de l'exemple qui suit enregistre un message de trace dans la catégorie `application` :

```
Yii::debug('start calculating average revenue');
```

> **Info :** les messages enregistrés peuvent être des chaînes de caractères aussi bien que des données complexes telles que des tableaux ou des objets. Il est de la responsabilité des cibles d'enregistrement de traiter correctement ces messages. Par défaut, si un message enregistré n'est pas un chaîne de caractères, il est exporté comme une chaîne de caractères en appelant la méthode `yii\helpers\VarDumper::export()`.

Pour mieux organiser et filtrer les messages enregistrés, il est recommandé que vous spécifiiez une catégorie appropriée pour chacun des messages. Vous pouvez choisir une schéma de nommage hiérarchisé pour les catégories, ce qui facilitera le filtrage des messages par les cibles d'enregistrement sur la base de ces catégories. Un schéma de nommage simple et efficace est d'utiliser la constante magique `__METHOD__` de PHP dans les noms de catégorie. Par exemple :

```
Yii::debug('start calculating average revenue', __METHOD__);
```

La constante magique `__METHOD__` est évaluée comme le nom de la méthode (préfixée par le nom pleinement qualifié de la classe), là où la constante apparaît. Par exemple, elle est égale à `'app\controllers\RevenueController::calculate'` si la ligne suivante est utilisée dans cette méthode.

> **Info :** les méthodes d'enregistrement décrites plus haut sont en fait des raccourcis pour la méthode `log()` de l'`objet logger` qui est un singleton accessible via l'expression `Yii::getLogger()`. Lorsque suffisamment de messages ont été enregistrés, ou quand l'application se termine, l'objet *logger* appelle un `distributeur de messages` pour envoyer les messages enregistrés aux cibles d'enregistrement.

4.7.2 Cibles d'enregistrement

Une cible d'enregistrement est une instance de la classe `yii\log\Target` ou d'une de ses classe filles. Elle filtre les messages enregistrés selon leur degré de sévérité et leur catégorie et les exporte vers un média donné. Par exemple, une `cible base données` exporte les messages enregistrés et filtrés vers une base de données, tandis qu'une `cible courriel` exporte les messages vers l'adresse de courriel spécifiée.

Vous pouvez enregistrer plusieurs cibles d'enregistrement dans votre application en les configurant, via le composant d'application`log` dans la configuration de l'application, de la manière suivante :

```
return [
    // le composant "log" doit être chargé lors de la période d'amorçage
    'bootstrap' => ['log'],
    // le composant "log" traite les messages avec un horodatage
    (timestamp). Définissez le fuseau horaire  PHP pour créer l'horodate
    correcte :
    'timeZone' => 'America/Los_Angeles',

    'components' => [
        'log' => [
            'targets' => [
                [
                    'class' => 'yii\log\DbTarget',
                    'levels' => ['error', 'warning'],
                ],
                [
                    'class' => 'yii\log\EmailTarget',
                    'levels' => ['error'],
                    'categories' => ['yii\db\*'],
                    'message' => [
                       'from' => ['log@example.com'],
                       'to' => ['admin@example.com',
                       'developer@example.com'],
                       'subject' => 'Database errors at example.com',
                    ],
                ],
            ],
        ],
    ],
];
```

> **Note :** le composant `log` doit être chargé durant le processus d'amorçage afin qu'il puisse distribuer les messages enregistrés aux cibles rapidement. C'est pourquoi il est listé dans le tableau `bootstrap` comme nous le montrons ci-dessus.

Dans le code précédent, deux cibles d'enregistrement sont enregistrées dan la propriété `yii\log\Dispatcher::$targets` :

— la première cible sélectionne les messages d'erreurs et les avertissements et les sauvegarde dans une table de base de données ;

— la deuxième cible sélectionne les messages d'erreur dont le nom de la catégorie commence par `yii\db\`, et les envoie dans un courriel à la fois à `admin@example.com` et à `developer@example.com`.

Yii est fourni avec les cibles pré-construites suivantes. Reportez-vous à la documentation de l'API pour en savoir plus sur ces classes, en particulier comment les configurer et les utiliser.

— `yii\log\DbTarget` : stocke les messages enregistrés dans une table de base de données.
— `yii\log\EmailTarget` : envoie les messages enregistrés vers une adresse de courriel spécifiée préalablement.
— `yii\log\FileTarget` : sauvegarde les messages enregistrés dans des fichiers.
— `yii\log\SyslogTarget` : sauvegarde les messages enregistrés vers *syslog* en appelant la fonction PHP `syslog()`.

Dans la suite de ce document, nous décrivons les fonctionnalités communes à toutes les cibles d'enregistrement.

Filtrage des messages

Vous pouvez configurer les propriétés `levels` et `categories` de chacune des cibles d'enregistrement pour spécifier les niveaux de sévérité et les catégories que la cible doit traiter.

La propriété `levels` accepte un tableau constitué d'une ou plusieurs des valeurs suivantes :

— `error` : correspondant aux messages enregistrés par `Yii::error()`.
— `warning` : correspondant aux messages enregistrés par `Yii::warning()`.
— `info` : correspondant aux messages enregistrés par `Yii::info()`.
— `trace` : correspondant aux messages enregistrés par `Yii::debug()`.
— `profile` : correspondant aux messages enregistrés par `Yii::beginProfile()` et `Yii::endProfile()`, et qui sera expliqué en détails dans la sous-section Profilage de la performance.

Si vous ne spécifiez pas la propriété `levels`, cela signifie que la cible traitera les messages de *n'importe quel* niveau de sévérité.

La propriété `categories` accepte un tableau constitué de noms ou de motifs de noms de catégorie de messages. Une cible ne traite que les messages dont la catégorie est trouvée ou correspond aux motifs de ce tableau. Un motif de nom de catégorie est un préfixe de nom de catégorie suivi d'une astérisque `*`. Un nom de catégorie correspond à un motif de nom de catégorie s'il commence par le préfixe du motif. Par exemple, `yii\db\Command::execute` et `yii\db\Command::query` sont utilisés comme noms de catégorie pour les messages enregistrés dans la classe `yii\db\Command`. Ils correspondent tous deux au motif `yii\db\*`.

Si vous ne spécifiez pas la propriété `categories`, cela signifie que le cible traite les messages de *n'importe quelle* catégorie.

En plus d'inscrire des catégories en liste blanche via la propriété `categories`, vous pouvez également inscrire certaines catégories en liste noire via la propriété `except`. Si la catégorie d'un message est trouvée ou correspond à un des motifs de cette propriété, ce message n'est PAS traité par la cible.

La configuration suivante de cible spécifie que la cible traitera les messages d'erreur ou d'avertissement des catégories dont le nom correspond soit à `yii\db\*`, soit `yii\web\HttpException:*`, mais pas `yii\web\HttpException:404`.

```
[
    'class' => 'yii\log\FileTarget',
    'levels' => ['error', 'warning'],
    'categories' => [
        'yii\db\*',
        'yii\web\HttpException:*',
    ],
    'except' => [
        'yii\web\HttpException:404',
    ],
]
```

> **Info :** lorsqu'une exception HTTP est capturée par le gestionnaire d'erreur, un message d'erreur est enregistré avec un non de catégorie dont le format est `yii\web\HttpException:ErrorCode`. Par exemple, l'exception `yii\web\NotFoundHttpException` provoque un message d'erreur de catégorie `yii\web\HttpException:404`.

Formatage des messages

Les cibles d'enregistrement exportent les messages enregistrés et filtrés dans un certain format. Par exemple, si vous installez une cible d'enregistrement de classe `yii\log\FileTarget`, vous pouvez trouver un message enregistré similaire au suivant dans le fichier `runtime/log/app.log` file :

```
2014-10-04 18:10:15 [::1][][-][trace][yii\base\Module::getModule] Loading
module: debug
```

Par défaut, les messages enregistrés sont formatés comme suit par la méthode `yii\log\Target::formatMessage()` :

```
Horodate [adresse IP][identifiant utilisateur][identifiant de
session][niveau de sévérité][catégorie] Texte du message
```

Vous pouvez personnaliser ce format en configurant la propriété `yii\log\Target::$prefix` qui accepte une fonction PHP appelable qui retourne un message de préfixe personnalisé. Par exemple, le code suivant configure une cible d'enregistrement pour qu'elle préfixe chaque message enregistré avec l'identifiant de l'utilisateur courant (l'adresse IP et l'identifiant de session étant retirés pour des raisons de protection de la vie privée).

```
[
    'class' => 'yii\log\FileTarget',
    'prefix' => function ($message) {
        $user = Yii::$app->has('user', true) ? Yii::$app->get('user') :
        null;
        $userID = $user ? $user->getId(false) : '-';
        return "[$userID]";
    }
]
```

En plus des préfixes de messages, les cibles d'enregistrement ajoutent aussi quelques informations de contexte à chaque lot de messages enregistrés. Par défaut, les valeurs de ces variables PHP globales sont incluses : `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION` et `$_SERVER`. Vous pouvez ajuster ce comportement en configurant la propriété `yii\log\Target::$logVars` avec les noms des variables globales que vous voulez que la cible d'enregistrement inclue. Par exemple, la cible d'enregistrement suivante spécifie que seules les valeurs de la variable `$_SERVER` seront ajoutées aux messages enregistrés. `php [

```
'class' => 'yii\log\FileTarget',
'logVars' => ['_SERVER'],
```

] `

Vous pouvez configurer `logVars` comme un tableau vide pour désactiver totalement l'inclusion d'informations de contexte. Ou si vous voulez mettre en œuvre votre propre façon de fournir les informations contextuelles, vous pouvez redéfinir la méthode `yii\log\Target::getContextMessage()`.

Niveaux de la trace de message

Lors du développement, vous cherchez souvent à voir d'où provient chacun des messages enregistrés. Cela est possible en configurant la propriété `traceLevel` du composant`log` de la façon suivante :

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [...],
        ],
    ],
];
```

La configuration de l'application ci-dessus statue que le `niveau de trace` sera 3 si `YII_DEBUG` est activé et 0 si `YII_DEBUG` est désactivé. Cela veut dire que, si `YII_DEBUG` est activé, au plus trois niveaux de la pile des appels seront ajoutés à chaque message enregistré, là où le messages est enregistré ; et, si `YII_DEBUG` est désactivé, aucune information de la pile des appels ne sera incluse.

> **Info :** obtenir les informations de la pile des appels n'a rien de trivial. En conséquence, vous ne devriez utiliser cette fonctionnalité que durant le développement ou le débogage d'une application.

Purge et exportation des messages

Comme nous l'avons dit plus haut, les messages enregistrés sont conservés dans un tableau par l'`objet` . Pour limiter la consommation de mémoire par ce tableau, l'objet *logger* purge les messages enregistrés vers les cibles d'enregistrement chaque fois que leur nombre atteint une certaine valeur. Vous pouvez personnaliser ce nombre en configurant la propriété `flushInterval` du composant `log` :

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 100,   // default is 1000
            'targets' => [...],
        ],
    ],
];
```

> **Info :** la purge des messages intervient aussi lorsque l'application se termine, ce qui garantit que les cibles d'enregistrement reçoivent des messages enregistrés complets.

Lorsque l'`objet` purge les messages enregistrés vers les cibles d'enregistrement, ils ne sont pas exportés immédiatement. Au lieu de cela, l'exportation des messages ne se produit que lorsque la cible d'enregistrement a accumulé un certain nombre de messages filtrés. Vous pouvez personnaliser ce nombre en configurant la propriété `exportInterval` de chacune des cibles d'enregistrement, comme ceci :

```
[
    'class' => 'yii\log\FileTarget',
    'exportInterval' => 100,  // default is 1000
]
```

À cause des niveaux de purge et d'exportation, par défaut, lorsque vous appelez `Yii::debug()` ou toute autre méthode d'enregistrement, vous ne voyez PAS immédiatement le message enregistré dans la cible. Cela peut représenter un problème pour pour certaines applications de console qui durent longtemps. Pour faire en sorte que les messages apparaissent immédiatement dans les cibles d'enregistrement, vous devriez définir les propriétés `flushInterval` et `exportInterval` toutes deux à 1, comme montré ci-après :

```
return [
    'bootstrap' => ['log'],
```

```
    'components' => [
        'log' => [
            'flushInterval' => 1,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'exportInterval' => 1,
                ],
            ],
        ],
    ],
];
```

> **Note :** la purge et l'exportation fréquentes de vos messages dégradent la performance de votre application.

Activation, désactivation des cibles d'enregistrement

Vous pouvez activer ou désactiver une cible d'enregistrement en configurant sa propriété `enabled`. Vous pouvez le faire via la configuration de la cible d'enregistrement ou en utilisant l'instruction suivante dans votre code PHP :

```
Yii::$app->log->targets['file']->enabled = false;
```

Le code ci-dessus, nécessite que nommiez une cible `file`, comme montré ci-dessous en utilisant des clés sous forme de chaînes de caractères dans le tableau `targets` :

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'targets' => [
                'file' => [
                    'class' => 'yii\log\FileTarget',
                ],
                'db' => [
                    'class' => 'yii\log\DbTarget',
                ],
            ],
        ],
    ],
];
```

Création d'une cible d'enregistrement

La création d'une classe de cible d'enregistrement est très simple. Vous devez essentiellement implémenter `yii\log\Target::export()` en envoyant le contenu du tableau des `yii\log\Target::$messages` vers un média désigné.

Vous pouvez appeler la méthode `yii\log\Target::formatMessage()` pour formater chacun des messages. Pour plus de détails, reportez-vous à n'importe quelle classe de cible de messages incluse dans la version de Yii.

> **Conseil :** au lieu de créer vos propres journaliseurs, vous pouvez essayez n'importe quel journaliseur compatible PSR-3 tel que Monolog [14] en utilisant PSR log target extension [15].

4.7.3 Profilage de la performance

Le profilage de la performance est un type particulier d'enregistrement de messages qui est utilisé pour mesurer le temps d'exécution de certains blocs de code et pour déterminer les goulots d'étranglement. Par exemple, la classe `yii\db\Command` utilise le profilage de performance pour connaître le temps d'exécution de chacune des requêtes de base de données.

Pour utiliser le profilage de la performance, commencez par identifier les blocs de code qui ont besoin d'être profilés. Puis, entourez-les de la manière suivante :

```
\Yii::beginProfile('myBenchmark');

...le bloc de code à profiler...

\Yii::endProfile('myBenchmark');
```

où `myBenchmark` représente un jeton unique identifiant un bloc de code. Plus tard, lorsque vous examinez le résultat du profilage, vous pouvez utiliser ce jeton pour localiser le temps d'exécution du bloc correspondant.

Il est important de vous assurer que les paires `beginProfile` et `endProfile` sont correctement imbriquées. Par exemple,

```
\Yii::beginProfile('block1');

    // du code à profiler

    \Yii::beginProfile('block2');
        // un autre bloc de code à profiler
    \Yii::endProfile('block2');

\Yii::endProfile('block1');
```

Si vous omettez `\Yii::endProfile('block1')` ou inversez l'ordre de `\Yii::endProfile('block1')` et de `\Yii::endProfile('block2')`, le profilage de performance ne fonctionnera pas.

Pour chaque bloc de code profilé, un message est enregistré avec le niveau de sévérité `profile`. Vous pouvez configurer une cible d'enregistrement pour

14. https://github.com/Seldaek/monolog
15. https://github.com/samdark/yii2-psr-log-target

collecter de tels messages et les exporter. L'outil *Yii debugger* comporte un panneau d'affichage pré-construit des résultats de profilage.

Chapitre 5

Concepts Clés

5.1 Composants

Les composants sont les blocs de constructions principaux de vos applications Yii. Les composants sont des instances de la classe `yii\base\Component`, ou de ses classes filles. Les trois fonctionnalités principales fournies par les composants aux autres classes sont :

- Les propriétés ;
- Les événements ;
- Les comportements.

Séparément et en combinaisons, ces fonctionnalités rendent les classes de Yii beaucoup plus personnalisables et faciles à utiliser. Par exemple, l'`yii\jui\DatePicker` inclus, un composant d'interface utilisateur, peut être utilisé dans une vue pour générer un sélecteur de date interactif :

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'ru',
    'name'  => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

Les propriétés de l'objet graphique sont faciles à écrire car la classe étend `yii\base\Component`.

Tandis que les composants sont très puissants, ils sont un peu plus lourds que les objets normaux. Cela est dû au fait que, en particulier, la prise en charge des fonctionnalités event et behavior requiert un peu plus de mémoire et de temps du processeur. Si vos composants n'ont pas besoin de ces deux fonctionnalités, vous devriez envisager d'étendre la classe `yii\base\BaseObject` au lieu de la classe `yii\base\Component`. Ce faisant,

votre composant sera aussi efficace que les objets PHP normaux, mais avec la prise en charge des propriétés.

Lorsque votre classe étend la classe `yii\base\Component` ou `yii\base\BaseObject`, il est recommandé que suiviez ces conventions :

— Si vous redéfinissez le constructeur, spécifiez un paramètre `$config` en tant que *dernier* paramètre du constructeur et passez le au constructeur du parent.
— Appelez toujours le constructeur du parent *à la fin* de votre constructeur redéfini.
— Si vous redéfinissez la méthode `yii\base\BaseObject::init()`, assurez-vous que vous appelez la méthode `init()` mise en œuvre par le parent *au début* de votre méthodes `init()`.

Par exemple :

```php
<?php

namespace yii\components\MyClass;

use yii\base\BaseObject;

class MyClass extends BaseObject
{
    public $prop1;
    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... initialisation avant l'application de la configuration

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();

        // ... initialization après l'application de la configuration
    }
}
```

Le respect de ces conseils rend vos composants configurables lors de leur création. Par exemple :

```php
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
// alternatively
$component = \Yii::createObject([
    'class' => MyClass::class,
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

> **Info :** bien que l'approche qui consiste à appeler la méthode `Yii::createObject()` semble plus compliquée, elle est plus puissante car elle est mise en œuvre sur un conteneur d'injection de dépendances.

La classe `yii\base\BaseObject` fait appliquer le cycle de vie suivant de l'objet :

1. Pré-initialisation dans le constructeur. Vous pouvez définir les propriétés par défaut à cet endroit.
2. Configuration de l'objet via `$config`. La configuration peut écraser les valeurs par défaut définies dans le constructeur.
3. Post-initialisation dans la méthode `init()`. Vous pouvez redéfinir cette méthode pour effectuer des tests sanitaires et normaliser les propriétés.
4. Appel des méthodes de l'objet.

Les trois premières étapes arrivent toutes durant la construction de l'objet. Cela signifie qu'une fois que vous avez obtenu une instance de la classe (c.-à-d. un objet), cet objet a déjà été initialisé dans un état propre et fiable.

5.2 Propriétés

En PHP, les variables membres des classes sont aussi appelées *propriétés*. Ces variables font partie de la définition de la classe et sont utilisées pour représenter l'état d'une instance de cette classe (c.-à-d. à différentier une instance de la classe d'une autre). En pratique, vous désirez souvent gérer la lecture et l'écriture de ces propriété d'une manière particulière. Par exemple, vous pouvez désirer qu'une chaîne de caractères soit toujours nettoyée avant de l'assigner à une propriété `label`. Vous *pouvez* utiliser le code suivant pour arriver à cette fin :

```
$object->label = trim($label);
```

Le revers du code ci-dessus est que vous devez appeler `trim()` partout ou vous voulez définir la propriété `label`. Si, plus tard, la propriété `label` devient sujette à de nouvelles exigences, telles que la première lettre doit être une capitale, vous auriez à modifier toutes les parties de code qui assignent une valeur à la propriété `label`. La répétition de code conduit à des bogues, et c'est une pratique courante de l'éviter autant que faire se peut.

Pour résoudre ce problème, Yii introduit une classe de base nommée `yii\base\BaseObject` qui prend en charge la définition de propriétés sur la base de méthodes d'obtention (*getter*) et de méthode d'assignation (*setters*). Si une classe a besoin de cette fonctionnalité, il suffit qu'elle étende la classe`yii\base\BaseObject`, ou une de ses classes filles.

> **Info :** presque toutes les classes du noyau du framework Yii étendent la classe `yii\base\BaseObject` ou une de ses classes filles. Cela veut dire, que chaque fois que vous trouvez une méthode d'obtention ou d'assignation dans une classe du noyau, vous pouvez l'utiliser comme une propriété.

Une méthode d'obtention est une méthode dont le nom commence par le mot `get` (obtenir) et une méthode d'assignation est une méthode dont le nom commence par le mot `set` (assigner, définir). Le nom après les mots préfixes `get` ou `set` définit le nom d'une propriété. Par exemple, une méthode d'obtention `getLabel` et/ou une méthode d'assignation `setLabel` obtient et assigne, respectivement, une propriété nommée `label`, comme le montre le code suivant :

```
namespace app\components;

use yii\base\BaseObject;

class Foo extends BaseObject
{
    private $_label;

    public function getLabel()
    {
        return $this->_label;
    }

    public function setLabel($value)
    {
        $this->_label = trim($value);
    }
}
```

Pour être tout à fait exact, les méthodes d'obtention et d'assignation créent la propriété `label`, qui dans ce cas fait référence en interne à une propriété privée nommée `_label`.

Les propriétés définies par les méthodes d'obtention et d'assignation peuvent être utilisées comme des variables membres de la classe. La différence principale est que, lorsqu'une telle propriété est lue, la méthode d'obtention correspondante est appelée ; lorsqu'une valeur est assignée à la propriété, la méthode d'assignation correspondante est appelée. Par exemple :

```
// équivalent à $label = $object->getLabel();
$label = $object->label;

// équivalent à $object->setLabel('abc');
$object->label = 'abc';
```

Une propriété définie par une méthode d'obtention (*getter*) sans méthode d'assignation (*setter*) est une propriété *en lecture seule*. Essayer d'assigner

une valeur à une telle propriété provoque une exception `InvalidCallException`. De façon similaire, une propriété définie par une méthode d'assignation sans méthode d'obtention est *en écriture seule*. Essayer de lire une telle propriété provoque une exception. Il n'est pas courant d'avoir des propriétés *en écriture seule*.

Il existe plusieurs règles spéciales pour les propriétés définies via des méthodes d'obtention et d'assignation, ainsi que certaines limitations sur elles.

— Le nom de telles propriétés sont *insensibles à la casse*. Par exemple, `$object->label` et `$object->Label` sont identiques. Cela est dû au fait que le nom des méthodes dans PHP est insensible à la casse.
— Si le nom d'une telle propriété est identique à celui d'une variable membre de la classe, le dernier prévaut. Par exemple, si la classe ci-dessus `Foo` possède une variable momée `label`, alors l'assignation `$object->label = 'abc'` affecte la *variable membre* `label` ; cette ligne ne fait pas appel à la méthode d'assignation `setLabel()`.
— Ces propriétés ne prennent pas en charge la visibilité. Cela ne fait aucune différence pour les méthodes d'obtention et d'assignation qui définissent une propriété, que cette propriété soit publique, protégée ou privée.
— Les propriétés peuvent uniquement être définies par des méthodes d'obtention et d'assignation *non-statiques*. Les méthodes statiques ne sont pas traitées de la même manière.
— Un appel normal à la méthode `property_exists()` ne fonctionne pas pour déterminer des propriétés magiques. Vous devez appeler `canGetProperty()` ou `canSetProperty()` respectivement.

En revenant au problème évoqué au début de ce guide, au lieu d'appeler `trim()` partout où une valeur est assignée à `label`, vous pouvez vous contenter d'appeler `trim()` dans la méthode d'assignation `setLabel()`. Et si une nouvelle exigence apparaît – comme celle de mettre la première lettre en capitale – la méthode `setLabel()` peut être rapidement modifiée sans avoir à toucher à d'autres parties du code. Cet unique modification affecte l'ensemble des assignation de `label`.

5.3 Événements

Les événement vous permettent d'injecter du code personnalisé dans le code existant à des points précis de son exécution. Vous pouvez attacher du code personnalisé à un événement de façon à ce que, lorsque l'événement est déclenché, le code s'exécute automatiquement. Par exemple, un objet serveur de courriel peut déclencher un événement `messageSent` (message envoyé) quand il réussit à envoyer un message. Si vous voulez conserver une trace des messages dont l'envoi a réussi, vous pouvez simplement attacher le code

de conservation de la trace à l'événement `messageSent`.

Yii introduit une classe de base appelée `yii\base\Component` pour prendre en charge les événements. Si une classe a besoin de déclencher des événements, elle doit étendre la classe `yii\base\Component`, ou une de ses classes filles.

5.3.1 Gestionnaires d'événements

Un gestionnaire d'événement est une fonction de rappel PHP [1] qui est exécutée lorsque l'événement à laquelle elle est attachée est déclenché. Vous pouvez utiliser n'importe laquelle des fonctions de rappel suivantes :

- une fonction PHP globale spécifiée sous forme de chaîne de caractères (sans les parenthèses) p. ex., `'trim'` ;
- une méthode d'objet spécifiée sous forme de tableau constitué d'un nom d'objet et d'un nom de méthode sous forme de chaîne de caractères (sans les parenthèses), p. ex., `[$object, 'methodName']` ;
- une méthode d'une classe statique spécifiée sous forme de tableau constitué d'un nom de classe et d'un nom de méthode sous forme de chaîne de caractères (sans les parenthèses), p. ex., `['ClassName', 'methodName']` ;
- une fonction anonyme p. ex., `function ($event) { ... }`.

La signature d'un gestionnaire d'événement est :

```
function ($event) {
    // $event est un objet de la classe  yii\base\Event ou des ses classes
    filles
}
```

Via le paramètre `$event`, un gestionnaire d'événement peut obtenir l'information suivante sur l'événement qui vient de se produire :

- le `nom de l'événement` ;
- l'`émetteur de l'événement` : l'objet dont la méthode `trigger()` a été appelée ;
- les `données personnalisées` : les données fournies lorsque le gestionnaire d'événement est attaché (les explications arrivent bientôt).

5.3.2 Attacher des gestionnaires d'événements

Vous pouvez attacher un gestionnaire d'événement en appelant la méthode `yii\base\Component::on()` du composant. Par exemple :

```
$foo = new Foo();

// le gestionnaire est une fonction globale
$foo->on(Foo::EVENT_HELLO, 'function_name');
```

1. https://www.php.net/manual/fr/language.types.callable.php

```php
// le gestionnaire est une méthode d'objet
$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);

// le gestionnaire est une méthode d'une classe statique
$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// le gestionnaire est un fonction anonyme
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // event handling logic
});
```

Vous pouvez aussi attacher des gestionnaires d'événements via les configurations. Pour plus de détails, reportez-vous à la section Configurations.

Lorsque vous attachez un gestionnaire d'événement, vous pouvez fournir des données additionnelles telles que le troisième paramètre de `yii\base\Component::on()`. Les données sont rendues disponibles au gestionnaire lorsque l'événement est déclenché et que le gestionnaire est appelé. Par exemple :

```php
// Le code suivant affiche  "abc" lorsque l'événement est déclenché
// parce que  $event->data contient les données passées en tant que
troisième argument à la méthode "on"
$foo->on(Foo::EVENT_HELLO, 'function_name', 'abc');

function function_name($event) {
    echo $event->data;
}
```

5.3.3 Ordre des gestionnaires d'événements

Vous pouvez attacher un ou plusieurs gestionnaires à un seul événement. Lorsqu'un événement est déclenché, les gestionnaires attachés sont appelés dans l'ordre dans lequel ils ont été attachés à l'événement. Si un gestionnaire a besoin d'arrêter l'appel des gestionnaires qui viennent après lui, il doit définir la propriété [[yii\base\Event : :handled (géré)]] du paramètre `$event` à `true` :

```php
$foo->on(Foo::EVENT_HELLO, function ($event) {
    $event->handled = true;
});
```

Par défaut, un gestionnaire nouvellement attaché est ajouté à la file des gestionnaires de l'événement. En conséquence, le gestionnaire est appelé en dernier lorsque l'événement est déclenché. Pour insérer un événement nouvellement attaché en tête de file pour qu'il soit appelé le premier, vous devez appeler `yii\base\Component::on()`, en lui passant `false` pour le quatrième paramètre `$append` :

```php
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // ...
}, $data, false);
```

5.3.4 Déclenchement des événements

Les événements sont déclenchés en appelant la méthode `yii\base\Component::trigger()`. La méthode requiert un *nom d'événement* et, en option, un objet événement qui décrit les paramètres à passer aux gestionnaires de cet événement. Par exemple :

```
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class Foo extends Component
{
    const EVENT_HELLO = 'hello';

    public function bar()
    {
        $this->trigger(self::EVENT_HELLO);
    }
}
```

Avec le code précédent, tout appel à `bar()` déclenche un événement nommé `hello`.

> **Conseil :** il est recommandé d'utiliser des constantes de classe pour représenter les noms d'événement. Dans l'exemple qui précède, la constante `EVENT_HELLO` représente l'événement `hello`. Cette approche procure trois avantages. Primo, elle évite les erreurs de frappe. Secundo, elle permet aux événements d'être reconnus par le mécanisme d'auto-complètement des EDI. Tertio, vous pouvez dire quels événements sont pris en charge par une classe en vérifiant la déclaration de ses constantes.

Parfois, lors du déclenchement d'un événement, vous désirez passer des informations additionnelles aux gestionnaires de cet événement. Par exemple, un serveur de courriels peut souhaiter passer les informations sur le message aux gestionnaires de l'événement `messageSent` pour que ces derniers soient informés de certaines particularités des messages envoyés. Pour ce faire, vous pouvez fournir un objet événement comme deuxième paramètre de la méthode `yii\base\Component::trigger()`. L'objet événement doit simplement être une instance de la classe `yii\base\Event` ou d'une de ses classes filles. Par exemple :

```
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class MessageEvent extends Event
```

```
{
    public $message;
}

class Mailer extends Component
{
    const EVENT_MESSAGE_SENT = 'messageSent';

    public function send($message)
    {
        // ...sending $message...

        $event = new MessageEvent;
        $event->message = $message;
        $this->trigger(self::EVENT_MESSAGE_SENT, $event);
    }
}
```

Lorsque la méthode `yii\base\Component::trigger()` est appelée, elle appelle tous les gestionnaires attachés à l'événement nommé.

5.3.5 Détacher des gestionnaires d'événements

Pour détacher un gestionnaire d'un événement, appelez la méthode `yii\base\Component::off()`. Par exemple :

```
// le gestionnaire est une fonction globale
$foo->off(Foo::EVENT_HELLO, 'function_name');

// le gestionnaire est une méthode d'objet
$foo->off(Foo::EVENT_HELLO, [$object, 'methodName']);

// le gestionnaire est une méthode d'une classe statique
$foo->off(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// le gestionnaire est une fonction anonyme
$foo->off(Foo::EVENT_HELLO, $anonymousFunction);
```

Notez qu'en général, vous ne devez pas essayer de détacher une fonction anonyme sauf si vous l'avez stockée quelque part lorsque vous l'avez attachée à un événement. Dans l'exemple ci-dessus, on suppose que la fonctions anonyme est stockée dans une variable nommée `$anonymousFunction`.

Pour détacher *tous* les gestionnaires d'un événement, appelez simplement la méthode `yii\base\Component::off()` sans le deuxième paramètre :

```
$foo->off(Foo::EVENT_HELLO);
```

5.3.6 Gestionnaire d'événement au niveau de la classe

Les sections précédent décrivent comment attacher un gestionnaire à un événement au *niveau d'une instance*. Parfois, vous désirez répondre à un

événement déclenché par *chacune des* instances d'une classe plutôt que par une instance spécifique. Au lieu d'attacher l'événement à chacune des instances, vous pouvez attacher le gestionnaire au *niveau de la classe* en appelant la méthode statique `yii\base\Event::on()`.

Par exemple, un objet Active Record déclenche un événement `EVENT_AFTER_INSERT` à chaque fois qu'il insère un nouvel enregistrement dans la base de données. Afin de suivre les insertions faites par tous les objets Active Record, vous pouvez utiliser le code suivant :

```
use Yii;
use yii\base\Event;
use yii\db\ActiveRecord;

Event::on(ActiveRecord::class, ActiveRecord::EVENT_AFTER_INSERT, function
($event) {
    Yii::debug(get_class($event->sender) . ' is inserted');
});
```

Le gestionnaire d'événement est invoqué à chaque fois qu'une instance de la classe `ActiveRecord`, ou d'une de ses classes filles, déclenche l'événement `EVENT_AFTER_INSERT`. Dans le gestionnaire, vous pouvez obtenir l'objet qui a déclenché l'événement via `$event->sender`.

Lorsqu'un objet déclenche un événement, il commence par appeler les gestionnaires attachés au niveau de l'instance, puis les gestionnaires attachés au niveau de la classe.

Vous pouvez déclencher un événement au *niveau de la classe* en appelant la méthode statique `yii\base\Event::trigger()`. Un événement déclenché au niveau de la classe n'est associé à aucun objet en particulier. En conséquence, il provoque l'appel des gestionnaires attachés au niveau de la classe seulement. Par exemple :

```
use yii\base\Event;

Event::on(Foo::class, Foo::EVENT_HELLO, function ($event) {
    var_dump($event->sender);  // displays "null"
});

Event::trigger(Foo::class, Foo::EVENT_HELLO);
```

Notez que, dans ce cas, `$event->sender` fait référence au nom de la classe qui a déclenché l'événement plutôt qu'à une instance de classe.

> **Note :** comme les gestionnaires attachés au niveau de la classe répondent aux événements déclenchés par n'importe quelle instance de cette classe, ou de ses classes filles, vous devez utiliser cette fonctionnalité avec précaution, en particulier si la classe est une classe de bas niveau comme la classe `yii\base\BaseObject`.

Pour détacher un gestionnaire attaché au niveau de la classe, appelez `yii\base\Event::off()`. Par exemple :

```
// détache $handler
Event::off(Foo::class, Foo::EVENT_HELLO, $handler);

// détache tous les gestionnaires de Foo::EVENT_HELLO
Event::off(Foo::class, Foo::EVENT_HELLO);
```

5.3.7 Événement utilisant des interfaces

Il y a encore une manière plus abstraite d'utiliser les événements. Vous pouvez créer une interface séparée pour un événement particulier et l'implémenter dans des classes dans lesquelles vous en avez besoin.

Par exemple, vous pouvez créer l'interface suivante :

```
namespace app\interfaces;

interface DanceEventInterface
{
    const EVENT_DANCE = 'dance';
}
```

Et ajouter deux classes qui l'implémente :

```
class Dog extends Component implements DanceEventInterface
{
    public function meetBuddy()
    {
        echo "Woof!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}

class Developer extends Component implements DanceEventInterface
{
    public function testsPassed()
    {
        echo "Yay!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}
```

Pour gérer l'événement `EVENT_DANCE` déclenché par n'importe laquelle de ces classes, appelez `Event::on()` et passez-lui le nom de l'interface comme premier argument :

```
Event::on('app\interfaces\DanceEventInterface',
DanceEventInterface::EVENT_DANCE, function ($event) {
    Yii::debug(get_class($event->sender) . ' just danced'); // Will log that
    Dog or Developer danced
});
```

Vous pouvez déclencher l'événement de ces classes :

```
// trigger event for Dog class
Event::trigger(Dog::class, DanceEventInterface::EVENT_DANCE);

// trigger event for Developer class
Event::trigger(Developer::class, DanceEventInterface::EVENT_DANCE);
```

Notez bien que vous ne pouvez pas déclencher l'événement de toutes les classes qui implémentent l'interface :,

```
// NE FONCTIONNE PAS
Event::trigger('app\interfaces\DanceEventInterface',
DanceEventInterface::EVENT_DANCE);
```

Pour détacher le gestionnaire d'événement, appelez `Event::off()`. Par exemple :

```
// détache $handler
Event::off('app\interfaces\DanceEventInterface',
DanceEventInterface::EVENT_DANCE, $handler);

// détache tous les gestionnaires de DanceEventInterface::EVENT_DANCE
Event::off('app\interfaces\DanceEventInterface',
DanceEventInterface::EVENT_DANCE);
```

5.3.8 Événements globaux

Yii prend en charge ce qu'on appelle les *événements globaux*, qui est une astuce basée sur le mécanisme des événements décrit ci-dessus. L'événement global requiert un singleton accessible globalement tel que l'instance de l'application elle-même.

Pour créer l'événement global, un émetteur d'événement appelle la méthode `trigger()` du singleton pour déclencher l'événement au lieu d'appeler la méthode `trigger()` propre à l'émetteur. De façon similaire, les gestionnaires d'événement sont attachés à l'événement sur le singleton. Par exemple :

```
use Yii;
use yii\base\Event;
use app\components\Foo;

Yii::$app->on('bar', function ($event) {
    echo get_class($event->sender);  // affiche "app\components\Foo"
});

Yii::$app->trigger('bar', new Event(['sender' => new Foo]));
```

Un avantage de l'utilisation d'événement globaux est que vous n'avez pas besoin d'un objet lorsque vous attachez un gestionnaire à l'événement qui est déclenché par l'objet. Au lieu de cela, vous attachez le gestionnaire et déclenchez l'événement via le singleton (p. ex. l'instance d'application).

Néanmoins, parce que l'espace de noms des événements globaux est partagé par toutes les parties, vous devez nommer les événements globaux avec prudence, par exemple en introduisant une sorte d'espace de noms (p. ex. "frontend.mail.sent", "backend.mail.sent").

5.3.9 Événements génériques

Depuis la version 2.0.14, vous pouvez définir un gestionnaire d'événement pour de multiples événement correspondant à un motif générique. Par exemple :

```
use Yii;

$foo = new Foo();

$foo->on('foo.event.*', function ($event) {
    // déclenché pour tout événement dont le nom commence par 'foo.event.'
    Yii::debug('trigger event: ' . $event->name);
});
```

Les motifs génériques peuvent être utilisés pour des événements au niveau de la classe. Par exemple :

```
use yii\base\Event;
use Yii;

Event::on('app\models\*', 'before*', function ($event) {
    // déclenché pour toute classe de l'espace de noms  'app\models' pour
    tout événement dont le nom commence par 'before'
    Yii::debug('trigger event: ' . $event->name . ' for class: ' .
    get_class($event->sender));
});
```

Cela vous permet d'attraper tous les événement de l'application par un unique gestionnaire en utilisant le code suivant :

```
use yii\base\Event;
use Yii;

Event::on('*', '*', function ($event) {
    // déclenché pour tout événement de n'importe quelle classe
    Yii::debug('trigger event: ' . $event->name);
});
```

> **Note :** l'utilisation de motifs génériques pour la définition des gestionnaires d'événement peut réduire la performance de l'application . Il vaut mieux l'éviter si possible.

Afin de détacher un gestionnaire d'événement spécifié par un motif générique, vous devez répéter le même motif en invoquant `yii\base\Component::off()` ou `yii\base\Event::off()`. Soyez conscient que le passage d'un motif générique

lors du détachement d'un gestionnaire d'événement ne détache que le gestionnaire attaché avec ce motif, tandis que les gestionnaires attachés par des noms réguliers d'événement resteront attachés même si leur nom correspond au motif. Par exemple :

```
use Yii;

$foo = new Foo();

// attache un gestionnaire de façon régulière
$foo->on('event.hello', function ($event) {
    echo 'direct-handler'
});

// attache un gestionnaire par un motif générique
$foo->on('*', function ($event) {
    echo 'wildcard-handler'
});

// ne détache que le gestionnaire attaché par le motif générique
$foo->off('*');

$foo->trigger('event.hello'); // outputs: 'direct-handler'
```

5.4 Comportements

Les comportements (*behaviors* sont des instances de la classe `yii\base\Behavior`, ou de ses classes filles. Les comportements, aussi connus sous le nom de mixins [2], vous permettent d'améliorer les fonctionnalités d'une classe de `composant` existante sans avoir à modifier les héritages de cette classe. Le fait d'attacher un comportement à un composant injecte les méthodes et les propriétés de ce comportement dans le composant, rendant ces méthodes et ces propriétés accessibles comme si elles avaient été définies dans la classe du composant lui-même. En outre, un comportement peut répondre aux événements déclenchés par le composant, ce qui permet aux comportements de personnaliser l'exécution normale du code du composant.

5.4.1 Définition des comportements

Pour définir un comportement, vous devez créer une classe qui étend la classe `yii\base\Behavior`, ou une des ses classes filles. Par exemple :

```
namespace app\components;

use yii\base\Behavior;
```

2. https://fr.wikipedia.org/wiki/Mixin

```
class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;

    public function getProp2()
    {
        return $this->_prop2;
    }

    public function setProp2($value)
    {
        $this->_prop2 = $value;
    }

    public function foo()
    {
        // ...
    }
}
```

Le code ci-dessus définit la classe de comportement `app\components\MyBehavior` avec deux propriété — `prop1` et `prop2` — et une méthode `foo()`. Notez que la propriété `prop2` est définie via la méthode d'obtention `getProp2` et la méthode d'assignation `setProp2`. Cela est le cas parce que la classe `yii\base\Behavior` étend la classe `yii\base\BaseObject` et, par conséquent, prend en charge la définition des propriétés via les méthodes d'obtention et d'assignation.

Comme cette classe est un comportement, lorsqu'elle est attachée à un composant, ce composant acquiert alors les propriétés `prop1` et `prop2`, ainsi que la méthode `foo()`.

> **Conseil :** depuis l'intérieur d'un comportement, vous avez accès au composant auquel le comportement est attaché via la propriété `yii\base\Behavior::$owner`.
>
> **Note :** dans le cas où les méthodes `yii\base\Behavior::__get()` et/ou `yii\base\Behavior::__set()` du comportement sont redéfinies, vous devez redéfinir les méthodes `yii\base\Behavior::canGetProperty()` et/ou `yii\base\Behavior::canSetProperty()` également.

5.4.2 Gestion des événements du composant

Si un comportement a besoin de répondre aux événements déclenchés par le composant auquel il est attaché, il doit redéfinir la méthode `yii\base\Behavior::events()`. Par exemple :

```
namespace app\components;

use yii\db\ActiveRecord;
```

```
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
        // ...
    }
}
```

La méthode `events()` doit retourner une liste d'événements avec leur gestionnaire correspondant. L'exemple ci-dessus déclare que l'événement `EVENT_BEFORE_VALIDATE` existe et définit son gestionnaire `beforeValidate()`. En spécifiant un gestionnaire d'événement, vous pouvez utiliser un des formats suivants :

— une chaîne de caractères qui fait référence au nom d'une méthode de la classe du comportement, comme dans l'exemple ci-dessus ;
— un tableau constitué d'un nom d'objet ou de classe et d'un nom de méthode sous forme de chaîne de caractères (sans les parenthèses), p. ex. `[$object, 'methodName']` ;
— une fonction anonyme.

La signature d'un gestionnaire d'événement doit être similaire à ce qui suit, où `event` fait référence au paramètre événement. Reportez-vous à la section Événements pour plus de détail sur les événements.

```
function ($event) {
}
```

5.4.3 Attacher des comportements

Vous pouvez attacher un comportement à un `composant` soit de manière statique, soit de manière dynamique. Le première manière est une pratique plus habituelle.

Pour attacher un comportement de manière statique, redéfinissez la méthode `behaviors()` de la classe du composant auquel le comportement va être attaché. La méthode `behaviors()` doit retourner une liste de configurations de comportements. Chaque comportement peut être soit un nom de classe de comportement, soit un tableau de configuration :

```
namespace app\models;
```

```php
use yii\db\ActiveRecord;
use app\components\MyBehavior;

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // comportement anonyme, nom de la classe de comportement
            seulement
            MyBehavior::class,

            // comportement nommé, nom de classe de comportement seulement
            'myBehavior2' => MyBehavior::class,

            // comportement anonyme, tableau de configuration
            [
                'class' => MyBehavior::class,
                'prop1' => 'value1',
                'prop2' => 'value2',
            ],

            // comportement nommé, tableau de configuration
            'myBehavior4' => [
                'class' => MyBehavior::class,
                'prop1' => 'value1',
                'prop2' => 'value2',
            ]
        ];
    }
}
```

Vous pouvez associer un nom au comportement en spécifiant la clé de tableau correspondant à la configuration du comportement. Dans ce cas, le comportement est appelé *comportement nommé*. Dans l'exemple ci-dessus, il y a deux comportements nommés : `myBehavior2` et `myBehavior4`. Si un comportement n'est pas associé à un nom, il est appelé *comportement anonyme*.

Pour attacher un comportement de manière dynamique, appelez la méthode `yii\base\Component::attachBehavior()` du composant auquel le comportement va être attaché :

```php
use app\components\MyBehavior;

// attache un objet comportement
$component->attachBehavior('myBehavior1', new MyBehavior());

// attache un classe de comportement
$component->attachBehavior('myBehavior2', MyBehavior::class);

// attache un tableau de configuration
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::class,
```

```
    'prop1' => 'value1',
    'prop2' => 'value2',
]);
```

Vous pouvez attacher plusieurs comportements à la fois en utilisant la méthode `yii\base\Component::attachBehaviors()` :

```
$component->attachBehaviors([
    'myBehavior1' => new MyBehavior(), // un comportement nommé
    MyBehavior::class,                 // un comportement anonyme
]);
```

Vous pouvez aussi attacher des comportements via les configurations comme ceci :

```
[
    'as myBehavior2' => MyBehavior::class,

    'as myBehavior3' => [
        'class' => MyBehavior::class,
        'prop1' => 'value1',
        'prop2' => 'value2',
    ],
]
```

Pour plus de détails, reportez-vous à la section Configurations.

5.4.4 Utilisation des comportements

Pour utiliser un comportement, commencez par l'attacher à un `composant` en suivant les instructions données ci-dessus. Une fois le comportement attaché au composant, son utilisation est évidente.

Vous pouvez accéder à une variable membre *publique*, ou à une propriété définie par une méthode d'obtention et/ou une méthode d'assignation (*getter* et *setter*), du comportement, via le composant auquel ce comportement est attaché :

```
// "prop1" est une propriété définie dans la classe du comportement
echo $component->prop1;
$component->prop1 = $value;
```

Vous pouvez aussi appeler une méthode *publique* du comportement de façon similaire :

```
// foo() est une méthode publique définie dans la classe du comportement
$component->foo();
```

Comme vous pouvez le voir, bien que le composant `$component` ne définissent pas `prop1` et`foo()`, elles peuvent être utilisées comme si elles faisaient partie de la définition du composant grâce au comportement attaché.

Si deux comportement définissent la même propriété ou la même méthode, et que ces deux comportement sont attachés au même composant, le comportement qui a été attaché le *premier* prévaut lorsque la propriété ou la méthode est accédée.

Un comportement peut être associé à un nom lorsqu'il est attaché à un composant. Dans un tel cas, vous pouvez accéder à l'objet comportement en utilisant ce nom :

```
$behavior = $component->getBehavior('myBehavior');
```

Vous pouvez aussi obtenir tous les comportements attachés au composant :

```
$behaviors = $component->getBehaviors();
```

5.4.5 Détacher des comportements

Pour détacher un comportement, appelez `yii\base\Component::detachBehavior()` avec le nom associé au comportement :

```
$component->detachBehavior('myBehavior1');
```

Vous pouvez aussi détacher *tous les* comportements :

```
$component->detachBehaviors();
```

5.4.6 Utilisation de

Pour aller à l'essentiel, jetons un coup d'œil à `yii\behaviors\TimestampBehavior`. Ce comportement prend automatiquement en charge la mise à jour de l'attribut *timestamp* (horodate) d'un modèle `enregistrement actif` à chaque fois qu'il est sauvegardé via les méthodes `insert()`, `update()` ou `save()`.

Tout d'abord, attachez ce comportement à la classe `Active Record (enregistrement actif)` que vous envisagez d'utiliser :

```
namespace app\models\User;

use yii\db\ActiveRecord;
use yii\behaviors\TimestampBehavior;

class User extends ActiveRecord
{
    // ...

    public function behaviors()
    {
        return [
            [
                'class' => TimestampBehavior::class,
                'attributes' => [
```

```
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at',
                    'updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
                // si vous utilisez datetime au lieur de l'UNIX timestamp:
                // 'value' => new Expression('NOW()'),
            ],
        ];
    }
}
```

Le comportement ci-dessus spécifie que lorsque l'enregistrement est :

— inséré, le comportement doit assigner l'horodate UNIX courante aux attributs `created_at` (créé le) et `updated_at` (mis à jour le) ;

— mis à jour, le comportement doit assigner l'horodate UNIX courante à l'attribut `updated_at` ;

> **Note :** pour que la mise en œuvre ci-dessus fonctionne avec une base de données MySQL, vous devez déclarer les colonnes (`created_at`, `updated_at`) en tant que `int(11)` pour qu'elles puissent représenter des horodates UNIX.

Avec ce code en place, si vous avez un objet `User` (utilisateur) et que vous essayez de le sauvegarder, il verra ses attributs `created_at` et `updated_at` automatiquement remplis avec l'horodate UNIX :

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
echo $user->created_at;  // affiche l'horodate courante
```

Le comportement `TimestampBehavior` offre également une méthode utile `touch()`, qui assigne l'horodate courante à un attribut spécifié et le sauvegarde dans la base de données :

```
$user->touch('login_time');
```

5.4.7 Autres comportements

Il existe plusieurs comportements pré-inclus et extérieurs disponibles :

— `yii\behaviors\BlameableBehavior` – remplit automatiquement les attributs spécifiés avec l'identifiant de l'utilisateur courant.

— `yii\behaviors\SluggableBehavior` – remplit automatiquement l'attribut spécifié avec une valeur utilisable en tant que chaîne purement ASCII (*slug*) dans une URL.

— `yii\behaviors\AttributeBehavior` – assigne automatiquement une valeur spécifiée à un ou plusieurs attributs d'un objet enregistrement actif lorsque certains événements se produisent.

— yii2tech\ar\softdelete\SoftDeleteBehavior [3] – fournit des méthodes pour une suppression douce et une restauration douce d'un enregistrement actif c.-à-d. positionne un drapeau ou un état qui marque l'enregistrement comme étant effacé.
— yii2tech\ar\position\PositionBehavior [4] – permet la gestion de l'ordre des enregistrements dans un champ entier (*integer*) en fournissant les méthodes de remise dans l'ordre.

5.4.8 Comparaison des comportement et des traits

Bien que les comportements soient similaires aux traits [5] par le fait qu'ils *injectent* tous deux leurs propriétés et leurs méthodes dans la classe primaire, ils diffèrent par de nombreux aspects. Comme nous l'expliquons ci-dessous, ils ont chacun leurs avantages et leurs inconvénients. Ils sont plus des compléments l'un envers l'autre, que des alternatives.

Raisons d'utiliser des comportements

Les classes de comportement, comme les classes normales, prennent en charge l'héritage. Les traits, par contre, peuvent être considérés comme des copier coller pris en charge par le langage. Ils ne prennent pas en charge l'héritage.

Les comportements peuvent être attachés et détachés à un composant de manière dynamique sans qu'une modification de la classe du composant soit nécessaire. Pour utiliser un trait, vous devez modifier le code de la classe qui l'utilise.

Les comportements sont configurables mais les traits ne le sont pas.

Les comportement peuvent personnaliser l'exécution du code d'un composant en répondant à ses événements.

Lorsqu'il se produit des conflits de noms entre les différents comportements attachés à un même composant, les conflits sont automatiquement résolus en donnant priorité au comportement attaché le premier. Les conflits de noms causés par différents traits nécessitent une résolution manuelle en renommant les propriétés et méthodes concernées.

Raisons d'utiliser des traits

Les traits sont beaucoup plus efficaces que les comportements car les comportements sont des objets qui requièrent plus de temps du processeur et plus de mémoire.

Les environnement de développement intégrés (EDI) sont plus conviviaux avec les traits car ces derniers sont des constructions natives du langage.

3. https://github.com/yii2tech/ar-softdelete
4. https://github.com/yii2tech/ar-position
5. https://www.php.net/manual/fr/language.oop5.traits.php

5.5 Configurations

Les configurations sont très largement utilisées dans Yii lors de la création d'objets ou l'initialisation d'objets existants. Les configurations contiennent généralement le nom de la classe de l'objet en cours de création, et une liste de valeurs initiales qui doivent être assignées aux propriétés de l'objet. Elles peuvent aussi comprendre une liste de gestionnaires qui doivent être attachés aux événements de l'objet et/ou une liste de comportements qui doivent être attachés à l'objet.

Dans ce qui suit, une configuration est utilisée pour créer et initialiser une connexion à une base de données :

```
$config = [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];

$db = Yii::createObject($config);
```

La méthode `Yii::createObject()` prend un tableau de configuration en tant qu'argument et crée un objet en instanciant la classe nommée dans la configuration. Lorsque l'objet est instancié, le reste de la configuration est utilisé pour initialiser les propriétés de l'objet, ses gestionnaires d'événement et ses comportements.

Si vous disposez déjà d'un objet, vous pouvez utiliser la méthode `Yii::configure()` pour initialiser les propriétés de l'objet avec un tableau de configuration :

```
Yii::configure($object, $config);
```

Notez bien que dans ce cas, le tableau de configuration ne doit pas contenir d'élément `class`.

5.5.1 Format d'une configuration

Le format d'une configuration peut être formellement décrit comme suit :

```
[
    'class' => 'ClassName',
    'propertyName' => 'propertyValue',
    'on eventName' => $eventHandler,
    'as behaviorName' => $behaviorConfig,
]
```

où

— L'élément `class` spécifie un nom de classe pleinement qualifié pour l'objet à créer.
— L'élément `propertyName` spécifie les valeurs initiales d'une propriété nommé property. Les clés sont les noms de propriété et les valeurs correspondantes les valeurs initiales. Seules les variables membres publiques et les propriétés définies par des méthodes d'obtention (*getters*) et/ou des méthodes d'assignation (*setters*) peuvent être configurées.
— Les éléments `on eventName` spécifient quels gestionnaires doivent être attachés aux événements de l'objet. Notez que les clés du tableau sont formées en préfixant les noms d'événement par `on`. Reportez-vous à la section événements pour connaître les formats des gestionnaires d'événement pris en charge.
— L'élément `as behaviorName` spécifie quels comportements doivent être attachés à l'objet. Notez que les clés du tableau sont formées en préfixant les noms de comportement par `as` ; la valeur `$behaviorConfig` représente la configuration pour la création du comportement, comme une configuration normale décrite ici.

Ci-dessous, nous présentons un exemple montrant une configuration avec des valeurs initiales de propriétés, des gestionnaires d'événement et des comportements.

```
[
    'class' => 'app\components\SearchEngine',
    'apiKey' => 'xxxxxxxx',
    'on search' => function ($event) {
        Yii::info("Keyword searched: " . $event->keyword);
    },
    'as indexer' => [
        'class' => 'app\components\IndexerBehavior',
        // ... property init values ...
    ],
]
```

5.5.2 Utilisation des configurations

Les configurations sont utilisées en de nombreux endroits dans Yii. Au début de cette section, nous avons montré comment créer un objet obéissant à une configuration en utilisant la méthode `Yii::createObject()`. Dans cette sous-section, nous allons décrire les configurations d'applications et les configurations d'objets graphiques (*widget*) – deux utilisations majeures des configurations.

Configurations d'applications

La configuration d'une application est probablement l'un des tableaux les plus complexes dans Yii. Cela est dû au fait que la classe `application` dispose d'un grand nombre de propriétés et événements configurables. De

première importance, se trouve sa propriété `components` qui peut recevoir un tableau de configurations pour créer des composants qui sont enregistrés durant l'exécution de l'application. Ce qui suit est un résumé de la configuration de l'application du modèle de projet *basic*.

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
        'log' => [
            'class' => 'yii\log\Dispatcher',
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                ],
            ],
        ],
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=stay2',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
];
```

La configuration n'a pas de clé `class`. Cela tient au fait qu'elle est utilisée comme indiqué ci-dessous dans un script d'entrée, dans lequel le nom de la classe est déjà donné :

```
(new yii\web\Application($config))->run();
```

Plus de détails sur la configuration de la propriété `components` d'une application sont donnés dans la section Applications et dans la section Localisateur de services.

Depuis la version 2.0.11, la configuration de l'application prend en charge la configuration du Conteneur d'injection de dépendances via la propriété `container`. Par exemple :

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
```

```
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'container' => [
        'definitions' => [
            'yii\widgets\LinkPager' => ['maxButtonCount' => 5]
        ],
        'singletons' => [
            // Configuration du singleton Dependency Injection Container
        ]
    ]
];
```

Pour en savoir plus sur les valeurs possibles des tableaux de configuration de `definitions` et `singletons` et avoir des exemples de la vie réelle, reportez-vous à la sous-section Utilisation pratique avancée de l'article Conteneur d'injection de dépendances.

Configurations des objets graphiques

Lorsque vous utilisez des objets graphiques, vous avez souvent besoin d'utiliser des configurations pour personnaliser les propriétés de ces objets graphiques. Les méthodes `yii\base\Widget::widget()` et `yii\base\Widget::begin()` peuvent toutes deux être utilisées pour créer un objet graphique. Elles acceptent un tableau de configuration, comme celui qui suit :

```
use yii\widgets\Menu;

echo Menu::widget([
    'activateItems' => false,
    'items' => [
        ['label' => 'Home', 'url' => ['site/index']],
        ['label' => 'Products', 'url' => ['product/index']],
        ['label' => 'Login', 'url' => ['site/login'], 'visible' =>
        Yii::$app->user->isGuest],
    ],
]);
```

La configuration ci-dessus crée un objet graphique nommé `Menu` et initialise sa propriété `activateItems` à `false` (faux). La propriété `items` est également configurée avec les items de menu à afficher.

Notez que, comme le nom de classe est déjà donné, le tableau de configuration ne doit PAS contenir de clé `class`.

5.5.3 Fichiers de configuration

Lorsqu'une configuration est très complexe, une pratique courante est de la stocker dans un ou plusieurs fichiers PHP appelés *fichiers de configuration*. Un fichier de configuration retourne un tableau PHP représentant la configuration. Par exemple, vous pouvez conserver une configuration d'application dans un fichier nommé `web.php`, comme celui qui suit :

```
return [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'components' => require __DIR__ . '/components.php',
];
```

Parce que la configuration `components` et elle aussi complexe, vous pouvez la stocker dans un fichier séparé appelé `components.php` et requérir ce fichier dans `web.php` comme c'est montré ci-dessus. Le contenu de `components.php` ressemble à ceci :

```
return [
    'cache' => [
        'class' => 'yii\caching\FileCache',
    ],
    'mailer' => [
        'class' => 'yii\swiftmailer\Mailer',
    ],
    'log' => [
        'class' => 'yii\log\Dispatcher',
        'traceLevel' => YII_DEBUG ? 3 : 0,
        'targets' => [
            [
                'class' => 'yii\log\FileTarget',
            ],
        ],
    ],
    'db' => [
        'class' => 'yii\db\Connection',
        'dsn' => 'mysql:host=localhost;dbname=stay2',
        'username' => 'root',
        'password' => '',
        'charset' => 'utf8',
    ],
];
```

Pour obtenir une configuration stockée dans un fichier de configuration, il vous suffit requérir ce fichier avec "require", comme ceci :

```
$config = require 'path/to/web.php';
(new yii\web\Application($config))->run();
```

5.5.4 Configurations par défaut

La méthode `Yii::createObject()` est implémentée sur la base du conteneur d'injection de dépendances. Cela vous permet de spécifier un jeu de configurations dites *configurations par défaut* qui seront appliquées à TOUTES les instances des classes spécifiées lors de leur création en utilisant `Yii::createObject()`. Les configurations par défaut peuvent être spécifiées en appelant `Yii::$container->set()` dans le code d'amorçage.

Par exemple, si vous voulez personnaliser l'objet graphique `yii\widgets\LinkPager` de façon à ce que TOUS les fonctions de mise en page (pagers) affichent au plus 5 boutons de page (la valeur par défaut est 10), vous pouvez utiliser le code suivant pour atteindre ce but :

```
\Yii::$container->set('yii\widgets\LinkPager', [
    'maxButtonCount' => 5,
]);
```

Sans les configurations par défaut, vous devez configurer la propriété `maxButtonCount` partout où vous utilisez un pagineur.

5.5.5 Constantes d'environment

Les configurations varient souvent en fonction de l'environnement dans lequel les applications s'exécutent. Par exemple, dans l'environnement de développement, vous désirez peut être utiliser la base de données nommée `mydb_dev`, tandis que sur un serveur en production, vous désirez utiliser la base de données nommée `mydb_prod`. Pour faciliter le changement d'environnement, Yii fournit une constante nommée `YII_ENV` que vous pouvez définir dans le script d'entrée de votre application. Par exemple :

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

Vous pouvez assigner à `YII_ENV` une des valeurs suivantes :

- `prod` : environnement de production. La constante `YII_ENV_PROD` est évaluée comme étant `true` (vrai). C'est la valeur par défaut de `YII_ENV`.
- `dev` : environnement de développement. La constante `YII_ENV_DEV` est évaluée comme étant `true` (vrai).
- `test` : environnement de test. La constante `YII_ENV_TEST` est évaluée comme étant `true` (vrai).

Avec ces constantes d'environnement, vous pouvez spécifier les configurations en fonction de l'environnement courant. Par exemple, votre configuration d'application peut contenir le code suivant pour activer la barre de débogage et le module de débogage dans l'environnement de développement seulement :

```
$config = [...];

if (YII_ENV_DEV) {
    // ajustement de la configuration pour l'environnement 'dev'
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
}

return $config;
```

5.6 Alias

Les alias sont utilisés pour représenter des chemins de fichier ou des URL de façon à ce que vous n'ayez pas besoin d'écrire ces chemins ou ces URL en entier dans votre code. Un alias doit commencer par le caractère arobase `@` pour être différentié des chemins de fichier et des URL normaux. Les alias définis sans ce caractère de tête `@` sont automatiquement préfixés avec ce dernier.

Yii possèdent de nombreux alias pré-définis déjà disponibles. Par exemple, l'alias `@yii` représente le chemin d'installation de la base structurée de développement PHP (*framework*), Yii ; L'alias `@web` représente l'URL de base de l'application Web en cours d'exécution.

5.6.1 Définition des alias

Vous pouvez définir un alias pour un chemin de fichier ou pour une URL en appelant `Yii::setAlias()` :

```
// un alias pour un chemin de fichier
Yii::setAlias('@foo', '/path/to/foo');

// un alias pour une URL
Yii::setAlias('@bar', 'https://www.example.com');


// un alias de fichier concrêt qui contient une classe  \foo\Bar
Yii::setAlias('@foo/Bar.php', '/definitely/not/foo/Bar.php');
```

> **Note :** le chemin de fichier ou l'URL pour qui un alias est créé peut *ne pas* nécessairement faire référence à un fichier ou une ressource existante.

Étant donné un alias, vous pouvez dériver un autre alias – sans faire appel à `Yii::setAlias()`) – en y ajoutant une barre oblique de division `/` suivi d'un ou plusieurs segments de chemin. Les alias définis via `Yii::setAlias()` sont des *alias racines*, tandis que les alias qui en dérivent sont des *alias dérivés*. Par exemple, `@foo` est un alias racine, alors que `@foo/bar/file.php` est un alias dérivé.

Vous pouvez définir un alias en utilisant un autre alias (qu'il soit racine ou dérivé) :

```
Yii::setAlias('@foobar', '@foo/bar');
```

Les alias racines sont ordinairement définis pendant l'étape d'amorçage. Par exemple, vous pouvez appeler `Yii::setAlias()` dans le script d'entrée. Pour commodité, la classe Application fournit une propriété nommée `aliases` que vous pouvez configurer dans la configuration de l'application :

```
return [
    // ...
    'aliases' => [
        '@foo' => '/path/to/foo',
        '@bar' => 'https://www.example.com',
    ],
];
```

5.6.2 Résolution des alias

Vous pouvez appeler `Yii::getAlias()` pour résoudre un alias racine en le chemin de fichier ou l'URL qu'il représente. La même méthode peut aussi résoudre un alias dérivé en le chemin de fichier ou l'URL correspondant :

```
echo Yii::getAlias('@foo');               // affiche : /path/to/foo
echo Yii::getAlias('@bar');               // affiche :
https://www.example.com
echo Yii::getAlias('@foo/bar/file.php'); // affiche :
/path/to/foo/bar/file.php
```

Le chemin ou l'URL que représente un alias dérivé est déterminé en remplaçant l'alias racine par le chemin ou l'URL qui lui correspond dans l'alias dérivé.

> **Note :** la méthode `Yii::getAlias()` ne vérifie pas que le chemin ou l'URL qui en résulte fait référence à un fichier existant ou à une ressource existante.

Un alias racine peut également contenir des barres obliques de division `/`. La méthode `Yii::getAlias()` est suffisamment intelligente pour dire quelle partie d'un alias est un alias racine et, par conséquent, déterminer correctement le chemin de fichier ou l'URL qui correspond :

```
Yii::setAlias('@foo', '/path/to/foo');
Yii::setAlias('@foo/bar', '/path2/bar');
Yii::getAlias('@foo/test/file.php');  // affiche :
/path/to/foo/test/file.php
Yii::getAlias('@foo/bar/file.php');   // affiche : /path2/bar/file.php
```

Si `@foo/bar` n'est pas défini en tant qu'alias racine, la dernière instruction affiche `/path/to/foo/bar/file.php`.

5.6.3 Utilisation des alias

Les alias sont reconnus en différents endroits dans Yii sans avoir besoin d'appeler `Yii::getAlias()` pour les convertir en chemin ou URL. Par exemple, `yii\caching\FileCache::$cachePath` accepte soit un chemin de fichier, soit un alias représentant un chemin de fichier, grâce au préfixe `@` qui permet de différentier un chemin de fichier d'un alias.

```php
use yii\caching\FileCache;

$cache = new FileCache([
    'cachePath' => '@runtime/cache',
]);
```

Reportez-vous à la documentation de l'API pour savoir si une propriété ou une méthode prend en charge les alias.

5.6.4 Alias prédéfinis

Yii prédéfinit un jeu d'alias pour faire référence à des chemins de fichier ou à des URL d'utilisation courante :

- `@yii`, le dossier où le fichier `BaseYii.php` se trouve – aussi appelé dossier de la base structurée de développement PHP (*framework*).
- `@app`, le `chemin de base` de l'application en cours d'exécution.
- `@runtime`, le `chemin du dossier runtime` de l'application en cours d'exécution. Valeur par défaut `@app/runtime`.
- `@webroot`, le dossier Web racine de l'application en cours d'exécution. Il est déterminé en se basant sur le dossier qui contient le script d'entrée.
- `@web`, l'URL de base de l'application en cours d'exécution. Cet alias a la même valeur que `yii\web\Request::$baseUrl`.
- `@vendor`, le `dossier vendor de Composer`. Valeur par défaut `@app/vendor`.
- `@bower`, le dossier racine des paquets bower [6]. Valeur par défaut `@vendor/bower`.
- `@npm`, le dossier racine des paquets npm [7]. Valeur par défaut `@vendor/npm`.

L'alias `@yii` est défini lorsque vous incluez le fichier `Yii.php` dans votre script d'entrée. Les alias restants sont définis dans le constructeur de l'application au moment où la configuration de l'application est appliquée. .

5.6.5 Alias d'extension

Un alias est automatiquement défini par chacune des extensions qui sont installées par Composer. Chaque alias est nommé d'après le nom de l'extension déclaré dans le fichier `composer.json`. Chaque alias représente le dossier racine du paquet. Par exemple, si vous installez l'extension `yiisoft/yii2-jui`, vous obtiendrez automatiquement l'alias `@yii/jui` défini durant l'étape d'amorçage, et équivalent à :

```php
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

6. https://bower.io/
7. https://www.npmjs.com/

5.7 Chargement automatique des classes

Yii compte sur le mécanisme de chargement automatique des classes [8] pour localiser et inclure tous les fichiers de classes requis. Il fournit un chargeur automatique de classes de haute performance qui est conforme à la norme PSR-4 [9]. Le chargeur automatique est installé lorsque vous incluez le fichier `Yii.php`.

> **Note :** pour simplifier la description, dans cette section, nous ne parlerons que du chargement automatique des classes. Néanmoins, gardez présent à l'esprit que le contenu que nous décrivons ici s'applique aussi au chargement automatique des interfaces et des traits.

5.7.1 Utilisation du chargeur automatique de Yii

Pour utiliser le chargeur automatique de classes de Yii, vous devez suivre deux règles simples lorsque vous créez et nommez vos classes :

— Chaque classe doit être placée sous un espace de noms [10] (p. ex. `foo\bar\MyClass`)
— Chaque classe doit être sauvegardée sous forme d'un fichier individuel dont le chemin est déterminé par l'algorithme suivant :

```
// $className est un nom de classe pleinement qualifié sans la barre oblique
inversée de tête
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) .
'.php');
```

For exemple, si le nom de classe et l'espace de noms sont `foo\bar\MyClass`, l'alias pour le chemin du fichier de classe correspondant est `@foo/bar/MyClass.php`. Pour que cet alias puisse être résolu en un chemin de fichier, soit `@foo`, soit `@foo/bar` doit être un alias racine.

Lorsque vous utilisez le modèle de projet *basic*, vous pouvez placer vos classes sous l'espace de noms de niveau le plus haut `app` afin qu'elles puissent être chargées automatiquement par Yii sans avoir besoin de définir un nouvel alias. Cela est dû au fait que `@app` est un alias prédéfini, et qu'un nom de classe comme `app\components\MyClass` peut être résolu en le fichier de classe `AppBasePath/components/MyClass.php`, en appliquant l'algorithme précédemment décrit.

Dans le modèle de projet avancé [11], chaque niveau possède son propre alias. Par exemple, le niveau « interface utilisateur » a l'alias `@frontend`, tandis que le niveau « interface d'administration » a l'alias `@backend`. En

8. https://www.php.net/manual/fr/language.oop5.autoload.php
9. https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md
10. https://www.php.net/manual/fr/language.namespaces.php
11. https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md

conséquence, vous pouvez mettre les classes de l'interface utilisateur sous l'espace de noms `frontend`, tandis que les classes de l'interface d'administration sont sous l'espace de noms `backend`. Cela permet à ces classes d'être chargées automatiquement par le chargeur automatique de Yii.

Pour ajouter un espace de noms personnalisé au chargeur automatique, vous devez définir un alias pour le dossier de base de l'espace de noms en utilisant `Yii::setAlias()`. Par exemple, pour charger des classes de l'espace de noms `foo` qui se trouvent dans le dossier `path/to/foo`, vous appelez `Yii::setAlias('@foo', 'path/to/foo')`.

5.7.2 Table de mise en correspondance des classes

Le chargeur automatique de classes de Yii prend en charge la fonctionnalité *table de mise en correspondance des classes*, qui met en correspondance les noms de classe avec les chemins de classe de fichiers. Lorsque le chargeur automatique charge une classe, il commence par chercher si la classe existe dans la table de mise en correspondance. Si c'est le cas, le chemin de fichier correspondant est inclus directement sans plus de recherche. Cela rend le chargement des classes très rapide. En fait, toutes les classes du noyau de Yii sont chargées de cette manière.

Vous pouvez ajouter une classe à la table de mise en correspondance des classes, stockée dans `Yii::$classMap`, avec l'instruction :

```
Yii::$classMap['foo\bar\MyClass'] = 'path/to/MyClass.php';
```

Les alias peuvent être utilisés pour spécifier des chemins de fichier de classe. Vous devez définir la table de mise en correspondance dans le processus d'amorçage afin qu'elle soit prête avant l'utilisation de vos classes.

5.7.3 Utilisation d'autres chargeurs automatiques

Comme Yii utilise Composer comme gestionnaire de dépendances de paquets, il est recommandé que vous installiez aussi le chargeur automatique de Composer. Si vous utilisez des bibliothèques de tierces parties qui ont besoin de leurs propres chargeurs, vous devez installer ces chargeurs également.

Lors de l'utilisation conjointe du chargeur automatique de Yii et d'autres chargeurs automatiques, vous devez inclure le fichier `Yii.php` *après* que tous les autres chargeurs automatiques sont installés. Cela fait du chargeur automatique de Yii le premier à répondre à une requête de chargement automatique de classe. Par exemple, le code suivant est extrait du script d'entrée du modèle de projet *basic*. La première ligne installe le chargeur automatique de Composer, tandis que la seconde installe le chargeur automatique de Yii :

```
require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';
```

Vous pouvez utiliser le chargeur automatique de Composer seul sans celui de Yii. Néanmoins, en faisant de cette manière, la performance de chargement de vos classes est dégradée et vous devez appliquer les règles de Composer pour que vos classes puissent être chargées automatiquement.

> **Info :** si vous voulez ne pas utiliser le chargeur automatique de Yii, vous devez créer votre propre version du fichier `Yii.php` et l'inclure dans votre script d'entrée.

5.7.4 Chargement automatique des classes d'extension

Le chargeur automatique de Yii est capable de charger automatiquement des classes d'extension. La seule exigence est que cette extension spécifie la section `autoload` correctement dans son fichier `composer.json`. Reportez-vous à la documentation de Composer [12] pour plus de détails sur la manière de spécifier `autoload`.

Dans le cas où vous n'utilisez pas le chargeur automatique de Yii, le chargeur automatique de Composer peut toujours charger les classes d'extensions pour vous.

5.8 Localisateur de services

Un localisateur de services est un objet que sait comment fournir toutes sortes de services (ou composants) dont une application peut avoir besoin. Dans le localisateur de services, chaque composant existe seulement sous forme d'une unique instance, identifiée de manière unique par un identifiant. Vous utilisez l'identifiant pour retrouver un composant du localisateur de services.

Dans Yii, un localisateur de service est simplement une instance de `yii\di\ServiceLocator` ou d'une de ses classes filles.

Le localisateur de service le plus couramment utilisé dans Yii est l'objet *application*, auquel vous avez accès via `\Yii::$app`. Les services qu'il procure, tels les composants `request`, `response` et `urlManager`, sont appelés *composants d'application*. Vous pouvez configurer ces trois composants, ou même les remplacer facilement avec votre propre implémentation, en utilisant les fonctionnalités procurées par le localisateur de services.

En plus de l'objet application, chaque objet module est aussi un localisateur de services.

Pour utiliser un localisateur de service, la première étape est d'enregistrer le composant auprès de lui. Un composant peut être enregistré via la méthode `yii\di\ServiceLocator::set()`. Le code suivant montre différentes manières d'enregistrer des composants :

12. https://getcomposer.org/doc/04-schema.md#autoload

```php
use yii\di\ServiceLocator;
use yii\caching\FileCache;

$locator = new ServiceLocator;

// enregistre "cache" en utilisant un nom de classe qui peut être utilisé
pour créer un composant
$locator->set('cache', 'yii\caching\ApcCache');

// enregistre "db" en utilisant un tableau de configuration qui peut être
utilisé pour créer un composant
$locator->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=demo',
    'username' => 'root',
    'password' => '',
]);

// enregistre "search" en utilisant une fonction anonyme qui construit un
composant
$locator->set('search', function () {
    return new app\components\SolrService;
});

// enregistre "pageCache" en utilisant un composant
$locator->set('pageCache', new FileCache);
```

Une fois qu'un composant a été enregistré, vous pouvez y accéder via son identifiant, d'une des deux manières suivantes :

```php
$cache = $locator->get('cache');
// ou en alternative
$cache = $locator->cache;
```

Comme montré ci-dessus, `yii\di\ServiceLocator` vous permet d'accéder à un composant comme à une propriété en utilisant l'identifiant du composant.

Lorsque vous accédez à un composant pour la première fois, `yii\di\ServiceLocator` utilise l'information d'enregistrement du composant pour créer une nouvelle instance du composant et la retourner. Par la suite, si on accède à nouveau au composant, le localisateur de service retourne la même instance.

Vous pouvez utiliser `yii\di\ServiceLocator::has()` pour savoir si un identifiant de composant a déjà été enregistré. Si vous appelez `yii\di\ServiceLocator::get()` avec un identifiant invalide, une exception est levée.

Comme les localisateurs de services sont souvent créés avec des configurations, une propriété accessible en écriture, et nommée `components`, est fournie. Cela vous permet de configurer et d'enregistrer plusieurs composants à la fois. Le code suivant montre un tableau de configuration qui peut être utilisé pour configurer un localisateur de services (p. ex. une application) avec les composants `db`, `cache`, `tz` et `search` :

```
return [
    // ...
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],
        'cache' => 'yii\caching\ApcCache',
        'tz' => function() {
            return new \DateTimeZone(Yii::$app->formatter->defaultTimeZone);
        },
        'search' => function () {
            $solr = new app\components\SolrService('127.0.0.1');
            // ... other initializations ...
            return $solr;
        },
    ],
];
```

Dans ce qui précède, il y a une façon alternative de configurer le composant `search`. Au lieu d'écrire directement une fonction de rappel PHP qui construit une instance de `SolrService`, vous pouvez utiliser une méthode de classe statique pour retourner une telle fonction de rappel, comme c'est montré ci-dessous :

```
class SolrServiceBuilder
{
    public static function build($ip)
    {
        return function () use ($ip) {
            $solr = new app\components\SolrService($ip);
            // ... autres initialisations ...
            return $solr;
        };
    }
}

return [
    // ...
    'components' => [
        // ...
        'search' => SolrServiceBuilder::build('127.0.0.1'),
    ],
];
```

Cette approche alternative est à utiliser de préférence lorsque vous publiez une composant Yii qui encapsule quelques bibliothèques de tierces parties. Vous utilisez la méthode statique comme c'est montré ci-dessus pour représenter la logique complexe de construction de l'objet de tierce partie, et

l'utilisateur de votre composant doit seulement appeler la méthode statique pour configurer le composant.

5.8.1 Parcours d'un arbre

Les modules acceptent les inclusions arbitraires ; une application Yii est essentiellement un arbre de modules. Comme chacun de ces modules est un localisateur de services, cela a du sens pour les enfants d'accéder à leur parent. Cela permet aux modules d'utiliser `$this->get('db')` au lieu de faire référence au localisateur de services racine `Yii::$app->get('db')`. Un bénéficie supplémentaire pour le développeur est de pouvoir redéfinir la configuration dans un module.

Toute requête d'un service à l'intérieur d'un module est passée à son parent dans le cas où le module lui-même est incapable de la satisfaire.

Notez que la configuration depuis des composants dans un module n'est jamais fusionnée avec celle depuis un composant du module parent. Le modèle de localisateur de services nous permet de définir des services nommés mais on ne peut supposer que des services du même nom utilisent les mêmes paramètres de configuration.

5.9 Conteneur d'injection de dépendances

Un conteneur d'injection de dépendances (DI container) est un objet qui sait comment instancier et configurer des objets et tous leurs objets dépendants. Cet article de Martin Fowler [13] explique très bien en quoi un conteneur d'injection de dépendances est utile. Ici nous expliquons essentiellement l'utilisation qui est faite du conteneur d'injection de dépendances que fournit Yii.

5.9.1 Injection de dépendances

Yii fournit la fonctionnalité conteneur d'injection de dépendances via la classe `yii\di\Container`. Elle prend en charge les sortes d'injections de dépendance suivantes :

— Injection par le constructeur ;
— Injection par les méthodes ;
— Injection par les méthodes d'assignation et les propriétés ;
— Injection par une méthode de rappel PHP ;

Injection par le constructeur

Le conteneur d'injection de dépendances prend en charge l'injection dans le constructeur grâce à l'allusion au type pour les paramètres du construc-

13. https://martinfowler.com/articles/injection.html

teur. L'allusion au type indique au conteneur de quelles classes ou de quelles interfaces dépend l'objet concerné par la construction. Le conteneur essaye de trouver les instances des classes dont l'objet dépend pour les injecter dans le nouvel objet via le constructeur. Par exemple :

```
class Foo
{
    public function __construct(Bar $bar)
    {
    }
}

$foo = $container->get('Foo');
// qui est équivalent à ce qui suit
$bar = new Bar;
$foo = new Foo($bar);
```

Injection par les méthodes

Ordinairement, les classes dont une classe dépend sont passées à son constructeur et sont disponibles dans la classe durant tout son cycle de vie. Avec l'injection par les méthodes, il est possible de fournir une classe qui est seulement nécessaire à une unique méthode de la classe, et qu'il est impossible de passer au constructeur, ou qui pourrait entraîner trop de surplus de travail dans la majorité des classes qui l'utilisent.

Une méthode de classe peut être définie comme la méthode `doSomething()` de l'exemple suivant :

```
class MyClass extends \yii\base\Component
{
    public function __construct(/*ici, quelques classes légères dont la
    classe dépend*/, $config = [])
    {
        // ...
    }

    public function doSomething($param1, \ma\dependance\Lourde $something)
    {
        // faire quelque chose avec $something
    }
}
```

Vous pouvez appeler la méthode, soit en passant une instance de `\ma\dependance\Lourde` vous-même, soit en utilisant `yii\di\Container::invoke()` comme ceci :

```
$obj = new MyClass(/*...*/);
Yii::$container->invoke([$obj, 'doSomething'], ['param1' => 42]); //
$something est fournie par le conteneur d'injection de dépendances
```

Injection par les méthodes d'assignation et les propriétés

L'injection par les méthodes d'assignation et les propriétés est prise en charge via les configurations. Lors de l'enregistrement d'une dépendance ou lors de la création d'un nouvel objet, vous pouvez fournir une configuration qui est utilisée par le conteneur pour injecter les dépendances via les méthodes d'assignation ou les propriétés correspondantes. Par exemple :

```
use yii\base\BaseObject;

class Foo extends BaseObject
{
    public $bar;

    private $_qux;

    public function getQux()
    {
        return $this->_qux;
    }

    public function setQux(Qux $qux)
    {
        $this->_qux = $qux;
    }
}

$container->get('Foo', [], [
    'bar' => $container->get('Bar'),
    'qux' => $container->get('Qux'),
]);
```

> **Info :** la méthode `yii\di\Container::get()` accepte un tableau de configurations qui peut être appliqué à l'objet en création comme troisième paramètre. Si la classe implémente l'interface `yii\base\Configurable` (p. ex. `yii\base\BaseObject`), le tableau de configuration est passé en tant que dernier paramètre du constructeur de la classe ; autrement le tableau de configuration serait appliqué *après* la création de l'objet.

Injection par une méthode de rappel PHP

Dans ce cas, le conteneur utilise une fonction de rappel PRP enregistrée pour construire de nouvelles instances d'une classe. À chaque fois que `yii\di\Container::get()` est appelée, la fonction de rappel correspondante est invoquée. Cette fonction de rappel est chargée de la résolution des dépendances et de leur injection appropriée dans les objets nouvellement créés. Par exemple :

```
$container->set('Foo', function ($container, $params, $config) {
    $foo = new Foo(new Bar);
```

```
    // ... autres initialisations ...
    return $foo;
});

$foo = $container->get('Foo');
```

Pour cacher la logique complexe de construction des nouveaux objets, vous pouvez utiliser un méthode de classe statique en tant que fonction de rappel. Par exemple :

```
class FooBuilder
{
    public static function build($container, $params, $config)
    {
        $foo = new Foo(new Bar);
        // ... autres initialisations ...
        return $foo;
    }
}

$container->set('Foo', ['app\helper\FooBuilder', 'build']);

$foo = $container->get('Foo');
```

En procédant de cette manière, la personne qui désire configurer la classe `Foo` n'a plus besoin de savoir comment elle est construite.

5.9.2 Enregistrement des dépendances

Vous pouvez utiliser `yii\di\Container::set()` pour enregistrer les dépendances. L'enregistrement requiert un nom de dépendance et une définition de dépendance. Un nom de dépendance peut être un nom de classe, un nom d'interface, ou un nom d'alias ; et une définition de dépendance peut être une nom de classe, un tableau de configuration, ou une fonction de rappel PHP.

```
$container = new \yii\di\Container;

// enregistre un nom de classe tel quel. Cela peut être sauté.
$container->set('yii\db\Connection');

// enregistre une interface
// Lorsqu'une classe dépend d'une interface, la classe correspondante
// est instanciée en tant qu'objet dépendant
$container->set('yii\mail\MailInterface', 'yii\swiftmailer\Mailer');

// enregistre un nom d'alias. Vous pouvez utiliser $container->get('foo')
// pour créer une instance de Connection
$container->set('foo', 'yii\db\Connection');

// enregistre une classe avec une configuration. La configuration
// est appliquée lorsque la classe est instanciée par  get()
```

```php
$container->set('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// enregistre un nom d'alias avec une configuration de classe
// Dans ce cas, un élément "class" est requis pour spécifier la classe
$container->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// enregistre une fonction de rappel PHP
// La fonction de rappel est exécutée à chaque fois que
$container->get('db') est appelée
$container->set('db', function ($container, $params, $config) {
    return new \yii\db\Connection($config);
});

// enregistre une interface de composant
// $container->get('pageCache') retourne la même instance à chaque fois
qu'elle est appelée
$container->set('pageCache', new FileCache);
```

> **Conseil :** si un nom de dépendance est identique à celui de la définition de dépendance correspondante, vous n'avez pas besoin de l'enregistrer dans le conteneur d'injection de dépendances.

Une dépendance enregistrée via `set()` génère une instance à chaque fois que la dépendance est nécessaire. Vous pouvez utiliser `yii\di\Container::setSingleton()` pour enregistrer une dépendance qui ne génère qu'une seule instance :

```php
$container->setSingleton('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

5.9.3 Résolution des dépendances

Une fois que vous avez enregistré des dépendances, vous pouvez utiliser le conteneur d'injection de dépendances pour créer de nouveau objets, et le conteneur résout automatiquement les dépendances en les instanciant et en les injectant dans les nouveaux objets. Le résolution des dépendances est

récursive, ce qui signifie que si une dépendance a d'autres dépendances, ces dépendances sont aussi résolue automatiquement.

Vous pouvez utiliser `yii\di\Container::get()` soit pour créer, soit pour obtenir une instance d'un objet. La méthode accepte un nom de dépendance qui peut être un nom de classe, un nom d'interface ou un nom d'alias. Le nom de dépendance, peut être enregistré `set()` ou `setSingleton()`. En option, vous pouvez fournir une liste de paramètres du constructeur de la classe et une configuration pour configurer l'objet nouvellement créé. Par exemple :

```
// "db" est un nom d'alias enregistré préalablement
$db = $container->get('db');

// équivalent à : $engine = new \app\components\SearchEngine($apiKey,
$apiSecret, ['type' => 1]);
$engine = $container->get('app\components\SearchEngine', [$apiKey,
$apiSecret], ['type' => 1]);
```

En coulisses, le conteneur d'injection de dépendances ne fait rien de plus que de créer l'objet. Le conteneur inspecte d'abord le constructeur de la classe pour trouver les classes dépendantes ou les noms d'interface et résout ensuite ces dépendances récursivement.

Le code suivant montre un exemple plus sophistiqué. La classe `UserLister` dépend d'un objet implémentant l'interface `UserFinderInterface` ; la classe `UserFinder` implémente cet interface et dépend de l'objet `Connection`. Toutes ces dépendances sont déclarées via l'allusion au type des paramètres du constructeur de la classe. Avec l'enregistrement des dépendances de propriétés, le conteneur d'injection de dépendances est capable de résoudre ces dépendances automatiquement et de créer une nouvelle instance de `UserLister` par un simple appel à `get('userLister')`.

```
namespace app\models;

use yii\base\BaseObject;
use yii\db\Connection;
use yii\di\Container;

interface UserFinderInterface
{
    function findUser();
}

class UserFinder extends BaseObject implements UserFinderInterface
{
    public $db;

    public function __construct(Connection $db, $config = [])
    {
        $this->db = $db;
        parent::__construct($config);
    }
```

```
    public function findUser()
    {
    }
}

class UserLister extends BaseObject
{
    public $finder;

    public function __construct(UserFinderInterface $finder, $config = [])
    {
        $this->finder = $finder;
        parent::__construct($config);
    }
}

$container = new Container;
$container->set('yii\db\Connection', [
    'dsn' => '...',
]);
$container->set('app\models\UserFinderInterface', [
    'class' => 'app\models\UserFinder',
]);
$container->set('userLister', 'app\models\UserLister');

$lister = $container->get('userLister');

// qui est équivalent à :

$db = new \yii\db\Connection(['dsn' => '...']);
$finder = new UserFinder($db);
$lister = new UserLister($finder);
```

5.9.4 Utilisation pratique

Yii crée un conteneur d'injection de dépendances lorsque vous incluez le fichier `Yii.php` dans le script d'entrée de votre application. Le conteneur d'injection de dépendances est accessible via `Yii::$container`. Lorsque vous appelez `Yii::createObject()`, la méthode appelle en réalité la méthode `get()` du conteneur pour créer le nouvel objet. Comme c'est dit plus haut, le conteneur d'injection de dépendances résout automatiquement les dépendances (s'il en existe) et les injecte dans l'objet obtenu. Parce que Yii utilise `Yii::createObject()` dans la plus grande partie du code de son noyau pour créer de nouveaux objets, cela signifie que vous pouvez personnaliser ces objets globalement en utilisant `Yii::$container`.

Par exemple, personnalisons globalement le nombre de boutons de pagination par défaut de l'objet graphique `yii\widgets\LinkPager` :

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

Maintenant, si vous utilisez l'objet graphique dans une vue avec le code suivant, la propriété `maxButtonCount` est initialisée à la valeur 5 au lieu de la valeur par défaut 10 qui est définie dans la classe. `php echo \yii\widgets\LinkPager : :widget() ; `

Vous pouvez encore redéfinir la valeur définie par le conteneur d'injection de dépendances via :

```
echo \yii\widgets\LinkPager::widget(['maxButtonCount' => 20]);
```

> **Conseil :** peu importe de quel type de valeur il s'agit, elle est redéfinie, c'est pourquoi vous devez vous montrer prudent avec les tableaux d'options. Ils ne sont pas fusionnés.

Un autre exemple est de profiter de l'injection automatique par le constructeur du conteneur d'injection de dépendances. Supposons que votre classe de contrôleur dépende de quelques autres objets, comme un service de réservation d'hôtel. Vous pouvez déclarer la dépendance via un paramètre de constructeur et laisser le conteneur d'injection de dépendances la résoudre pour vous.

```
namespace app\controllers;

use yii\web\Controller;
use app\components\BookingInterface;

class HotelController extends Controller
{
    protected $bookingService;

    public function __construct($id, $module, BookingInterface
    $bookingService, $config = [])
    {
        $this->bookingService = $bookingService;
        parent::__construct($id, $module, $config);
    }
}
```

Si vous accédez au contrôleur à partir du navigateur, vous verrez un message d'erreur se plaignant que l'interface `BookingInterface` ne peut pas être instanciée. Cela est dû au fait que vous devez dire au conteneur d'injection de dépendances comment s'y prendre avec cette dépendance :

```
\Yii::$container->set('app\components\BookingInterface',
'app\components\BookingService');
```

Maintenant, si vous accédez à nouveau au contrôleur, une instance de `app\components\BookingService` est créée et injectée en tant que troisième paramètre du constructeur.

5.9.5 Utilisation pratique avancée

Supposons que nous travaillions sur l'API de l'application et ayons :S

— la classe `app\components\Request` qui étende `yii\web\Request` et fournisse une fonctionnalité additionnelle,

— la classe `app\components\Response` qui étende `yii\web\Response` et devrait avoir une propriété `format` définie à `json` à la création,

— des classes `app\storage\FileStorage` et `app\storage\DocumentsReader` qui mettent en œuvre une certaine logique pour travailler sur des documents qui seraient situés dans un dossier :

```
class FileStorage
{
    public function __construct($root) {
        // whatever
    }
}

class DocumentsReader
{
    public function __construct(FileStorage $fs) {
        // whatever
    }
}
```

Il est possible de configurer de multiples définitions à la fois, en passant un tableau de configurations à la méthode `setDefinitions()` ou à la méthode `setSingletons()`. En itérant sur le tableau de configuration, les méthodes appellent `set()` ou `setSingleton()` respectivement pour chacun des items.

Le format du tableau de configurations est :

— `key` : nom de classe, nom d'interface ou alias. La clé est passée à la méthode `set()` comme premier argument `$class`.

— `value` : la définition associée à `$class`. Les valeurs possibles sont décrites dans la documentation `set()` du paramètre `$definition`. Est passé à la méthode `set()` comme deuxième argument `$definition`.

Par exemple, configurons notre conteneur pour répondre aux exigences mentionnées précédemment :

```
$container->setDefinitions([
    'yii\web\Request' => 'app\components\Request',
    'yii\web\Response' => [
        'class' => 'app\components\Response',
        'format' => 'json'
    ],
    'app\storage\DocumentsReader' => function ($container, $params, $config)
    {
        $fs = new app\storage\FileStorage('/var/tempfiles');
        return new app\storage\DocumentsReader($fs);
    }
```

```
]);

$reader = $container->get('app\storage\DocumentsReader');
// Crée un objet DocumentReader avec ses dépendances tel que décrit dans la
configuration.
```

> **Conseil :** le conteneur peut être configuré dans le style déclaratif en utilisant la configuration de l'application depuis la version 2.0.11. Consultez la sous-section Configurations des applications de l'article du guide Configurations.

Tout fonctionne, mais au cas où, nous devons créer une classe `DocumentWriter`, nous devons copier-coller la ligne qui crée un objet `FileStorage`, ce qui n'est pas la manière la plus élégante, c'est évident.

Comme cela est décrit à la sous-section Résolution des dépendances subsection, `set()` et `setSingleton()` peuvent facultativement des paramètres du constructeur de dépendances en tant que troisième argument. Pour définir les paramètres du constructeur, vous pouvez utiliser le format de tableau de configuration suivant :

— `key` : nom de classe, nom d'interface ou alias. La clé est passée à la méthode `set()` comme premier argument `$class`.

— `value` : un tableau de deux éléments. Le premier élément est passé à la méthode `set()` comme deuxième argument `$definition`, le second — comme `$params`.

Modifions notre exemple :

```
$container->setDefinitions([
    'tempFileStorage' => [ // we've created an alias for convenience
        ['class' => 'app\storage\FileStorage'],
        ['/var/tempfiles'] // pourrait être extrait de certains fichiers de
        configuration
    ],
    'app\storage\DocumentsReader' => [
        ['class' => 'app\storage\DocumentsReader'],
        [Instance::of('tempFileStorage')]
    ],
    'app\storage\DocumentsWriter' => [
        ['class' => 'app\storage\DocumentsWriter'],
        [Instance::of('tempFileStorage')]
    ]
]);

$reader = $container->get('app\storage\DocumentsReader');
// Se comporte exactement comme l'exemple précédent
```

Vous noterez la notation `Instance::of('tempFileStorage')`. cela siginifie que le `Container` fournit implicitement une dépendance enregistrée avec le nom de `tempFileStorage` et la passe en tant que premier argument du constructeur of `app\storage\DocumentsWriter`.

> **Note :** `setDefinitions()` and `setSingletons()` methods are available since version 2.0.11.

Une autre étape de l'optimisation de la configuration est d'enregistrer certaines dépendances sous forme de singletons. Une dépendance enregistrée via `set()` est instanciée à chaque fois qu'on en a besoin. Certaines classes ne changent pas l'état au moment de l'exécution, par conséquent elles peuvent être enregistrées sous forme de singletons afin d'augmenter la performance de l'application.

Un bon exemple serait la classe `app\storage\FileStorage`, qui effectue certaines opérations sur le système de fichiers avec une API simple (p. ex. `$fs->read()`, `$fs->write()`). Ces opération ne changent pas l'état interne de la classe, c'est pourquoi nous pouvons créer son instance une seule fois et l'utiliser de multiples fois.

```
$container->setSingletons([
    'tempFileStorage' => [
        ['class' => 'app\storage\FileStorage'],
        ['/var/tempfiles']
    ],
]);

$container->setDefinitions([
    'app\storage\DocumentsReader' => [
        ['class' => 'app\storage\DocumentsReader'],
        [Instance::of('tempFileStorage')]
    ],
    'app\storage\DocumentsWriter' => [
        ['class' => 'app\storage\DocumentsWriter'],
        [Instance::of('tempFileStorage')]
    ]
]);

$reader = $container->get('app\storage\DocumentsReader');
```

5.9.6 À quel moment enregistrer les dépendances

Comme les dépendances sont nécessaires lorsque de nouveaux objets sont créés, leur enregistrement doit être fait aussi tôt que possible. Les pratiques recommandées sont :

— Si vous êtes le développeur d'une application, vous pouvez enregistrer les dépendances dans le script d'entrée de votre application ou dans un script qui est inclus par le script d'entrée.
— Si vous êtes le développeur d'une extension distribuable, vous pouvez enregistrer les dépendances dans la classe d'amorçage de l'extension.

5.9.7 Résumé

L'injection de dépendances et le localisateur de services sont tous deux des modèles de conception populaires qui permettent des construire des logiciels d'une manière faiblement couplée et plus testable. Nous vous recommandons fortement de lire l'article de Martin [14] pour acquérir une compréhension plus profonde de l'injection de dépendances et du localisateur de services.

Yii implémente son localisateur de services par dessus le conteneur d'injection de dépendances. Lorsqu'un localisateur de services essaye de créer une nouvelle instance d'un objet, il appelle le conteneur d'injection de dépendances. Ce dernier résout les dépendances automatiquement comme c'est expliqué plus haut.

14. https://martinfowler.com/articles/injection.html

Chapitre 6

Travailler avec les Bases de Données

6.1 Objets d'accès aux bases de données

Construits au-dessus des objets de bases de données PHP (PDO – PHP Data Objects) [1], les objets d'accès aux bases de données de Yii (DAO – Database Access Objects) fournissent une API orientée objets pour accéder à des bases de données relationnelles. C'est la fondation pour d'autres méthodes d'accès aux bases de données plus avancées qui incluent le constructeur de requêtes (*query builder*) et l'enregistrement actif (*active record*).

Lorsque vous utilisez les objets d'accès aux bases de données de Yii, vous manipulez des requêtes SQL et des tableaux PHP. En conséquence, cela reste le moyen le plus efficace pour accéder aux bases de données. Néanmoins, étant donné que la syntaxe du langage SQL varie selon le type de base de données, l'utilisation des objets d'accès aux bases de données de Yii signifie également que vous avez à faire un travail supplémentaire pour créer une application indifférente au type de base de données.

Dans Yii 2.0, les objets d'accès aux bases de données prennent en charge les bases de données suivantes sans configuration supplémentaire :

— MySQL [2]
— MariaDB [3]
— SQLite [4]
— PostgreSQL [5] : version 8.4 ou plus récente.
— CUBRID [6] : version 9.3 ou plus récente.

1. https://www.php.net/manual/fr/book.pdo.php
2. https://www.mysql.com/
3. https://mariadb.com/
4. https://sqlite.org/
5. https://www.postgresql.org/
6. https://www.cubrid.org/

— Oracle [7]
— MSSQL [8] : version 2008 ou plus récente.

Info : depuis Yii 2.1, la prise en charge des objets d'accès aux bases de données pour CUBRID, Oracle et MSSQL n'est plus fournie en tant que composants du noyau. Cette prise en charge nécessite l'installation d'extensions séparées. Parmi les extensions officielles [9], on trouve yiisoft/yii2-oracle [10] et yiisoft/yii2-mssql [11].

Note : la nouvelle version de pdo_oci pour PHP 7 n'existe pour le moment que sous forme de code source. Suivez les instructions de la communauté [12] pour la compiler ou utilisez la couche d'émulation de PDO [13].

6.1.1 Création de connexions à une base de données

Pour accéder à une base de données, vous devez d'abord vous y connecter en créant une instance de la classe `yii\db\Connection` :

```
$db = new yii\db\Connection([
    'dsn' => 'mysql:host=localhost;dbname=example',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

Comme souvent vous devez accéder à une base de données en plusieurs endroits, une pratique commune est de la configurer en terme de composant d'application comme ci-après :

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=example',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
    // ...
];
```

7. https://www.oracle.com/database/
8. https://www.microsoft.com/en-us/sqlserver/default.aspx
9. https://www.yiiframework.com/extensions/official
10. https://www.yiiframework.com/extension/yiisoft/yii2-oracle
11. https://www.yiiframework.com/extension/yiisoft/yii2-mssql
12. https://github.com/yiisoft/yii2/issues/10975#issuecomment-248479268
13. https://github.com/taq/pdooci

Vous pouvez ensuite accéder à la base de données via l'expression `Yii::$app->db`.

> **Conseil :** vous pouvez configurer plusieurs composants d'application « base de données » si votre application a besoin d'accéder à plusieurs bases de données.

Lorsque vous configurez une connexion à une base de données, vous devez toujours spécifier le nom de sa source de données (DSN – Data Source Name) via la propriété `dsn`. Les formats des noms de source de données varient selon le type de base de données. Reportez-vous au manuel de PHP [14] pour plus de détails. Ci-dessous, nous donnons quelques exemples :

- MySQL, MariaDB : `mysql:host=localhost;dbname=mydatabase`
- SQLite : `sqlite:/path/to/database/file`
- PostgreSQL : `pgsql:host=localhost;port=5432;dbname=mydatabase`
- CUBRID : `cubrid:dbname=demodb;host=localhost;port=33000`
- MS SQL Server (via sqlsrv driver) : `sqlsrv:Server=localhost;Database=mydatabase`
- MS SQL Server (via dblib driver) : `dblib:host=localhost;dbname=mydatabase`
- MS SQL Server (via mssql driver) : `mssql:host=localhost;dbname=mydatabase`
- Oracle : `oci:dbname=//localhost:1521/mydatabase`

Notez que si vous vous connectez à une base de données en utilisant ODBC (Open Database Connectivity), vous devez configurer la propriété `yii\db\Connection::$driverName` afin que Yii connaisse le type réel de base de données. Par exemple :

```
'db' => [
    'class' => 'yii\db\Connection',
    'driverName' => 'mysql',
    'dsn' => 'odbc:Driver={MySQL};Server=localhost;Database=test',
    'username' => 'root',
    'password' => '',
],
```

En plus de la propriété `dsn`, vous devez souvent configurer les propriétés `username (nom d'utilisateur)` et `password (mot de passe)`. Reportez-vous à `yii\db\Connection` pour une liste exhaustive des propriétés configurables.

> **Info :** lorsque vous créez une instance de connexion à une base de données, la connexion réelle à la base de données n'est pas établie tant que vous n'avez pas exécuté la première requête SQL ou appelé la méthode `open()` explicitement.
>
> **Conseil :** parfois, vous désirez effectuer quelques requêtes juste après l'établissement de la connexion à la base de données pour initialiser quelques variables d'environnement (p. ex. pour définir le fuseau horaire ou le jeu de caractères). Vous pouvez le faire

14. https://www.php.net/manual/fr/pdo.construct.php

en enregistrant un gestionnaire d'événement pour l'événement `afterOpen` de la connexion à la base de données. Vous pouvez enregistrer le gestionnaire directement dans la configuration de l'application comme ceci :

```
'db' => [
    // ...
    'on afterOpen' => function($event) {
        // $event->sender refers to the DB connection
        $event->sender->createCommand("SET time_zone =
        'UTC'")->execute();
    }
],
```

6.1.2 Execution de requêtes SQL

Une fois que vous avez une instance de connexion à la base de données, vous pouvez exécuter une requête SQL en suivant les étapes suivantes :

1. Créer une `commande` avec une requête SQL simple ;
2. Lier les paramètres (facultatif) ;
3. Appeler l'une des méthodes d'exécution SQL dans la `commande`.

L'exemple qui suit montre différentes façons d'aller chercher des données dans une base de données :

```
// retourne un jeu de lignes. Chaque ligne est un tableau associatif
(couples clé-valeur) dont les clés sont des noms de colonnes
// retourne un tableau vide si la requête ne retourne aucun résultat
$posts = Yii::$app->db->createCommand('SELECT * FROM post')
            ->queryAll();

// retourne une ligne unique (la première ligne)
// retourne false si la requête ne retourne aucun résultat
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=1')
           ->queryOne();

// retourne une colonne unique (la première colonne)
//retourne un tableau vide si la requête ne retourne aucun résultat
$titles = Yii::$app->db->createCommand('SELECT title FROM post')
             ->queryColumn();

// retourne une valeur scalaire
// retourne false si la requête ne retourne aucun résultat
$count = Yii::$app->db->createCommand('SELECT COUNT(*) FROM post')
             ->queryScalar();
```

Note : pour préserver la précision, les données extraites des bases de données sont toutes représentées sous forme de chaînes de caractères, même si les colonnes sont de type numérique.

Liaison des paramètres

Lorsque vous créez une commande de base de données à partir d'une requête SQL avec des paramètres, vous devriez presque toujours utiliser l'approche de liaison des paramètres pour éviter les attaques par injection SQL. Par exemple :

```
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status')
            ->bindValue(':id', $_GET['id'])
            ->bindValue(':status', 1)
            ->queryOne();
```

Dans l'instruction SQL, vous pouvez incorporer une ou plusieurs valeurs à remplacer pour les paramètres (p. ex. `:id` dans l'exemple ci-dessus). Une valeur à remplacer pour un paramètre doit être une chaîne de caractères commençant par le caractère deux-points `:`. Vous pouvez ensuite appeler l'une des méthodes de liaison de paramètres suivantes pour lier les valeurs de paramètre :

- `bindValue()` : lie une unique valeur de paramètre
- `bindValues()` : lie plusieurs valeurs de paramètre en un seul appel
- `bindParam()` : similaire à `bindValue()` mais prend aussi en charge la liaison de références à des paramètres

L'exemple suivant montre les manières alternatives de lier des paramètres :

```
$params = [':id' => $_GET['id'], ':status' => 1];

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status')
            ->bindValues($params)
            ->queryOne();

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status', $params)
            ->queryOne();
```

La liaison des paramètres est implémentée via des instructions préparées [15]. En plus d'empêcher les attaques par injection SQL, cela peut aussi améliorer la performance en préparant l'instruction SQL une seule fois et l'exécutant de multiples fois avec des paramètres différents. Par exemple :

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id');

$post1 = $command->bindValue(':id', 1)->queryOne();
$post2 = $command->bindValue(':id', 2)->queryOne();
// ...
```

15. https://www.php.net/manual/fr/mysqli.quickstart.prepared-statements.php

Comme la méthode `bindParam()` prend en charge la liaison des paramètres par référence, le code ci-dessus peut aussi être écrit comme suit :

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id')
              ->bindParam(':id', $id);

$id = 1;
$post1 = $command->queryOne();

$id = 2;
$post2 = $command->queryOne();
// ...
```

Notez que vous devez lier la valeur à remplacer à la variable `$id` avant l'exécution, et ensuite changer la valeur de cette variable avant chacune des exécutions subséquentes (cela est souvent réalisé dans une boucle). L'exécution de requêtes de cette façon peut être largement plus efficace que d'exécuter une nouvelle requête pour chacune des valeurs du paramètre).

> **Info :** la liaison de paramètres n'est utilisée qu'en des endroits où les valeurs doivent être insérées dans des chaînes de caractères qui contiennent du langage SQL. Dans de nombreux endroits dans des couches plus abstraites comme le query builder (constructeur de requêtes) et active record (enregistrement actif) vous spécifiez souvent un tableau de valeurs qui est transformé en SQL. À ces endroits, la liaison de paramètres est assurée par Yii en interne. Il n'est donc pas nécessaire de spécifier ces paramètres manuellement.

Exécution de requête sans sélection

Les méthodes `queryXyz()` introduites dans les sections précédentes concernent toutes des requêtes SELECT qui retournent des données de la base de données. Pour les instructions qui ne retournent pas de donnée, vous devez appeler la méthode `yii\db\Command::execute()` à la place. Par exemple :

```
Yii::$app->db->createCommand('UPDATE post SET status=1 WHERE id=1')
   ->execute();
```

La méthode `yii\db\Command::execute()` exécute retourne le nombre de lignes affectées par l'exécution de la requête SQL.

Pour les requeêtes INSERT, UPDATE et DELETE, au lieu d'écrire des instructions SQL simples, vous pouvez appeler les méthodes `insert()`, `update()` ou `delete()`, respectivement, pour construire les instructions SQL correspondantes. Ces méthodes entourent correctement les noms de tables et de colonnes par des marques de citation et lient les paramètres. Par exemple :

```
// INSERT (table name, column values)
Yii::$app->db->createCommand()->insert('user', [
    'name' => 'Sam',
    'age' => 30,
])->execute();

// UPDATE (table name, column values, condition)
Yii::$app->db->createCommand()->update('user', ['status' => 1], 'age >
30')->execute();

// DELETE (table name, condition)
Yii::$app->db->createCommand()->delete('user', 'status = 0')->execute();
```

Vous pouvez aussi appeler `batchInsert()` pour insérer plusieurs lignes en un seul coup, ce qui est bien plus efficace que d'insérer une ligne à la fois :

```
// noms de table, noms de colonne, valeurs de colonne
Yii::$app->db->createCommand()->batchinsère('user', ['name', 'age'], [
    ['Tom', 30],
    ['Jane', 20],
    ['Linda', 25],
])->execute();
```

Une autre méthode utile est `upsert()`. Upsert est une opération atomique qui insère des lignes dans une table de base de données si elles n'existent pas déjà (répondant à une contrainte unique), ou les mets à jour si elles existent :

```
Yii::$app->db->createCommand()->upsert('pages', [
    'name' => 'Front page',
    'url' => 'https://example.com/', // url is unique
    'visits' => 0,
], [
    'visits' => new \yii\db\Expression('visits + 1'),
], $params)->execute();
```

Le code ci-dessus, soit insère un enregistrement pour une nouvelle page, soit incrémente sont compteur de visite automatiquement.

Notez que les méthodes mentionnées ci-dessus ne font que créer les requêtes, vous devez toujours appeler `execute()` pour les exécuter réellement.

6.1.3 Entourage de noms de table et de colonne par des marque de citation

Lorsque l'on écrit du code indifférent au type de base de données, entourer correctement les noms table et de colonne avec des marques de citation (p. ex. guillemets ou simple apostrophe) et souvent un casse-tête parce que les différentes base de données utilisent des règles de marquage de citation différentes. Pour vous affranchir de cette difficulté, vous pouvez utiliser la syntaxe de citation introduite par Yii :

— `[[column name]]` : entourez un nom de colonne qui doit recevoir les marques de citation par des doubles crochets ;
— `{{table name}}` : entourez un nom de table qui doit recevoir les marques de citation par des doubles accolades ;

Les objets d'accès aux base de données de Yii convertissent automatiquement de telles constructions en les noms de colonne ou de table correspondants en utilisant la syntaxe spécifique au système de gestion de la base de données. Par exemple :

```
// exécute cette instruction SQL pour MySQL: SELECT COUNT(`id`) FROM
`employee`
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM
{{employee}}")
            ->queryScalar();
```

Utilisation des préfixes de table

La plupart des noms de table de base de données partagent un préfixe commun. Vous pouvez utiliser la fonctionnalité de gestion du préfixe de noms de table procurée par les objets d'accès aux bases de données de Yii.

Tout d'abord, spécifiez un préfixe de nom de table via la propriété `yii\db\Connection::$tablePrefix` dans la configuration de l'application :

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            // ...
            'tablePrefix' => 'tbl_',
        ],
    ],
];
```

Ensuite dans votre code, à chaque fois que vous faites référence à une table dont le nom commence par ce préfixe, utilisez la syntaxe `{{%table_name}}`. Le caractère pourcentage `%` est automatiquement remplacé par le préfixe que vous avez spécifié dans la configuration de la connexion à la base de données. Par exemple :

```
// exécute cette instruction SQL pour MySQL: SELECT COUNT(`id`) FROM
`tbl_employee`
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM
{{%employee}}")
            ->queryScalar();
```

6.1.4 Réalisation de transactions

Lorsque vous exécutez plusieurs requêtes liées en séquence, il arrive que vous ayez besoin de les envelopper dans une transactions pour garantir

l'intégrité et la cohérence de votre base de données. Si une des requêtes échoue, la base de données est ramenée en arrière dans l'état dans lequel elle se trouvait avant qu'aucune de ces requêtes n'ait été exécutée.

Le code suivant montre une façon typique d'utiliser les transactions :

```
Yii::$app->db->transaction(function($db) {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... exécution des autres instruction SQL ...
});
```

Le code précédent est équivalent à celui qui suit, et qui vous donne plus de contrôle sur le code de gestion des erreurs :

```
$db = Yii::$app->db;
$transaction = $db->beginTransaction();
try {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... exécutions des autres instructions SQL  ...

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

En appelant la méthode `beginTransaction()`, une nouvelle transaction est démarrée. La transaction est représentée sous forme d'objet `yii\db\Transaction` stocké dans la variable `$transaction`. Ensuite, les requêtes à exécuter sont placées dans un bloc `try...catch...`. Si toutes les requêtes réussissent, la méthode `commit()` est appelée pour entériner la transaction. Autrement, si une exception a été levée et capturée, la méthode `rollBack()` est appelée pour défaire les changements faits par les requêtes de la transaction antérieures à celle qui a échoué. `throw $e` est alors à nouveau exécutée comme si l'exception n'avait jamais été capturée, ce qui permet au processus normal de gestion des erreurs de s'en occuper.

> **Note :** dans le code précédent nous avons deux blocs « catch » pour compatibilité avec PHP 5.x et PHP 7.x. `\Exception` met en œuvre l'interface `\Throwable` [16] depuis PHP 7.0, ainsi vous pouvez sauter la partie avec `\Exception` si votre application utilise seulement PHP 7.0 et plus récent.

16. https://www.php.net/manual/fr/class.throwable.php

Spécification de niveaux d'isolation

Yii prend aussi en charge la définition de [niveaux d'isolation] pour vos transactions. Par défaut, lors du démarrage d'une nouvelle transaction, il utilise le niveau d'isolation par défaut défini par votre système de base de données. Vous pouvez redéfinir le niveau d'isolation comme indiqué ci-après :

```
$isolationLevel = \yii\db\Transaction::REPEATABLE_READ;

Yii::$app->db->transaction(function ($db) {
    ....
}, $isolationLevel);

// ou alternativement

$transaction = Yii::$app->db->beginTransaction($isolationLevel);
```

Yii fournit quatre constantes pour les niveaux d'isolation les plus courants :

- `yii\db\Transaction::READ_UNCOMMITTED` – le niveau le plus faible, des lectures sales (*dirty reads*) , des lectures non répétables) (*non-repeatable reads*) et des lectures fantômes (*phantoms*) peuvent se produire.
- `yii\db\Transaction::READ_COMMITTED` – évite les lectures sales.
- `yii\db\Transaction::REPEATABLE_READ` – évite les lectures sales et les lectures non répétables.
- `yii\db\Transaction::SERIALIZABLE` – le niveau le plus élevé, évite tous les problèmes évoqués ci-dessus.

En plus de l'utilisation des constantes présentées ci-dessus pour spécifier un niveau d'isolation, vous pouvez également utiliser des chaînes de caractères avec une syntaxe valide prise en charges par le système de gestion de base de données que vous utilisez. Par exemple, dans PostgreSQL, vous pouvez utiliser `"SERIALIZABLE READ ONLY DEFERRABLE"`.

Notez que quelques systèmes de gestion de base de données autorisent la définition des niveaux d'isolation uniquement au niveau de la connexion tout entière. Toutes les transactions subséquentes auront donc le même niveau d'isolation même si vous n'en spécifiez aucun. En utilisant cette fonctionnalité, vous avez peut-être besoin de spécifier le niveau d'isolation de manière explicite pour éviter les conflits de définition. Au moment d'écrire ces lignes, seules MSSQL et SQLite sont affectées par cette limitation.

> **Note :** SQLite ne prend en charge que deux niveaux d'isolation, c'est pourquoi vous ne pouvez utiliser que `READ UNCOMMITTED` et `SERIALIZABLE`. L'utilisation d'autres niveaux provoque la levée d'une exception.
>
> **Note :** PostgreSQL n'autorise pas la définition du niveau d'isolation tant que la transaction n'a pas démarré, aussi ne pouvez-vous pas spécifier le niveau d'isolation directement en démarrant la

transaction. Dans ce cas, vous devez appeler `yii\db\Transaction::setIsolationLevel()` après que la transaction a démarré.

Imbrication des transactions

Si votre système de gestion de base de données prend en charge Savepoint, vous pouvez imbriquer plusieurs transactions comme montré ci-dessous :

```
Yii::$app->db->transaction(function ($db) {
    //  transaction extérieure

    $db->transaction(function ($db) {
        //  transaction intérieure
    });
});
```

Ou en alternative, `php $db = Yii : :$app->db ; $outerTransaction = $db->beginTransaction() ; try {

```
$db->createCommand($sql1)->execute();

$innerTransaction = $db->beginTransaction();
try {
    $db->createCommand($sql2)->execute();
    $innerTransaction->commit();
} catch (\Exception $e) {
    $innerTransaction->rollBack();
    throw $e;
} catch (\Throwable $e) {
    $innerTransaction->rollBack();
    throw $e;
}

$outerTransaction->commit();
```

} catch (\Exception $e) {

```
$outerTransaction->rollBack();
throw $e;
```

} catch (\Throwable $e) {

```
$outerTransaction->rollBack();
throw $e;
```

} `

6.1.5 Réplication et éclatement lecture-écriture

Beaucoup de systèmes de gestion de bases de données prennent en charge la réplication de la base de données [17] pour obtenir une meilleure disponibilité et des temps de réponse de serveur plus courts. Avec la réplication de la base de données, les données sont répliquées depuis les serveurs dits *serveurs maîtres* vers les serveurs dit *serveurs esclaves*. Toutes les écritures et les mises à jour ont lieu sur les serveurs maîtres, tandis que les lectures ont lieu sur les serveurs esclaves.

Pour tirer parti de la réplication des bases de données et réaliser l'éclatement lecture-écriture, vous pouvez configurer un composant `yii\db\Connection` comme le suivant :

```
[
    'class' => 'yii\db\Connection',

    // configuration pour le maître
    'dsn' => 'dsn pour le serveur maître',
    'username' => 'master',
    'password' => '',

    //  configuration commune pour les esclaves
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // utilise un temps d'attente de connexion plus court
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // liste des configurations d'esclave
    'slaves' => [
        ['dsn' => 'dsn pour le serveur esclave 1'],
        ['dsn' => 'dsn pour le serveur esclave 2'],
        ['dsn' => 'dsn pour le serveur esclave 3'],
        ['dsn' => 'dsn pour le serveur esclave 4'],
    ],
]
```

La configuration ci-dessus spécifie une configuration avec un unique maître et de multiples esclaves. L'un des esclaves est connecté et utilisé pour effectuer des requêtes en lecture, tandis que le maître est utilisé pour effectuer les requêtes en écriture. Un tel éclatement lecture-écriture est accompli automatiquement avec cette configuration. Par exemple :

```
// crée une instance de Connection en utilisant la configuration ci-dessus
Yii::$app->db = Yii::createObject($config);
```

17. https://fr.wikipedia.org/wiki/R%C3%A9plication_(informatique)

```
// effectue une requête auprès d'un des esclaves
$rows = Yii::$app->db->createCommand('SELECT * FROM user LIMIT
10')->queryAll();

// effectue une requête auprès du maître
Yii::$app->db->createCommand("UPDATE user SET username='demo' WHERE
id=1")->execute();
```

> **Info :** les requêtes effectuées en appelant `yii\db\Command::execute()` sont considérées comme des requêtes en écriture, tandis que toutes les autres requêtes faites via l'une des méthodes « *query* » sont des requêtes en lecture. Vous pouvez obtenir la connexion couramment active à un des esclaves via `Yii::$app->db->slave`.

Le composant `Connection` prend en charge l'équilibrage de charge et de basculement entre esclaves. Lorsque vous effectuez une requête en lecture par la première fois, le composant `Connection` sélectionne un esclave de façon aléatoire et essaye de s'y connecter. Si l'esclave set trouvé « mort », il en essaye un autre. Si aucun des esclaves n'est disponible, il se connecte au maître. En configurant un `cache d'état du serveur`, le composant mémorise le serveur « mort » et ainsi, pendant un `certain intervalle de temps`, n'essaye plus de s'y connecter.

> **Info :** dans la configuration précédente, un temps d'attente de connexion de 10 secondes est spécifié pour chacun des esclaves. Cela signifie que, si un esclave ne peut être atteint pendant ces 10 secondes, il est considéré comme « mort ». Vous pouvez ajuster ce paramètre en fonction de votre environnement réel.

Vous pouvez aussi configurer plusieurs maîtres avec plusieurs esclaves. Par exemple :

```
[
    'class' => 'yii\db\Connection',

    // configuration commune pour les maîtres
    'masterConfig' => [
        'username' => 'master',
        'password' => '',
        'attributes' => [
            // utilise un temps d'attente de connexion plus court
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // liste des configurations de maître
    'masters' => [
        ['dsn' => 'dsn for master server 1'],
        ['dsn' => 'dsn for master server 2'],
    ],
```

```
    // configuration commune pour les esclaves
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // liste des configurations d'esclave
    'slaves' => [
        ['dsn' => 'dsn for slave server 1'],
        ['dsn' => 'dsn for slave server 2'],
        ['dsn' => 'dsn for slave server 3'],
        ['dsn' => 'dsn for slave server 4'],
    ],
]
```

La configuration ci-dessus spécifie deux maîtres et quatre esclaves. Le composant `Connection` prend aussi en charge l'équilibrage de charge et le basculement entre maîtres juste comme il le fait pour les esclaves. Une différence est que, si aucun des maîtres n'est disponible, une exception est levée.

> **Note :** lorsque vous utilisez la propriété `masters` pour configurer un ou plusieurs maîtres, toutes les autres propriétés pour spécifier une connexion à une base de données (p. ex. `dsn`, `username`, `password`) avec l'objet `Connection` lui-même sont ignorées.

Par défaut, les transactions utilisent la connexion au maître. De plus, dans une transaction, toutes les opérations de base de données utilisent la connexion au maître. Par exemple :

```
$db = Yii::$app->db;
// la transaction est démarrée sur la connexion au maître
$transaction = $db->beginTransaction();

try {
    //  les deux requêtes sont effectuées auprès du maître
    $rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
    $db->createCommand("UPDATE user SET username='demo' WHERE
    id=1")->execute();

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
}
```

Si vous voulez démarrer une transaction avec une connexion à un esclave, vous devez le faire explicitement, comme ceci :

```
$transaction = Yii::$app->db->slave->beginTransaction();
```

Parfois, vous désirez forcer l'utilisation de la connexion au maître pour effectuer une requête en lecture . Cela est possible avec la méthode `useMaster()` :

```
$rows = Yii::$app->db->useMaster(function ($db) {
    return $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
});
```

Vous pouvez aussi définir directement `Yii::$app->db->enableSlaves` à `false` (faux) pour rediriger toutes les requêtes vers la connexion au maître.

6.1.6 Travail avec le schéma de la base de données

Les objets d'accès aux bases de données de Yii DAO fournissent un jeu complet de méthodes pour vous permettre de manipuler le schéma de la base de données, comme créer de nouvelles tables, supprimer une colonne d'une table, etc. Ces méthodes sont listées ci-après :

- `createTable()` : crée une table
- `renameTable()` : renomme une table
- `dropTable()` : supprime une table
- `truncateTable()` : supprime toutes les lignes dans une table
- `addColumn()` : ajoute une colonne
- `renameColumn()` : renomme une colonne
- `dropColumn()` : supprime une colonne
- `alterColumn()` : modifie une colonne
- `addPrimaryKey()` : ajoute une clé primaire
- `dropPrimaryKey()` : supprime une clé primaire
- `addForeignKey()` : ajoute un clé étrangère
- `dropForeignKey()` : supprime une clé étrangère
- `createIndex()` : crée un index
- `dropIndex()` : supprime un index

Ces méthodes peuvent être utilisées comme suit :

```
// CREATE TABLE
Yii::$app->db->createCommand()->createTable('post', [
    'id' => 'pk',
    'title' => 'string',
    'text' => 'text',
]);
```

Le tableau ci-dessus décrit le nom et le type des colonnes à créer. Pour les types de colonne, Yii fournit un jeu de types de donnée abstraits, qui permettent de définir un schéma de base de données indifférent au type de base de données. Ces types sont convertis en définition de types spécifiques au système de gestion de base de données qui dépendent de la base de données

dans laquelle la table est créée. Reportez-vous à la documentation de l'API de la méthode `createTable()` pour plus d'informations.

En plus de changer le schéma de la base de données, vous pouvez aussi retrouver les informations de définition d'une table via la méthode `getTableSchema()` d'une connexion à une base de données. Par exemple :

```
$table = Yii::$app->db->getTableSchema('post');
```

La méthode retourne un objet `yii\db\TableSchema` qui contient les information sur les colonnes de la table, les clés primaires, les clés étrangères, etc. Toutes ces informations sont essentiellement utilisées par le constructeur de requêtes et par l'enregistrement actif pour vous aider à écrire du code indifférent au type de la base de données.

6.2 Le constructeur de requêtes

Construit sur la base des objets d'accès aux bases de données (DAO), le constructeur de requêtes vous permet de construire des requêtes SQL par programme qui sont indifférentes au système de gestion de base de données utilisé. Comparé à l'écriture d'instructions SQL brutes, l'utilisation du constructeur de requêtes vous aide à écrire du code relatif à SQL plus lisible et à générer des instructions SQL plus sûres.

L'utilisation du constructeur de requêtes comprend ordinairement deux étapes :

1. Construire un objet `yii\db\Query` pour représenter différentes parties (p. ex. `SELECT`, `FROM`) d'une instruction SQL.
2. Exécuter une méthode de requête (p. ex. `all()`) de `yii\db\Query` pour retrouver des données dans la base de données.

Le code suivant montre une manière typique d'utiliser le constructeur de requêtes.

```
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->all();
```

Le code ci-dessus génère et exécute la requête SQL suivante, dans laquelle le paramètre `:last_name` est lié à la chaîne de caractères `'Smith'`.

```
SELECT `id`, `email`
FROM `user`
WHERE `last_name` = :last_name
LIMIT 10
```

> **Info :** génélalement vous travaillez essentiellement avec `yii\db\Query` plutôt qu'avec `yii\db\QueryBuilder`. Le dernier est implicitement invoqué par le premier lorsque vous appelez une des méthodes de requête. `yii\db\QueryBuilder` est la classe en charge de la génération des instructions SQL dépendantes du système de gestion de base de données (p. ex. entourer les noms de table/colonne par des marques de citation différemment) à partir d'objets `yii\db\Query` indifférents au système de gestion de base de données.

6.2.1 Construction des requêtes

Pour construire un objet `yii\db\Query`, vous appelez différentes méthodes de construction de requêtes pour spécifier différentes parties de la requête SQL. Les noms de ces méthodes ressemblent aux mots clés de SQL utilisés dans les parties correspondantes de l'instruction SQL. Par exemple, pour spécifier la partie `FROM` d'une requête SQL, vous appelez la méthode `from()`. Toutes les méthodes de construction de requêtes retournent l'objet *query* lui-même, ce qui vous permet d'enchaîner plusieurs appels.

Dans ce qui suit, nous décrivons l'utilisation de chacune des méthodes de requête.

`select()`

La méthode `select()` spécifie le fragment `SELECT` d'une instruction SQL. Vous pouvez spécifier les colonnes à sélectionner soit sous forme de chaînes de caractères, soit sous forme de tableaux, comme ci-après. Les noms des colonnes sélectionnées sont automatiquement entourés des marques de citation lorsque l'instruction SQL est générée à partir de l'objet *query* (requête).

```
$query->select(['id', 'email']);

// équivalent à:

$query->select('id, email');
```

Les noms des colonnes sélectionnées peuvent inclure des préfixes de table et/ou des alias de colonne, comme vous le faites en écrivant une requête SQL brute. Par exemple :

```
$query->select(['user.id AS user_id', 'email']);

// équivalent à:

$query->select('user.id AS user_id, email');
```

Si vous utilisez le format tableau pour spécifier les colonnes, vous pouvez aussi utiliser les clés du tableau pour spécifier les alias de colonne. Par exemple, le code ci-dessus peut être réécrit comme ceci :

```
$query->select(['user_id' => 'user.id', 'email']);
```

Si vous n'appelez pas la méthode `select()` en construisant une requête, `*` est sélectionné, ce qui signifie la sélection de *toutes* les colonnes.

En plus des noms de colonne, vous pouvez aussi sélectionner des expression de base de données. Vous devez utiliser le format tableau en sélectionnant une expression de base de données qui contient des virgules pour éviter des entourages automatiques incorrects des noms par des marques de citation. Par exemple :

```
$query->select(["CONCAT(first_name, ' ', last_name) AS full_name",
'email']);
```

Comme en tout lieu où il est fait appel à du SQL brut, vous devez utiliser la syntaxe des marques de citation indifférentes au système de gestion de base de données pour les noms de table et de colonne lorsque vous écrivez les expressions de base de données dans `select`.

Depuis la version 2.0.1, vous pouvez aussi sélectionner des sous-requêtes. Vous devez spécifier chacune des sous-requêtes en termes d'objet `yii\db\Query`. Par exemple :

```
$subQuery = (new Query())->select('COUNT(*)')->from('user');

// SELECT `id`, (SELECT COUNT(*) FROM `user`) AS `count` FROM `post`
$query = (new Query())->select(['id', 'count' => $subQuery])->from('post');
```

Pour sélectionner des lignes distinctes, vous pouvez appeler `distinct()`, comme ceci :

```
// SELECT DISTINCT `user_id` ...
$query->select('user_id')->distinct();
```

Vous pouvez appeler `addSelect()` pour sélectionner des colonnes additionnelles. Par exemple :

```
$query->select(['id', 'username'])
    ->addSelect(['email']);
```

`from()`

La méthode `from()` spécifie le fragment `FROM`d'une instruction. Par exemple :

```
// SELECT * FROM `user`
$query->from('user');
```

Vous pouvez spécifier les tables à sélectionner soit sous forme de chaînes de caractères, soit sous forme de tableaux. Les noms de table peuvent contenir des préfixes et/ou des alias de table. Par exemple :

```php
$query->from(['public.user u', 'public.post p']);

// équivalent à :

$query->from('public.user u, public.post p');
```

Si vous utilisez le format tableau, vous pouvez aussi utiliser les clés du tableau pour spécifier les alias de table, comme suit :

```php
$query->from(['u' => 'public.user', 'p' => 'public.post']);
```

En plus des noms de table, vous pouvez aussi sélectionner à partir de sous-requêtes en les spécifiant en termes d'objets `yii\db\Query`. Par exemple :

```php
$subQuery = (new Query())->select('id')->from('user')->where('status=1');

// SELECT * FROM (SELECT `id` FROM `user` WHERE status=1) u
$query->from(['u' => $subQuery]);
```

Préfixes Un `préfixe de table` peut aussi être appliqué. Les instructions de mise en œuvre sont données à la section "Entourage des noms de table et de colonne par des marques de citation" du guide sur les objets d'accès aux bases de données" .

`where()`

La méthode `where()` spécifie le fragment `WHERE`d'une requête SQL. Vous pouvez utiliser un des quatre formats suivants pour spécifier une condition `WHERE` :

- format chaîne de caractères, p. ex. `'status=1'`
- format haché, p. ex. `['status' => 1, 'type' => 2]`
- format opérateur, p. ex. `['like', 'name', 'test']`
- format objet, p. ex. `new LikeCondition('name', 'LIKE', 'test')`

Format chaîne de caractères Le format chaîne de caractères est celui qui convient le mieux pour spécifier des conditions très simples ou si vous avez besoin d'utiliser les fonctions incorporées au système de gestion de base de données. Il fonctionne comme si vous écriviez une requête SQL brute. Par exemple :

```php
$query->where('status=1');

// ou utilisez la liaison des paramètres pour lier des valeurs dynamiques
des paramètres.
$query->where('status=:status', [':status' => $status]);

//  SQL brute utilisant la fonction MySQL YEAR() sur un champ de date
$query->where('YEAR(somedate) = 2015');
```

N'imbriquez PAS les variables directement dans la condition comme ce qui suit, spécialement si les valeurs des variables proviennent d'entrées utilisateur, parce que cela rendrait votre application SQL sujette aux attaques par injections SQL.

```
// Dangereux! Ne faites PAS cela sauf si vous êtes tout à fait sûr que
$status est un entier
$query->where("status=$status");
```

Lorsque vous utilisez la `liaison des paramètres`, vous pouvez appeler `params()` ou `addParams()` pour spécifier les paramètres séparément.

```
$query->where('status=:status')
    ->addParams([':status' => $status]);
```

Comme dans tous les endroits ou il est fait appel à du SQL, vous pouvez utiliser la syntaxe d'entourage par des marques de citation indifférente au système de gestion de base de données pour les noms de table et de colonne lorsque vous écrivez les conditions au format chaîne de caractères.

Format haché Le format valeur de hachage convient le mieux pour spécifier de multiples sous-conditions concaténées par `AND`, chacune étant une simple assertion d'égalité. Il se présente sous forme de tableau dont les clés sont les noms des colonnes et les valeurs les valeurs correspondantes que les valeurs des colonnes devraient avoir. Par exemple :

```
// ...WHERE (`status` = 10) AND (`type` IS NULL) AND (`id` IN (4, 8, 15))
$query->where([
    'status' => 10,
    'type' => null,
    'id' => [4, 8, 15],
]);
```

Comme vous pouvez le voir, le constructeur de requêtes est assez intelligent pour manipuler correctement les valeurs qui sont soit nulles, soit des tableaux.

Vous pouvez utiliser aussi des sous-requêtes avec le format haché comme suit :

```
$userQuery = (new Query())->select('id')->from('user');

// ...WHERE `id` IN (SELECT `id` FROM `user`)
$query->where(['id' => $userQuery]);
```

En utilisant le format haché, Yii, en interne, utilise la liaison des paramètres pour les valeurs de façon à ce que, contrairement au format chaîne de caractères, vous n'ayez pas à ajouter les paramètres à la main. Cependant, notez que Yii ne procède pas à l'échappement des noms de colonne, c'est

pourquoi si vous passez un nom de variable obtenu de l'utilisateur en tant que nom de colonne sans vérification, l'application devient vulnérable à l'injection SQL. Afin de maintenir l'application sûre, soit n'utilisez pas de variables comme nom de colonne, soit filtrez les variables par une liste blanche. Dans le cas où vous avez besoin d'obtenir un nom de colonne de l'utilisateur, lisez l'article du guide Filtrage des données. Ainsi l'exemple de code suivant est vulnérable :

```
// Vulnerable code:
$column = $request->get('column');
$value = $request->get('value');
$query->where([$column => $value]);
// $value est sûre, mais le nom de colonne n'est pas encodé!
```

Format opérateur Le format opérateur vous permet de spécifier des conditions arbitraires par programmation. Il accepte les formats suivants :

```
[operator, operand1, operand2, ...]
```

dans lequel chacun des opérandes peut être spécifié au format chaîne de caractères, au format haché ou au format opérateur de façon récursive, tandis que l'opérateur peut être un de ceux qui suivent :

— `and` : les opérandes doivent être concaténés en utilisant `AND`. Par exemple, `['and', 'id=1', 'id=2']` génère `id=1 AND id=2`. Si un opérande est un tableau, il est converti en une chaîne de caractères en utilisant les règles décrites ici. Par exemple, `['and', 'type=1', ['or', 'id=1', 'id=2']]` génère `type=1 AND (id=1 OR id=2)`. La méthode ne procède à aucun entourage par des marques de citation, ni à aucun échappement.

— `or` : similaire à l'opérateur `and` sauf que les opérandes sont concaténés en utilisant `OR`.

— `not` : ne réclame que l'opérande 1, qui est emballé dans `NOT()`. Par exemple, `['not', 'id=1']` génère `NOT (id=1)`. L'opérande 1 peut aussi être un tableau pour décrire des expressions multiples. Par exemple `['not', ['status' => 'draft', 'name' => 'example']]` génère `NOT ((status='draft') AND (name='example'))`.

— `between` : l'opérande 1 doit être le nom de la colonne, et les opérandes 2 et 3 doivent être les valeurs de départ et de fin de la plage dans laquelle la colonne doit être. Par exemple, `['between', 'id', 1, 10]` génère `id BETWEEN 1 AND 10`. Dans le cas où vous avez besoin de construire une expression dans laquelle la valeur est entre deux colonnes (telle que `11 BETWEEN min_id AND max_id`), vous devez utiliser `BetweenColumnsCondition`. Reportez-vous au chapitre Conditions – Format d'objet pour en savoir plus sur la définition des conditions d'objet.

— `not between` : similaire à `between` sauf que `BETWEEN` est remplacé par `NOT BETWEEN` dans la condition générée.

— `in` : l'opérande 1 doit être une colonne ou une expression de base de données. L'opérande 2 peut être soit un tableau, soit un objet `Query`. Il génère une condition `IN`. Si l'opérande 2 est un tableau, il représente la plage des valeurs que la colonne ou l'expression de base de données peut prendre. Si l'opérande 2 est un objet `Query`, une sous-requête est générée et utilisée comme plage pour la colonne ou l'expression de base de données. Par exemple, `['in', 'id', [1, 2, 3]]` génère `id IN (1, 2, 3)`. La méthode assure correctement l'entourage des noms de colonnes par des marques de citation et l'échappement des valeurs de la plage. L'opérateur `in` prend aussi en charge les colonnes composites. Dans ce cas, l'opérande 1 doit être un tableau des colonnes, tandis que l'opérateur 2 doit être un tableau de tableaux, ou un objet `Query` représentant la plage de colonnes.
— `not in` : similaire à l'opérateur `in` sauf que `IN` est remplacé par `NOT IN` dans la condition générée.
— `like` : l'opérande 1 doit être une colonne ou une expression de base de données, tandis que l'opérande 2 doit être une chaîne de caractères ou un tableau représentant les valeurs que cette colonne ou cette expression de base de données peuvent être. Par exemple, `['like', 'name', 'tester']` génère `name LIKE '%tester%'`. Lorsque la plage de valeurs est donnée sous forme de tableau, de multiples prédicats `LIKE` sont générés et concaténés en utilisant `AND`. Par exemple, `['like', 'name', ['test', 'sample']]` génère `name LIKE '%test%' AND name LIKE '%sample%'`. Vous pouvez également fournir un troisième paramètre facultatif pour spécifier comment échapper les caractères spéciaux dans les valeurs. Les opérandes doivent être un tableau de correspondance entre les caractères spéciaux et les contre-parties échappées. Si cet opérande n'est pas fourni, une mise en correspondance par défaut est utilisée. Vous pouvez utiliser `false` ou un tableau vide pour indiquer que les valeurs sont déjà échappées et qu'aucun échappement ne doit être appliqué. Notez que lorsqu'un tableau de mise en correspondance pour l'échappement est utilisé (ou quand le troisième opérande n'est pas fourni), les valeurs sont automatiquement entourées par une paire de caractères `%`.

  > **Note :** lors de l'utilisation de PostgreSQL vous pouvez aussi utiliser `ilike`[18]
  > >à la place de `like` pour une mise en correspondance insensible à la casse.

— `or like` : similaire à l'opérateur `like` sauf que `OR`est utilisé pour concaténer les prédicats `LIKE` quand l'opérande 2 est un tableau.
— `not like` : similaire à l'opérateur `like` sauf que `LIKE` est remplacé par `NOT LIKE` dans la condition générée.

18. https://www.postgresql.org/docs/8.3/static/functions-matching.html#FUNCTIONS-LIKE

— `or not like` : similaire à l'opérateur `not like` sauf que `OR` est utilisé pour concaténer les prédicats `NOT LIKE`.
— `exists` : requiert un opérande que doit être une instance de `yii\db\Query` représentant la sous-requête.
Il construit une expression `EXISTS (sub-query)`.
— `not exists` : similaire à l'opérateur `exists` et construit une expression `NOT EXISTS (sub-query)`.
— `>`, `<=`, ou tout autre opérateur de base de données valide qui accepte deux opérandes : le premier opérande doit être un nom de colonne, tandis que le second doit être une valeur. Par exemple, `['>', 'age', 10]` génère `age>10`.

En utilisant le format opérateur, Yii, en interne, utilise la liaison des paramètres afin, que contrairement au format chaîne de caractères, ici, vous n'avez pas besoin d'ajouter les paramètres à la main. Cependant, notez que Yii ne procède pas à l'échappement des noms de colonne, c'est pourquoi si vous passez un nom de variable obtenu de l'utilisateur en tant que nom de colonne sans vérification, l'application devient vulnérable à l'injection SQL. Afin de maintenir l'application sûre, soit n'utilisez pas de variables comme nom de colonne, soit filtrez les variables par une liste blanche. Dans le cas où vous avez besoin d'obtenir un nom de colonne de l'utilisateur, lisez l'article du guide Filtrage des données. Ainsi l'exemple de code suivant est vulnérable :

```
// Code vulnérable:
$column = $request->get('column');
$value = $request->get('value');
$query->where(['=', $column, $value]);
// $value est sûre, mais le nom  $column n'est pas encodé !
```

Format objet Le format objet est disponible depuis 2.0.14 et est à la fois le moyen plus puissant et le plus complexe pour définir des conditions. Vous devez le suivre si vous voulez construire votre propre abstraction au-dessus du constructeur de requêtes (query builder) ou si vous voulez mettre en œuvre vos propres conditions complexes.

Les instances de classes de condition sont immuables. Le seul but est de stocker des données de condition et de fournir des obtenteurs (getters) pour les constructeurs de conditions. La classe « constructeur de condition » (condition builder) est une classe qui contient la logique qui transforme les données stockées en condition dans une expression SQL.

En interne, les formats décrits plus haut sont implicitement convertis en format objet avant de construire le SQL brut, aussi est-il possible de combiner les formats en une condition unique :

```
$query->andWhere(new OrCondition([
    new InCondition('type', 'in', $types),
    ['like', 'name', '%good%'],
```

```
    'disabled=false'
]))
```

La conversion du format opérateur au format objet est accomplie en fonction de la propriété `QueryBuilder::conditionClasses` , qui fait correspondre des noms d'opérateurs à des nom de classe représentatives :

- `AND`, `OR` -> `yii\db\conditions\ConjunctionCondition`
- `NOT` -> `yii\db\conditions\NotCondition`
- `IN`, `NOT IN` -> `yii\db\conditions\InCondition`
- `BETWEEN`, `NOT BETWEEN` -> `yii\db\conditions\BetweenCondition`

Et ainsi de suite.

L'utilisation du format objet rend possible de créer vos propres conditions ou de changer la manière dont celles par défaut sont construites. Reportez-vous au chapitre Ajout de conditions et d'expressions personnalisées pour en savoir plus.

Ajout de conditions Vous pouvez utiliser `andWhere()` ou `orWhere()` pour ajouter des conditions supplémentaires à une condition existante. Vous pouvez les appeler plusieurs fois pour ajouter plusieurs conditions séparément. Par exemple :

```
$status = 10;
$search = 'yii';

$query->where(['status' => $status]);

if (!empty($search)) {
   $query->andWhere(['like', 'title', $search]);
}
```

Si `$search` n'est pas vide, la condition `WHERE` suivante est générée :

```
WHERE (`status` = 10) AND (`title` LIKE '%yii%')
```

Conditions de filtrage Lors de la construction de conditions `WHERE` basées sur des entrées de l'utilisateur final, vous voulez généralement ignorer les valeurs entrées qui sont vides. Par exemple, dans un formulaire de recherche par nom d'utilisateur ou par adresse de courriel, vous aimeriez ignorer la condition nom d'utilisateur/adresse de courriel si l'utilisateur n'a rien saisi dans les champs correspondants. Vous pouvez faire cela en utilisant la méthode `filterWhere()` :

```
// $username et $email sont entrées par l'utilisateur
$query->filterWhere([
    'username' => $username,
    'email' => $email,
]);
```

La seule différence entre `filterWhere()` et `where()` est que la première ignore les valeurs vides fournies dans la condition au format haché. Ainsi si `$email` est vide alors que `$username` ne l'est pas, le code ci dessus produit la condition SQL `WHERE username=:username`.

> **Info :** une valeur est considérée comme vide si elle est nulle, un tableau vide, ou un chaîne de caractères vide, ou un chaîne de caractères constituée d'espaces uniquement.

Comme avec `andWhere()` et `orWhere()`, vous pouvez utiliser `andFilterWhere()` et `orFilterWhere()` pour ajouter des conditions de filtrage supplémentaires à une condition existante.

En outre, il y a `yii\db\Query::andFilterCompare()` qui peut déterminer intelligemment l'opérateur en se basant sur ce qu'il y a dans les valeurs :

```
$query->andFilterCompare('name', 'John Doe');
$query->andFilterCompare('rating', '>9');
$query->andFilterCompare('value', '<=100');
```

Vous pouvez aussi utiliser un opérateur explicitement :

```
$query->andFilterCompare('name', 'Doe', 'like');
```

Depuis Yii 2.0.1, il existe des méthodes similaires pour la condition `HAVING` :

- `filterHaving()`
- `andFilterHaving()`
- `orFilterHaving()`

`orderBy()`

La méthode `orderBy()` spécifie le fragment `ORDER BY` d'une requête SQL. Par exemple :

```
// ... ORDER BY `id` ASC, `name` DESC
$query->orderBy([
    'id' => SORT_ASC,
    'name' => SORT_DESC,
]);
```

Dans le code ci-dessus, les clés du tableau sont des noms de colonnes, tandis que les valeurs sont les instructions de direction de tri. La constante PHP `SORT_ASC` spécifie un tri ascendant et `SORT_DESC`, un tri descendant.

Si `ORDER BY` ne fait appel qu'à des noms de colonnes simples, vous pouvez le spécifier en utilisant une chaîne de caractères, juste comme vous le faites en écrivant des instructions SQL brutes. Par exemple :

```
$query->orderBy('id ASC, name DESC');
```

> **Note :** vous devez utiliser le format tableau si `ORDER BY` fait appel à une expression de base de données.

Vous pouvez appeler `addOrderBy()` pour ajouter des colonnes supplémentaires au fragment `ORDER BY`. Par exemple :

```
$query->orderBy('id ASC')
    ->addOrderBy('name DESC');
```

`groupBy()`

La méthode `groupBy()` spécifie le fragment `GROUP BY` d'une requête SQL. Par exemple :

```
// ... GROUP BY `id`, `status`
$query->groupBy(['id', 'status']);
```

Si `GROUP BY` ne fait appel qu'à des noms de colonnes simples, vous pouvez le spécifier en utilisant un chaîne de caractères, juste comme vous le faîtes en écrivant des instructions SQL brutes. Par exemple :

```
$query->groupBy('id, status');
```

> **Note :** vous devez utiliser le format tableau si `GROUP BY` fait appel à une expression de base de données.

Vous pouvez appeler `addGroupBy()` pour ajouter des colonnes au fragment `GROUP BY`. Par exemple :

```
$query->groupBy(['id', 'status'])
    ->addGroupBy('age');
```

`having()`

La méthode `having()` spécifie le fragment `HAVING` d'un requête SQL. Elle accepte une condition qui peut être spécifiée de la même manière que celle pour where(). Par exemple :

```
// ... HAVING `status` = 1
$query->having(['status' => 1]);
```

Reportez-vous à la documentation de where() pour plus de détails sur la manière de spécifier une condition.

Vous pouvez appeler `andHaving()` ou `orHaving()` pour ajouter des conditions supplémentaires au fragment `HAVING` fragment. Par exemple :

```
// ... HAVING (`status` = 1) AND (`age` > 30)
$query->having(['status' => 1])
    ->andHaving(['>', 'age', 30]);
```

`limit()` et `offset()`

Les méthodes `limit()` et `offset()` spécifient les fragments `LIMIT` et `OFFSET` d'une requête SQL. Par exemple :

```
// ... LIMIT 10 OFFSET 20
$query->limit(10)->offset(20);
```

Si vous spécifiez une limite ou un décalage (p. ex. une valeur négative), il est ignoré.

> **Info :** pour les systèmes de gestion de base de données qui ne prennent pas en charge `LIMIT` et `OFFSET` (p. ex. MSSQL), le constructeur de requêtes génère une instruction SQL qui émule le comportement `LIMIT`/`OFFSET`.

`join()`

La méthode `join()` spécifie le fragment `JOIN` d'une requête SQL. Par exemple :

```
// ... LEFT JOIN `post` ON `post`.`user_id` = `user`.`id`
$query->join('LEFT JOIN', 'post', 'post.user_id = user.id');
```

La méthode `join()` accepte quatre paramètres :

- `$type` : type de jointure , p. ex. `'INNER JOIN'`, `'LEFT JOIN'`.
- `$table` : le nom de la table à joindre.
- `$on` : facultatif, la condition de jointure, c.-à-d. le fragment `ON`. Reportez-vous à where() pour des détails sur la manière de spécifier une condition. Notez, que la syntaxe tableau ne fonctionne **PAS** pour spécifier une condition basée sur une colonne, p. ex. `['user.id' => 'comment.userId']` conduit à une condition où l'identifiant utilisateur doit être égal à la chaîne de caractères `'comment.userId'`. Vous devez utiliser la syntaxe chaîne de caractères à la place et spécifier la condition `'user.id = comment.userId'`.
- `$params` : facultatif, les paramètres à lier à la condition de jointure.

Vous pouvez utiliser les méthodes raccourcies suivantes pour spécifier `INNER JOIN`, `LEFT JOIN` et `RIGHT JOIN`, respectivement.

- `innerJoin()`
- `leftJoin()`
- `rightJoin()`

Par exemple :

```
$query->leftJoin('post', 'post.user_id = user.id');
```

Pour joindre plusieurs tables, appelez les méthodes join ci-dessus plusieurs fois, une fois pour chacune des tables.

En plus de joindre des tables, vous pouvez aussi joindre des sous-requêtes. Pour faire cela, spécifiez les sous-requêtes à joindre sous forme d'objets `yii\db\Query`. Par exemple :

```
$subQuery = (new \yii\db\Query())->from('post');
$query->leftJoin(['u' => $subQuery], 'u.id = author_id');
```

Dans ce cas, vous devez mettre la sous-requête dans un tableau et utiliser les clés du tableau pour spécifier les alias.

`union()`

La méthode `union()` spécifie le fragment `UNION` d'une requête SQL. Par exemple :

```
$query1 = (new \yii\db\Query())
    ->select("id, category_id AS type, name")
    ->from('post')
    ->limit(10);

$query2 = (new \yii\db\Query())
    ->select('id, type, name')
    ->from('user')
    ->limit(10);

$query1->union($query2);
```

Vous pouvez appeler `union()` plusieurs fois pour ajouter plus de fragments `UNION`.

6.2.2 Méthodes de requête

L'objet `yii\db\Query` fournit un jeu complet de méthodes pour différents objectifs de requêtes :

- `all()` : retourne un tableau de lignes dont chacune des lignes est un tableau associatif de paires clé-valeur.
- `one()` : retourne la première ligne du résultat.
- `column()` : retourne la première colonne du résultat.
- `scalar()` : retourne une valeur scalaire située au croisement de la première ligne et de la première colonne du résultat.
- `exists()` : retourne une valeur précisant si le résultat de la requête contient un résultat.
- `count()` : retourne le résultat d'une requête `COUNT`..
- D'autres méthodes d'agrégation de requêtes, y compris `sum($q)`, `average($q)`, `max($q)`, `min($q)`. Le paramètre `$q` est obligatoire pour ces méthodes et peut être soit un nom de colonne, soit une expression de base de données.

Par exemple :

```php
// SELECT `id`, `email` FROM `user`
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->all();

// SELECT * FROM `user` WHERE `username` LIKE `%test%`
$row = (new \yii\db\Query())
    ->from('user')
    ->where(['like', 'username', 'test'])
    ->one();
```

> **Note :** la méthode `one()` retourne seulement la première ligne du résultat de la requête. Elle n'ajoute PAS `LIMIT 1` à l'instruction SQL générée. Cela est bon et préférable si vous savez que la requête ne retourne qu'une seule ou quelques lignes de données (p. ex. si vous effectuez une requête avec quelques clés primaires). Néanmoins, si la requête peut potentiellement retourner de nombreuses lignes de données, vous devriez appeler `limit(1)` explicitement pour améliorer la performance, p. ex. `(new \yii\db\Query())->from('user')->limit(1)->one()`.

Toutes ces méthodes de requête acceptent un paramètre supplémentaire `$db` représentant la `connexion à la base de données` qui doit être utilisée pour effectuer la requête. Si vous omettez ce paramètre, le composant d'application `db` est utilisé en tant que connexion à la base de données. Ci-dessous, nous présentons un autre exemple utilisant la méthode `count()` :

```php
// exécute SQL: SELECT COUNT(*) FROM `user` WHERE `last_name`=:last_name
$count = (new \yii\db\Query())
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->count();
```

Lorsque vous appelez une méthode de requête de `yii\db\Query`, elle effectue réellement le travail suivant en interne :

- Appelle `yii\db\QueryBuilder` pour générer une instruction SQL basée sur la construction courante de `yii\db\Query` ;
- Crée un objet `yii\db\Command` avec l'instruction SQL générée ;
- Appelle une méthode de requête (p. ex. `queryAll()`) de `yii\db\Command` pour exécuter une instruction SQL et retrouver les données.

Parfois, vous voulez peut-être examiner ou utiliser une instruction SQL construite à partir d'un objet `yii\db\Query`. Vous pouvez faire cela avec le code suivant :

```php
$command = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
```

```
    ->createCommand();

// affiche l'instruction SQL
echo $command->sql;
// affiche les paramètres à lier
print_r($command->params);

// retourne toutes les lignes du résultat de la requête
$rows = $command->queryAll();
```

Indexation des résultats de la requête

Lorsque vous appelez `all()`, elle retourne un tableau de lignes qui sont indexées par des entiers consécutifs. Parfois, vous désirez peut-être les indexer différemment, comme les indexer par une colonne particulière ou par des expressions donnant une valeur. Vous pouvez le faire en appelant `indexBy()` avant `all()`. Par exemple :

```
// retourne [100 => ['id' => 100, 'username' => '...', ...], 101 => [...],
103 => [...], ...]
$query = (new \yii\db\Query())
    ->from('user')
    ->limit(10)
    ->indexBy('id')
    ->all();
```

Pour indexer par des valeurs d'expressions, passez une fonction anonyme à la méthode `indexBy()` :

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy(function ($row) {
        return $row['id'] . $row['username'];
    })->all();
```

Le fonction anonyme accepte un paramètre `$row` qui contient les données de la ligne courante et retourne une valeur scalaire qui est utilisée comme la valeur d'index de la ligne courante.

> **Note :** contrairement aux méthodes de requête telles que `groupBy()` ou `orderBy()` qui sont converties en SQL et font partie de la requête, cette méthode ne fait son travail qu'après que les données ont été retrouvées dans la base de données. Cela signifie que seuls les noms de colonne qui on fait partie du fragment SELECT dans votre requête peuvent être utilisés. De plus, si vous avez sélectionné une colonne avec un préfixe de table, p. ex. `customer.id`, le jeu de résultats ne contient que `id` c'est pourquoi vous devez appeler `->indexBy('id')` sans préfixe de table.

Requêtes par lots

Lorsque vous travaillez sur de grandes quantités de données, des méthodes telles que `yii\db\Query::all()` ne conviennent pas car elles requièrent le chargement de toutes les données en mémoire du client. Pour résoudre cet problème Yii assure la prise en charge de requêtes par lots. Le serveur conserve les résultats de la requête, et le client utilise un curseur pour itérer sur le jeu de résultats un lot à la fois.

> Attention : il existe des limitations connues et des solutions de contournement pour la mise en œuvre des requêtes par lots par MySQL.

Les requêtes par lots peuvent être utilisées comme suit :

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
    ->orderBy('id');

foreach ($query->batch() as $users) {
    // $users est dans un tableau de 100 ou moins lignes de la table user.
}

// ou si vous voulez itérer les lignes une par une
foreach ($query->each() as $user) {
     //les données sont retrouvées du serveur en lots de 100,
    // $user représente une ligne de données de la table user.
}
```

Les méthodes `yii\db\Query::batch()` et `yii\db\Query::each()` retournent un objet `yii\db\BatchQueryResult` qui implémente l'interface `Iterator` et qui, par conséquent, peut être utilisé dans une construction `foreach`. Durant la première itération, une requête SQL est faite à la base de données. Les données sont retrouvées en lots dans les itérations suivantes. Par défaut, la taille du lot est 100, ce qui signifie que 100 lignes sont retrouvées dans chacun des lots. Vous pouvez changer la taille du lot en passant le premier paramètre des méthodes `batch()` ou `each()`.

Comparée à la requête `yii\db\Query::all()`, la requête par lots ne charge que 100 lignes de données à la fois en mémoire. Si vous traitez les données et les détruisez tout de suite, la requête par lots réduit l'utilisation de la mémoire.

Si vous spécifiez l'indexation du résultat de la requête par une colonne via `yii\db\Query::indexBy()`, la requête par lots conserve l'index approprié. Par exemple :

Par exemple :

```
$query = (new \yii\db\Query())
    ->from('user')
```

```
    ->indexBy('username');

foreach ($query->batch() as $users) {
    // $users est indexé par la colonne "username"
}

foreach ($query->each() as $username => $user) {
    // ...
}
```

Limitations des requêtes par lots dans MySQL La mise en œuvre des requêtes par lots de MySQL s'appuie sur la bibliothèque du pilote PDO. Par défaut, les requêtes MySQL sont `mises en tampon`[19]. Cela empêche d'utiliser le curseur pour obtenir les données, parce que cela n'empêche pas le jeu résultant complet d'être chargé dans la mémoire du client par le pilote.

> **Note :** lorsque `libmysqlclient` est utilisé (typique de PHP5), la limite mémoire de PHP ne compte pas la mémoire utilisée par les jeux de résultats. Il peut sembler que les requêtes par lot fonctionnent correctement, mais en réalité l'intégralité du jeu de données est chargé dans la mémoire du client.

Pour désactiver la mise en tampon et réduire les exigences en mémoire client, la propriété connexion à PDOT `PDO::MYSQL_ATTR_USE_BUFFERED_QUERY` doit être définie à `false`. Cependant, jusqu'à ce que l'intégralité du jeu de données ait été retrouvé, aucune autre requête ne peut être faite via la même connexion. Cela peut empêcher `ActiveRecord` d'effectuer une requête pour obtenir le schéma de table lorsqu'il le doit. Si cela n'est pas un problème (le schéma de table est déjà mis en cache), il est possible de commuter la connexion originale en mode sans mise en tampon, et de revenir en arrière lorsque la requête par lots est terminée.

```
Yii::$app->db->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY,
false);

// Effectue la requête par lots

Yii::$app->db->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, true);
```

> **Note :** dans le cas de MyISAM, pour toute la durée de la requête par lots, la table peut devenir verrouillée, retardant ainsi ou refusant l'accès en écriture pour une autre connexion. Lors de l'utilisation de requêtes sans mise en tampon, essayez de conserver le curseur ouvert pour un temps aussi court que possible.

Si le schéma n'est pas mis en cache, ou s'il est nécessaire d'effectuer d'autres requêtes alors que la requête par lots est en cours de traitement, vous pouvez créer une connexion à la base de données séparée sans mise en tampon :

19. https://www.php.net/manual/fr/mysqlinfo.concepts.buffering.php

```
$unbufferedDb = new \yii\db\Connection([
    'dsn' => Yii::$app->db->dsn,
    'username' => Yii::$app->db->username,
    'password' => Yii::$app->db->password,
    'charset' => Yii::$app->db->charset,
]);
$unbufferedDb->open();
$unbufferedDb->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY,
false);
```

Si vous voulez garantir que la `$unbufferedDb` a exactement les mêmes attributs PDO gue la `$db` originale avec mise en tampon mais que `PDO::MYSQL_ATTR_USE_BUFFERED_QUERY` est `false`, envisagez une copie profonde de `$db` [20], définissez le à `false` manuellement.

Ensuite, les requêtes sont créées normalement. La nouvelle connexion est utilisée pour exécuter les requêtes par lots et retrouver des résultats soit par lots, soit un par un :

```
// obtention des données par lots de 1000
foreach ($query->batch(1000, $unbufferedDb) as $users) {
    // ...
}


// les données sont retrouvées dans le serveur par lots de 1000, mais elles
sont itérées une à une
foreach ($query->each(1000, $unbufferedDb) as $user) {
    // ...
}
```

Lorsque la connexion n'est plus nécessaire et que le jeu de résultats a été retrouvé, on peut le fermer :

```
$unbufferedDb->close();
```

> **Note :** une requête sans mise en tampon utilise moinsde mémoire du côté PHP, mais peut augmenter la charge du serveur MySQL. Il est recommandé de concevoir votre propre code avec votre pratique en production pour des données massives supplémentaires, par exemple, divisez la plage pour les clés entières, itérez sur elles avec des requêtes sans mise en tampon [21].

Ajout de conditions et expressions personnalisées

Comme cela a été mentionné au chapitre Conditions – Format Object, il est possible de créer des classe de condition personnalisées. Pour l'exemple, créons une condition qui vérifie que des colonnes spécifiques sont inférieures

20. https://github.com/yiisoft/yii2/issues/8420#issuecomment-301423833
21. https://github.com/yiisoft/yii2/issues/8420#issuecomment-296109257

à une valeur donnée. En utilisant le format opérateur, ça devrait ressembler à ce qui suit :

```
[
    'and',
    '>', 'posts', $minLimit,
    '>', 'comments', $minLimit,
    '>', 'reactions', $minLimit,
    '>', 'subscriptions', $minLimit
]
```

Lorsqu'une telle condition est utilisée une seule fois, tout va bien. Dans le cas où elle est utilisée de multiples fois dans une unique requête, cela peut être grandement optimisé. Créons un objet condition personnalisé pour le démontrer.

Yii dispose d'une classe `ConditionInterface`, qui peut être utilisée pour marquer des classes qui représentent une condition. Elle nécessite la mise en œuvre de la méthode `fromArrayDefinition()`, afin de rendre possible la création d'une condition à partir du format tableau. Dans le cas où vous n'en n'avez pas besoin, vous pouvez mettre cette méthode en œuvre avec lancement d'une exception.

Comme nous créons notre classe de condition personnalisée, nous pouvons construire une API qui s'adapte au mieux à notre tâche.

```
namespace app\db\conditions;

class AllGreaterCondition implements \yii\db\conditions\ConditionInterface
{
    private $columns;
    private $value;

    /**
     * @param string[] $columns tableau de colonnes qui doivent être plus
grande que $value
     * @param mixed $value la valeur à laquelle comparer chaque $column
     */
    public function __construct(array $columns, $value)
    {
        $this->columns = $columns;
        $this->value = $value;
    }

    public static function fromArrayDefinition($operator, $operands)
    {
        throw new InvalidArgumentException('Not implemented yet, but we will
        do it later');
    }

    public function getColumns() { return $this->columns; }
    public function getValue() { return $this->vaule; }
}
```

Ainsi nous pouvons créer un objet condition :

```
$conditon = new AllGreaterCondition(['col1', 'col2'], 42);
```

Mais `QueryBuilder` (le constructeur de requêtes) ne sait toujours pas comment élaborer une condition SQL à partir de cet objet. Maintenant nous devons créer un constructeur pour cette condition. Il doit mettre en œuvre une méthode `build()`.

```
namespace app\db\conditions;

class AllGreaterConditionBuilder implements
\yii\db\ExpressionBuilderInterface
{
    use \yii\db\Condition\ExpressionBuilderTrait; // Contient le
    constructeur et la propriété `queryBuilder`.

    /**
     * @param AllGreaterCondition $condition la condition à élaborer
     * @param array $params les paramètres de liaison.
     */
    public function build(ConditionInterface $condition, &$params)
    {
        $value = $condition->getValue();

        $conditions = [];
        foreach ($condition->getColumns() as $column) {
            $conditions[] = new SimpleCondition($column, '>', $value);
        }

        return $this->queryBuider->buildCondition(new
        AndCondition($conditions), $params);
    }
}
```

Ensuite, laissons simplement `QueryBuilder` prendre connaissance de notre nouvelle condition — établissons une correspondance entre lui et notre tableau `expressionBuilders`. Cela peut se faire directement à partir de la configuration de l'application :

```
'db' => [
    'class' => 'yii\db\mysql\Connection',
    // ...
    'queryBuilder' => [
        'expressionBuilders' => [
            'app\db\conditions\AllGreaterCondition' =>
            'app\db\conditions\AllGreaterConditionBuilder',
        ],
    ],
],
```

Maintenant nous sommes en mesure d'utiliser notre condition dans `where()` :

```
$query->andWhere(new AllGreaterCondition(['posts', 'comments', 'reactions',
'subscriptions'], $minValue));
```

Si nous voulons rendre possible la création de notre condition personnalisée en utilisant le format opérateur, nous devons le déclarer dans `QueryBuilder::conditionClasses` :

```
'db' => [
    'class' => 'yii\db\mysql\Connection',
    // ...
    'queryBuilder' => [
        'expressionBuilders' => [
            'app\db\conditions\AllGreaterCondition' =>
            'app\db\conditions\AllGreaterConditionBuilder',
        ],
        'conditionClasses' => [
            'ALL>' => 'app\db\conditions\AllGreaterCondition',
        ],
    ],
],
```

Et créer une mise en œuvre réelle de la méthode `AllGreaterCondition::fromArrayDefinition()` dans `app\db\conditions\AllGreaterCondition` :

```
namespace app\db\conditions;

class AllGreaterCondition implements \yii\db\conditions\ConditionInterface
{
    // ... see the implementation above

    public static function fromArrayDefinition($operator, $operands)
    {
        return new static($operands[0], $operands[1]);
    }
}
```

À la suite de cela, nous pouvons créer notre condition personnalisée en utilisant un format opérateur plus court :

```
$query->andWhere(['ALL>', ['posts', 'comments', 'reactions',
'subscriptions'], $minValue]);
```

Vous pouvez noter que deux conceptes ont été utilisés : Expressions et Conditions. Il y a une `yii\db\ExpressionInterface` qui doit être utilisée pour marquer les objets qui requièrent une classe constructrice d'expression qui met en œuvre `yii\db\ExpressionBuilderInterface` pour être construite. Il existe également une `yii\db\condition\ConditionInterface`, qui étend `ExpressionInterface` et doit être utilisée pour des objets qui peuvent être créés à partir d'un tableau de définition comme cela a été expliqué plus haut, mais qui peuvent aussi bien nécessiter le constructeur.

Pour résumer :

— Expression – est un objet de transfert de données — Data Transfer Object (DTO) — pour un jeu de données, qui peut être compilé en une instruction SQL (un opérateur, une chaîne de caractères, un tableau, JSON, etc).
— Condition – est un super jeu d'expressions, qui agrège de multiples expressions (ou valeurs scalaires) qui peut être compilé en une unique condition SQL.

Vous pouvez créer votre propre classe qui met en œuvre l'interface `ExpressionInterface` pour cacher la complexité de la transformation de données en instructions SQL. Vous en apprendrez plus sur d'autres exemples d'expressions dans le prochain article ;

6.3 Enregistrement actif (*Active Record*)

L'enregistrement actif[22] fournit une interface orientée objet pour accéder aux données stockées dans une base de données et les manipuler. Une classe d'enregistrement actif (ActiveRecord) est associée à une table de base de données, une instance de cette classe représente une ligne de cette table, et un *attribut* d'une instance d'enregistrement actif représente la valeur d'une colonne particulière dans cette ligne. Au lieu d'écrire des instructions SQL brutes, vous pouvez accéder aux attributs de l'objet enregistrement actif et appeler ses méthodes pour accéder aux données stockées dans les tables de la base de données et les manipuler.

Par exemple, supposons que `Customer` soit une classe d'enregistrement actif associée à la table `customer` et que `name` soit une colonne de la table `customer`. Vous pouvez écrire le code suivant pour insérer une nouvelle ligne dans la table `customer` :

```
$customer = new Customer();
$customer->name = 'Qiang';
$customer->save();
```

Le code ci-dessus est équivalent à l'utilisation de l'instruction SQL brute suivante pour MySQL, qui est moins intuitive, plus propice aux erreurs, et peut même poser des problèmes de compatibilité sur vous utilisez un système de gestion de base données différent.

```
$db->createCommand('INSERT INTO `customer` (`name`) VALUES (:name)', [
    ':name' => 'Qiang',
])->execute();
```

Yii assure la prise en charge de l'enregistrement actif (*Active Record*) pour les bases de données relationnelles suivantes :

— MySQL 4.1 ou versions postérieures : via `yii\db\ActiveRecord`

22. https://fr.wikipedia.org/wiki/Active_record

— PostgreSQL 7.3 ou versions postérieures : via `yii\db\ActiveRecord`
— SQLite 2 et 3 : via `yii\db\ActiveRecord`
— Microsoft SQL Server 2008 ou versions postérieures : via `yii\db\ActiveRecord`
— Oracle : via `yii\db\ActiveRecord`
— CUBRID 9.3 ou versions postérieures : via `yii\db\ActiveRecord` (Notez que, à cause d'un bogue [23] dans l'extension CUBRID 9.3, l'entourage des valeurs par des marques de citation ne fonctionne pas, c'est pourquoi vous avez besoin de CUBRID 9.3 à la fois comme client et comme serveur)
— Sphinx : via `yii\sphinx\ActiveRecord`, requiert l'extension `yii2-sphinx`
— ElasticSearch : via `yii\elasticsearch\ActiveRecord`, requiert l'extension `yii2-elasticsearch`

De plus, Yii prend aussi en charge l'enregistrement actif (*Active Record*) avec les bases de données non SQL suivantes :

— Redis 2.6.12 ou versions postérieures : via `yii\redis\ActiveRecord`, requiert l'extension `yii2-redis`
— MongoDB 1.3.0 ou versions postérieures : via `yii\mongodb\ActiveRecord`, requiert l'extension `yii2-mongodb`

Dans ce tutoriel, nous décrivons essentiellement l'utilisation de l'enregistrement actif pour des bases de données relationnelles. Cependant, la majeure partie du contenu décrit ici est aussi applicable aux bases de données non SQL.

6.3.1 Déclaration des classes d'enregistrement actif (*Active Record*)

Pour commencer, déclarez une classe d'enregistrement actif en étendant la classe `yii\db\ActiveRecord`.

Définir un nom de table

Par défaut, chacune des classes d'enregistrement actif est associée à une table de la base de données. La méthode `tableName()` retourne le nom de la table en convertissant le nom via `yii\helpers\Inflector::camel2id()`. Vous pouvez redéfinir cette méthode si le nom de la table ne suit pas cette convention.

Un `préfixe de table` par défaut peut également être appliqué.
Par exemple, si le `préfixe de table` est `tbl_`, `Customer` devient `tbl_customer` et `OrderItem` devient `tbl_order_item`.

Si un nom de table est fourni sous la forme `{{%TableName}}`, alors le caractère `%` est remplacé par le préfixe de table. Par exemple, `{{%post}}` devient `{{tbl_post}}`. Les accolades autour du nom de table sont utilisées pour l'entourage par des marques de citation dans une requête SQL .

23. https://jira.cubrid.org/browse/APIS-658

Dans l'exemple suivant, nous déclarons une classe d'enregistrement actif nommée `Customer` pour la table de base de données `customer`.

```
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    const STATUS_INACTIVE = 0;
    const STATUS_ACTIVE = 1;

    /**
     * @return string le nom de la table associée à cette classe
d'enregistrement actif.
     */
    public static function tableName()
    {
        return 'customer';
    }
}
```

Les enregistrements actifs sont appelés "modèles"

Les instances d'une classe d'enregistrement actif (*Active Record*) sont considérées comme des modèles. Pour cette raison, nous plaçons les classes d'enregistrement actif dans l'espace de noms `app\models`(ou autres espaces de noms prévus pour contenir des classes de modèles).

Comme la classe `yii\db\ActiveRecord` étend la classe `yii\base\Model`, elle hérite de *toutes* les fonctionnalités d'un modèle, comme les attributs, les règles de validation, la sérialisation des données, etc.

6.3.2 Connexion aux bases de données

Par défaut, l'enregistrement actif utilise le composant d'application `db` en tant que `connexion à une base de données` pour accéder aux données de la base de données et les manipuler. Comme expliqué dans la section Objets d'accès aux bases de données, vous pouvez configurer le composant `db` dans la configuration de l'application comme montré ci-dessous :

```
return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
        ],
    ],
];
```

Si vous voulez utiliser une connexion de base de données autre que le composant `db`, vous devez redéfinir la méthode `getDb()` :

```
class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        // utilise le composant d'application "db2"
        return \Yii::$app->db2;
    }
}
```

6.3.3 Requête de données

Après avoir déclaré une classe d'enregistrement actif, vous pouvez l'utiliser pour faire une requête de données de la table correspondante dans la base de données. Ce processus s'accomplit en général en trois étapes :

1. Créer un nouvel objet *query* (requête) en appelant la méthode `yii\db\ActiveRecord::find()` ;
2. Construire l'objet *query* en appelant des méthodes de construction de requête ;
3. Appeler une méthode de requête pour retrouver les données en terme d'instances d'enregistrement actif.

Comme vous pouvez le voir, cela est très similaire à la procédure avec le constructeur de requêtes. La seule différence est que, au lieu d'utiliser l'opérateur `new` pour créer un objet *query* (requête), vous appelez la méthode `yii\db\ActiveRecord::find()` pour retourner un nouvel objet *query* qui est de la classe `yii\db\ActiveQuery`.

Ce-dessous, nous donnons quelques exemples qui montrent comment utiliser l'*Active Query* (requête active) pour demander des données :

```
// retourne un client (*customer*) unique dont l'identifiant est 123
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::find()
    ->where(['id' => 123])
    ->one();

// retourne tous les clients actifs et les classe par leur identifiant
// SELECT * FROM `customer` WHERE `status` = 1 ORDER BY `id`
$customers = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->orderBy('id')
    ->all();

// retourne le nombre de clients actifs
// SELECT COUNT(*) FROM `customer` WHERE `status` = 1
```

```php
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// retourne tous les clients dans un tableau indexé par l'identifiant du
client
// SELECT * FROM `customer`
$customers = Customer::find()
    ->indexBy('id')
    ->all();
```

Dans le code ci-dessus, `$customer` est un objet `Customer` tandis que `$customers` est un tableau d'objets `Customer`. Ils sont tous remplis par les données retrouvées dans la table `customer`.

> **Info :** comme la classe `yii\db\ActiveQuery` étend la classe `yii\db\Query`, vous pouvez utiliser *toutes* les méthodes de construction et de requête comme décrit dans la section sur le constructeur de requête.

Parce que faire une requête de données par les valeurs de clés primaires ou par jeu de valeurs de colonne est une tâche assez courante, Yii fournit une prise en charge de méthodes raccourcis pour cela :

- `yii\db\ActiveRecord::findOne()` : retourne une instance d'enregistrement actif remplie avec la première ligne du résultat de la requête.
- `yii\db\ActiveRecord::findAll()` : retourne un tableau d'instances d'enregistrement actif rempli avec *tous* les résultats de la requête.

Les deux méthodes acceptent un des formats de paramètres suivants :

- une valeur scalaire : la valeur est traitée comme la valeur de la clé primaire à rechercher. Yii détermine automatiquement quelle colonne est la colonne de clé primaire en lisant les informations du schéma de la base de données.
- un tableau de valeurs scalaires : le tableau est traité comme les valeurs de clé primaire désirées à rechercher.
- un tableau associatif : les clés sont les noms de colonne et les valeurs sont les valeurs de colonne désirées à rechercher. Reportez-vous au format haché pour plus de détails.

Le code qui suit montre comment ces méthodes peuvent être utilisées :

```php
// retourne un client unique dont l'identifiant est 123
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// retourne les clients dont les identifiants sont 100, 101, 123 ou 124
// SELECT * FROM `customer` WHERE `id` IN (100, 101, 123, 124)
$customers = Customer::findAll([100, 101, 123, 124]);

// retourne un client actif dont l'identifiant est 123
// SELECT * FROM `customer` WHERE `id` = 123 AND `status` = 1
```

```php
$customer = Customer::findOne([
    'id' => 123,
    'status' => Customer::STATUS_ACTIVE,
]);

// retourne tous les clients inactifs
// SELECT * FROM `customer` WHERE `status` = 0
$customers = Customer::findAll([
    'status' => Customer::STATUS_INACTIVE,
]);
```

> Attention : si vous avez besoin de passer des saisies utilisateur à ces méthodes, assurez-vous que la valeurs saisie est un scalaire ou dans le cas d'une condition tableau, assurez-vous que la structure du tableau ne peut pas être changée depuis l'extérieur :
>
> ```php
> // yii\web\Controller garantit que $id est un scalaire
> public function actionView($id)
> {
>     $model = Post::findOne($id);
>     // ...
> }
>
> // spécifier explicitement la colonne à chercher, passer un scalaire
> ou un tableau ici, aboutit à retrouver un enregistrement unique
> $model = Post::findOne(['id' => Yii::$app->request->get('id')]);
>
> // n'utilisez PAS le code suivant si possible ! Il est possible
> d'injecter une condition tableau pour filtrer par des valeurs de
> colonne arbitraires !
> $model = Post::findOne(Yii::$app->request->get('id'));
> ```
>
> **Note :** ni `yii\db\ActiveRecord::findOne()`, ni `yii\db\ActiveQuery::one()` n'ajoutent `LIMIT 1` à l'instruction SQL générée. Si votre requête peut retourner plusieurs lignes de données, vous devez appeler `limit(1)` explicitement pour améliorer la performance, p. ex., `Customer::find()->limit(1)->one()`.

En plus d'utiliser les méthodes de construction de requête, vous pouvez aussi écrire du SQL brut pour effectuer une requête de données et vous servir des résultats pour remplir des objets enregistrements actifs. Vous pouvez le faire en appelant la méthode `yii\db\ActiveRecord::findBySql()` :

```php
// retourne tous les clients inactifs
$sql = 'SELECT * FROM customer WHERE status=:status';
$customers = Customer::findBySql($sql, [':status' =>
Customer::STATUS_INACTIVE])->all();
```

N'appelez pas de méthodes de construction de requêtes supplémentaires après avoir appelé `findBySql()` car elles seront ignorées.

6.3.4 Accès aux données

Comme nous l'avons mentionné plus haut, les données extraites de la base de données servent à remplir des instances de la classe d'enregistrement actif et chacune des lignes du résultat de la requête correspond à une instance unique de la classe d'enregistrement actif. Vous pouvez accéder accéder aux valeurs des colonnes en accédant aux attributs des instances de la classe d'enregistrement actif, par exemple :

```
// "id" et "email" sont les noms des colonnes de la table "customer"
$customer = Customer::findOne(123);
$id = $customer->id;
$email = $customer->email;
```

> **Note :** les attributs de l'instance de la classe d'enregistrement actif sont nommés d'après les noms des colonnes de la table associée en restant sensibles à la casse. Yii définit automatiquement un attribut dans l'objet enregistrement actif pour chacune des colonnes de la table associée. Vous ne devez PAS déclarer à nouveau l'un quelconque des ces attributs.

Comme les attributs de l'instance d'enregistrement actif sont nommés d'après le nom des colonnes, vous pouvez vous retrouver en train d'écrire du code PHP tel que `$customer->first_name`, qui utilise le caractère (_) *souligné* pour séparer les mots dans les noms d'attributs si vos colonnes de table sont nommées de cette manière. Si vous êtes attaché à la cohérence du style de codage, vous devriez renommer vos colonnes de tables en conséquence (p. ex. en utilisant la notation en dos de chameau).

Transformation des données

Il arrive souvent que les données entrées et/ou affichées soient dans un format qui diffère de celui utilisé pour stocker les données dans la base. Par exemple, dans la base de données, vous stockez la date d'anniversaire des clients sous la forme de horodates UNIX (bien que ce soit pas une conception des meilleures), tandis que dans la plupart des cas, vous avez envie de manipuler les dates d'anniversaire sous la forme de chaînes de caractères dans le format `'YYYY/MM/DD'`. Pour le faire, vous pouvez définir des méthodes de *transformation de données* dans la classe d'enregistrement actif comme ceci :

```
class Customer extends ActiveRecord
{
    // ...

    public function getBirthdayText()
    {
        return date('Y/m/d', $this->birthday);
```

```
    }

    public function setBirthdayText($value)
    {
        $this->birthday = strtotime($value);
    }
}
```

Désormais, dans votre code PHP, au lieu d'accéder à `$customer->birthday`, vous devez accéder à `$customer->birthdayText`, ce qui vous permet d'entrer et d'afficher les dates d'anniversaire dans le format `'YYYY/MM/DD'`.

> **Conseil :** l'exemple qui précède montre une manière générique de transformer des données dans différents formats. Si vous travaillez avec des valeurs de dates, vous pouvez utiliser DateValidator et `yii\jui\DatePicker`, qui sont plus faciles à utiliser et plus puissantes.

Retrouver des données dans des tableaux

Alors que retrouver des données en termes d'objets enregistrements actifs est souple et pratique, cela n'est pas toujours souhaitable lorsque vous devez extraire une grande quantité de données à cause de l'empreinte mémoire très importante. Dans ce cas, vous pouvez retrouver les données en utilisant des tableaux PHP en appelant `asArray()` avant d'exécuter une méthode de requête :

```
// retourne tous les clients
// chacun des clients est retourné sous forme de tableau associatif
$customers = Customer::find()
    ->asArray()
    ->all();
```

> **Note :** bien que cette méthode économise de la mémoire et améliore la performance, elle est plus proche de la couche d'abstraction basse de la base de données et perd la plupart des fonctionnalités de l'objet enregistrement actif. Une distinction très importante réside dans le type de données des valeurs de colonne. Lorsque vous retournez des données dans une instance d'enregistrement actif, les valeurs des colonnes sont automatiquement typées en fonction du type réel des colonnes ; par contre, lorsque vous retournez des données dans des tableaux, les valeurs des colonnes sont des chaînes de caractères (parce qu'elles résultent de PDO sans aucun traitement), indépendamment du type réel de ces colonnes.

Retrouver des données dans des lots

Dans la section sur le constructeur de requêtes, nous avons expliqué que vous pouvez utiliser des *requêtes par lots* pour minimiser l'utilisation de la mémoire lorsque vous demandez de grandes quantités de données de la base de données. Vous pouvez utiliser la même technique avec l'enregistrement actif. Par exemple :

```
// va chercher 10 clients (customer) à la fois
foreach (Customer::find()->batch(10) as $customers) {
    // $customers est un tableau de 10 (ou moins) objets Customer
}

// va chercher 10 clients (customers) à la fois et itère sur chacun d'eux
foreach (Customer::find()->each(10) as $customer) {
    // $customer est un objet Customer
}

// requête par lots avec chargement précoce
foreach (Customer::find()->with('orders')->each() as $customer) {
    // $customer est un objet Customer avec la relation 'orders' remplie
}
```

6.3.5 Sauvegarde des données

En utilisant l'enregistrement actif, vous pouvez sauvegarder facilement les données dans la base de données en suivant les étapes suivantes :

1. Préparer une instance de la classe d'enregistrement actif
2. Assigner de nouvelles valeurs aux attributs de cette instance
3. Appeler `yii\db\ActiveRecord::save()` pour sauvegarder les données dans la base de données.

Par exemple :

```
// insère une nouvelle ligne de données
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();

// met à jour une ligne de données existante
$customer = Customer::findOne(123);
$customer->email = 'james@newexample.com';
$customer->save();
```

La méthode `save()` peut soit insérer, soit mettre à jour une ligne de données, selon l'état de l'instance de l'enregistrement actif. Si l'instance est en train d'être créée via l'opérateur `new`, appeler `save()` provoque l'insertion d'une nouvelle ligne de données ; si l'instance est le résultat d'une méthode de requête, appeler `save()` met à jour la ligne associée à l'instance.

Vous pouvez différentier les deux états d'une instance d'enregistrement actif en testant la valeur de sa propriété `isNewRecord`. Cette propriété est aussi utilisée par `save()` en interne, comme ceci :

```
public function save($runValidation = true, $attributeNames = null)
{
    if ($this->getIsNewRecord()) {
        return $this->insert($runValidation, $attributeNames);
    } else {
        return $this->update($runValidation, $attributeNames) !== false;
    }
}
```

> Astuce : vous pouvez appeler `insert()` ou `update()` directement pour insérer ou mettre à jour une ligne.

Validation des données

Comme la classe `yii\db\ActiveRecord` étend la classe `yii\base\Model`, elle partage la même fonctionnalité de validation des données. Vous pouvez déclarer les règles de validation en redéfinissant la méthode `rules()` et effectuer la validation des données en appelant la méthode `validate()`.

Lorsque vous appelez la méthode `save()`, par défaut, elle appelle automatiquement la méthode `validate()`. C'est seulement si la validation réussit, que les données sont effectivement sauvegardées ; autrement elle retourne simplement `false`, et vous pouvez tester la propriété `errors` pour retrouver les messages d'erreurs de validation.

> Astuce : si vous avez la certitude que vos données n'ont pas besoin d'être validées (p. ex. vos données proviennent de sources fiables), vous pouvez appeler `save(false)` pour omettre la validation.

Assignation massive

Comme les modèles habituels, les instances d'enregistrement actif profitent de la fonctionnalité d'assignation massive. L'utilisation de cette fonctionnalité vous permet d'assigner plusieurs attributs d'un enregistrement actif en une seule instruction PHP, comme c'est montré ci-dessous. N'oubliez cependant pas que, seuls les attributs sûrs sont assignables en masse.

```
$values = [
    'name' => 'James',
    'email' => 'james@example.com',
];

$customer = new Customer();

$customer->attributes = $values;
$customer->save();
```

Mise à jour des compteurs

C'est une tâche courante que d'incrémenter ou décrémenter une colonne dans une table de base de données. Nous appelons ces colonnes « colonnes compteurs*. Vous pouvez utiliser la méthode `updateCounters()` pour mettre à jour une ou plusieurs colonnes de comptage. Par exemple :

```
$post = Post::findOne(100);

// UPDATE `post` SET `view_count` = `view_count` + 1 WHERE `id` = 100
$post->updateCounters(['view_count' => 1]);
```

> **Note :** si vous utilisez la méthode `yii\db\ActiveRecord::save()` pour mettre à jour une colonne compteur, vous pouvez vous retrouver avec un résultat erroné car il est probable que le même compteur soit sauvegardé par de multiples requêtes qui lisent et écrivent la même valeur de compteur.

Attributs sales (*Dirty Attributes*)

Lorsque vous appelez la méthode `save()` pour sauvegarder une instance d'enregistrement actif, seuls les attributs dit *attributs sales* sont sauvegardés. Un attribut est considéré comme *sale* si sa valeur a été modifiée depuis qu'il a été chargé depuis la base de données ou sauvegardé dans la base de données le plus récemment. Notez que la validation des données est assurée sans se préoccuper de savoir si l'instance d'enregistrement actif possède des attributs sales ou pas.

L'enregistrement actif tient à jour la liste des attributs sales. Il le fait en conservant une version antérieure des valeurs d'attribut et en les comparant avec les dernières. Vous pouvez appeler la méthode `yii\db\ActiveRecord::getDirtyAttributes()` pour obtenir les attributs qui sont couramment sales. Vous pouvez aussi appeler la méthode `yii\db\ActiveRecord::markAttributeDirty()` pour marquer explicitement un attribut comme sale.

Si vous êtes intéressé par les valeurs d'attribut antérieurs à leur plus récente modification, vous pouvez appeler la méthode `getOldAttributes()` ou la méthode `getOldAttribute()`.

> **Note :** la comparaison entre les anciennes et les nouvelles valeurs est faite en utilisant l'opérateur `===` , ainsi une valeur est considérée comme sale si le type est différent même si la valeur reste la même. Cela est souvent le cas lorsque le modèle reçoit des entrées utilisateur de formulaires HTML ou chacune des valeurs est représentée par une chaîne de caractères. Pour garantir le type correct pour p. ex. des valeurs entières, vous devez appliquer un filtre de validation : `['attributeName', 'filter', 'filter'`

`=> 'intval']`. Cela fonctionne pour toutes les fonctions de transformation de type de PHP comme intval() [24], floatval() [25], boolval [26], etc...

Valeurs d'attribut par défaut

Quelques unes de vos colonnes de tables peuvent avoir des valeurs par défaut définies dans la base de données. Parfois, vous voulez peut-être pré-remplir votre formulaire Web pour un enregistrement actif à partir des valeurs par défaut. Pour éviter d'écrire les mêmes valeurs par défaut à nouveau, vous pouvez appeler la méthode `loadDefaultValues()` pour remplir les attributs de l'enregistrement actif avec les valeurs par défaut prédéfinies dans la base de données :

```
$customer = new Customer();
$customer->loadDefaultValues();
// $customer->xyz recevra la valeur par défaut déclarée lors de la
définition de la colonne « xyz » column
```

Conversion de type d'attributs

Étant peuplé par les résultats des requêtes, l'`enregistrement actif` effectue des conversions automatiques de type pour ses valeurs d'attribut, en utilisant les informations du schéma des tables de base de données. Cela permet aux données retrouvées dans les colonnes de la table et déclarées comme entiers de peupler une instance d'enregistrement actif avec des entiers PHP, les valeurs booléennes avec des valeurs booléennes, et ainsi de suite. Néanmoins, le mécanisme de conversion de type souffre de plusieurs limitation :

— Les valeurs flottantes (Float) ne sont pas converties et sont représentées par des chaînes de caractères, autrement elles pourraient perdre de la précision.
— La conversion des valeurs entières dépend de la capacité du système d'exploitation utilisé. En particulier, les valeurs de colonne déclarée comme « entier non signé », ou « grand entier » (big integer) sont converties en entier PHP seulement pour les système d'exploitation 64 bits, tandis que sur les systèmes 32 bits, elles sont représentées par des chaînes de caractères.

Notez que la conversion de type des attributs n'est effectuée que lors du peuplement d'une instance d'enregistrement actif par les résultats d'une requête. Il n'y a pas de conversion automatique pour les valeurs chargées par une requête HTTP ou définies directement par accès à des propriétés. Le schéma

24. https://www.php.net/manual/fr/function.intval.php
25. https://www.php.net/manual/fr/function.floatval.php
26. https://www.php.net/manual/fr/function.boolval.php

de table est aussi utilisé lors de la préparation des instructions SQL pour la sauvegarde de l'enregistrement actif, garantissant ainsi que les valeurs sont liées à la requête avec le type correct. Cependant, les valeurs d'attribut d'une instance d'enregistrement actif ne sont pas converties durant le processus de sauvegarde.

> Astuce : vous pouvez utiliser `yii\behaviors\AttributeTypecastBehavior` pour faciliter la conversion de type des valeurs d'attribut lors de la validation ou la sauvegarde d'un enregistrement actif.

Depuis la version 2.0.14, la classe ActiveRecord de Yii prend en charge des types de données complexe tels que JSON ou les tableaux multi-dimensionnels.

JSON dans MySQL et PostgreSQL Après le peuplement par les données, la valeur d'une colonne JSON est automatiquement décodée selon les règles de décodage standard de JSON.

Pour sauvegarder une valeur d'attribut dans une colonne de type JSON, la classe ActiveRecord crée automatiquement un objet `JsonExpression` qui est encodé en une chaîne JSON au niveau du constructeur de requête.

Tableaux dans PostgreSQL Après le peuplement par les données, les valeurs issues de colonnes de type tableau sont automatiquement décodée de la notation PgSQL en un objet `ArrayExpression`. Il met en œuvre l'interface `ArrayAccess`, ainsi pouvez-vous l'utiliser comme un tableau, ou appeler `->getValue()` pour obtenir le tableau lui-même.

Pour sauvegarder une valeur d'attribut dans une colonne de type tableau,la classe ActiveRecord crée automatiquement un objet `ArrayExpression` qui est encodé par le constructeur de requête en une chaîne PgSQL représentant le tableau.

Vous pouvez aussi utiliser des conditions pour les colonnes de type JSON :

```
$query->andWhere(['=', 'json', new ArrayExpression(['foo' => 'bar'])
```

Pour en apprendre plus sur les le système de construction d'expressions, reportez-vous à l'article Constructeur de requêtes – Ajout de conditions et d'expressions personnalisées.

Mise à jour de plusieurs lignes

Les méthodes décrites ci-dessus fonctionnent toutes sur des instances individuelles d'enregistrement actif pour insérer ou mettre à jour des lignes individuelles de table. Pour mettre à jour plusieurs lignes à la fois, vous devez appeler la méthode statique `updateAll()`.

```
// UPDATE `customer` SET `status` = 1 WHERE `email` LIKE `%@example.com%`
Customer::updateAll(['status' => Customer::STATUS_ACTIVE], ['like', 'email',
'@example.com']);
```

De façon similaire, vous pouvez appeler `updateAllCounters()` pour mettre à jour les colonnes compteurs de plusieurs lignes à la fois.

```
// UPDATE `customer` SET `age` = `age` + 1
Customer::updateAllCounters(['age' => 1]);
```

6.3.6 Suppression de données

Pour supprimer une ligne unique de données, commencez par retrouver l'instance d'enregistrement actif correspondant à cette ligne et appelez la méthode `yii\db\ActiveRecord::delete()`.

```
$customer = Customer::findOne(123);
$customer->delete();
```

Vous pouvez appeler `yii\db\ActiveRecord::deleteAll()` pour effacer plusieurs ou toutes les lignes de données. Par exemple :

```
Customer::deleteAll(['status' => Customer::STATUS_INACTIVE]);
```

> Note : agissez avec prudence lorsque vous appelez `deleteAll()` parce que cela peut effacer totalement toutes les données de votre table si vous faites une erreur en spécifiant la condition.

6.3.7 Cycles de vie de l'enregistrement actif

Il est important que vous compreniez les cycles de vie d'un enregistrement actif lorsqu'il est utilisé à des fins différentes. Lors de chaque cycle de vie, une certaine séquence d'invocation de méthodes a lieu, et vous pouvez redéfinir ces méthodes pour avoir une chance de personnaliser le cycle de vie. Vous pouvez également répondre à certains événements de l'enregistrement actif déclenchés durant un cycle de vie pour injecter votre code personnalisé. Ces événements sont particulièrement utiles lorsque vous développez des comportements d'enregistrement actif qui ont besoin de personnaliser les cycles de vie d'enregistrement actifs.

Dans l'exemple suivant, nous résumons les différents cycles de vie d'enregistrement actif et les méthodes/événements à qui il est fait appel dans ces cycles.

Cycle de vie d'une nouvelle instance

Lorsque vous créez un nouvel enregistrement actif via l'opérateur `new`, le cycle suivant se réalise :

1. Construction de la classe.
2. `init()` : déclenche un événement `EVENT_INIT`.

Cycle de vie lors d'une requête de données

Lorsque vous effectuez une requête de données via l'une des méthodes de requête, chacun des enregistrements actifs nouvellement rempli entreprend le cycle suivant :

1. Construction de la classe.
2. `init()` : déclenche un événement `EVENT_INIT`.
3. `afterFind()` : déclenche un événement `EVENT_AFTER_FIND`.

Cycle de vie lors d'une sauvegarde de données

En appelant `save()` pour insérer ou mettre à jour une instance d'enregistrement actif, le cycle de vie suivant se réalise :

1. `beforeValidate()` : déclenche un événement `EVENT_BEFORE_VALIDATE` . Si la méthode retourne `false` (faux), ou si `yii\base\ModelEvent::$isValid` est `false`, les étapes suivantes sont sautées.
2. Effectue la validation des données. Si la validation échoue, les étapes après l'étape 3 saut sautées.
3. `afterValidate()` : déclenche un événement `EVENT_AFTER_VALIDATE`.
4. `beforeSave()` : déclenche un événement `EVENT_BEFORE_INSERT` ou un événement `EVENT_BEFORE_UPDATE`. Si la méthode retourne `false` ou si `yii\base\ModelEvent::$isValid` est `false`, les étapes suivantes sont sautées.
5. Effectue l'insertion ou la mise à jour réelle.
6. `afterSave()` : déclenche un événement `EVENT_AFTER_INSERT` ou un événement `EVENT_AFTER_UPDATE`.

Cycle de vie lors d'une suppression de données

En appelant `delete()` pour supprimer une instance d'enregistrement actif, le cycle suivant se déroule :

1. `beforeDelete()` : déclenche un événement `EVENT_BEFORE_DELETE`. Si la méthode retourne `false` ou si `yii\base\ModelEvent::$isValid` est `false`, les étapes suivantes sont sautées.
2. Effectue la suppression réelle des données.
3. `afterDelete()` : déclenche un événement `EVENT_AFTER_DELETE`.

   Note : l'appel de l'une des méthodes suivantes n'initie AUCUN des cycles vus ci-dessus parce qu'elles travaillent directement sur la base de données et pas sur la base d'un enregistrement actif :

   — `yii\db\ActiveRecord::updateAll()`
   — `yii\db\ActiveRecord::deleteAll()`
   — `yii\db\ActiveRecord::updateCounters()`
   — `yii\db\ActiveRecord::updateAllCounters()`

Cycle de vie lors du rafraîchissement des données

En appelant `refresh()` pour rafraîchir une instance d'enregistrement actif, l'événement `EVENT_AFTER_REFRESH` est déclenché si le rafraîchissement réussit et si la méthode retourne `true`.

6.3.8 Travail avec des transactions

Il y a deux façons d'utiliser les transactions lorsque l'on travaille avec un enregistrement actif.

La première façon consiste à enfermer explicitement les appels des différents méthodes dans un bloc transactionnel, comme ci-dessous :

```
$customer = Customer::findOne(123);

Customer::getDb()->transaction(function($db) use ($customer) {
    $customer->id = 200;
    $customer->save();
    // ...autres opérations de base de données...
});

// ou en alternative

$transaction = Customer::getDb()->beginTransaction();
try {
    $customer->id = 200;
    $customer->save();
    // ...other DB operations...
    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

> Note : dans le code précédent, nous utilisons deux blocs de capture pour être compatible avec PHP 5.x et PHP 7.x. `\Exception` met en œuvre l'interface `\Throwable`[27] à partir de PHP 7.0, c'est pourquoi vous pouvez sauter la partie avec `\Exception` si votre application utilise PHP 7.0 ou une version plus récente.

La deuxième façon consiste à lister les opérations de base de données qui nécessitent une prise en charge transactionnelle dans la méthode `yii\db\ActiveRecord::transactions()`. Par exemple :

```
class Customer extends ActiveRecord
{
```

27. https://www.php.net/manual/fr/class.throwable.php

```
    public function transactions()
    {
        return [
            'admin' => self::OP_INSERT,
            'api' => self::OP_INSERT | self::OP_UPDATE | self::OP_DELETE,
            // ce qui précède est équivalent à ce qui suit :
            // 'api' => self::OP_ALL,
        ];
    }
}
```

La méthode `yii\db\ActiveRecord::transactions()` doit retourner un tableau dont les clés sont les noms de scénario et les valeurs les opérations correspondantes qui doivent être enfermées dans des transactions. Vous devez utiliser les constantes suivantes pour faire référence aux différentes opérations de base de données :

— `OP_INSERT` : opération d'insertion réalisée par `insert()` ;
— `OP_UPDATE` : opération de mise à jour réalisée par `update()` ;
— `OP_DELETE` : opération de suppression réalisée par `delete()`.

Utilisez l'opérateur `|` pour concaténer les constantes précédentes pour indiquer de multiples opérations. Vous pouvez également utiliser la constante raccourci `OP_ALL` pour faire référence à l'ensemble des trois opération ci-dessus.

Les transactions qui sont créées en utilisant cette méthode sont démarrées avant d'appeler `beforeSave()` et sont entérinées après que la méthode `afterSave()` a été exécutée.

6.3.9 Verrous optimistes

Le verrouillage optimiste est une manière d'empêcher les conflits qui peuvent survenir lorsqu'une même ligne de données est mise à jour par plusieurs utilisateurs. Par exemple, les utilisateurs A et B sont tous deux, simultanément, en train de modifier le même article de wiki. Après que l'utilisateur A a sauvegardé ses modifications, l'utilisateur B clique sur le bouton « Sauvegarder » dans le but de sauvegarder ses modifications lui aussi. Comme l'utilisateur B est en train de travailler sur une version périmée de l'article, il serait souhaitable de disposer d'un moyen de l'empêcher de sauvegarder sa version de l'article et de lui montrer un message d'explication.

Le verrouillage optimiste résout le problème évoqué ci-dessus en utilisant une colonne pour enregistrer le numéro de version de chacune des lignes. Lorsqu'une ligne est sauvegardée avec un numéro de version périmée, une exception `yii\db\StaleObjectException` est levée, ce qui empêche la sauvegarde de la ligne. Le verrouillage optimiste, n'est seulement pris en charge que lorsque vous mettez à jour ou supprimez une ligne de données existante en utilisant les méthodes `yii\db\ActiveRecord::update()` ou `yii\db\ActiveRecord::delete()`,respectivement.

Pour utiliser le verrouillage optimiste :

1. Créez une colonne dans la table de base de données associée à la classe d'enregistrement actif pour stocker le numéro de version de chacune des lignes. La colonne doit être du type *big integer* (dans MySQL ce doit être `BIGINT DEFAULT 0`).
2. Redéfinissez la méthode `yii\db\ActiveRecord::optimisticLock()` pour qu'elle retourne le nom de cette colonne.
3. Dans la classe de votre modèle, mettez en œuvre `OptimisticLockBehavior` pour analyser automatiquement sa valeur des requêtes reçues.
4. Dans le formulaire Web qui reçoit les entrées de l'utilisateur, ajoutez un champ caché pour stocker le numéro de version courant de la ligne en modification. Retirez l'attribut version des règles de validation étant donné que `OptimisticLockBehavior` s'en charge.
5. Dans l'action de contrôleur qui met la ligne à jour en utilisant l'enregistrement actif, utiliser une structure *try-catch* pour l'exception `yii\db\StaleObjectException`. Mettez en œuvre la logique requise (p. ex. fusionner les modifications, avertir des données douteuses) pour résoudre le conflit. Par exemple, supposons que la colonne du numéro de version est nommée `version`. Vous pouvez mettre en œuvre le verrouillage optimiste avec un code similaire au suivant :

```
// ------ view code -------

use yii\helpers\Html;

// ...other input fields
echo Html::activeHiddenInput($model, 'version');


// ------ controller code -------

use yii\db\StaleObjectException;

public function actionUpdate($id)
{
    $model = $this->findModel($id);

    try {
        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {
            return $this->render('update', [
                'model' => $model,
            ]);
        }
    } catch (StaleObjectException $e) {
        // logique pour résoudre le  conflict
    }
```

```
}

// ------ model code -------

use yii\behaviors\OptimisticLockBehavior;

public function behaviors()
{
    return [
        OptimisticLockBehavior::class,
    ];
}

public function optimisticLock()
{
    return 'version';
}
```

> Note : comme `OptimisticLockBehavior` garantit que l'enregistrement n'est sauvegardé que si l'utilisateur soumet un numéro de version valide en analysant directement `getBodyParam()`, il peut être utile d'étendre votre classe de modèle et de réaliser l'étape 2 dans le modèle du parent lors de l'attachement du comportement (étape 3) à la classe enfant ; ainsi vous pouvez disposer d'une instance dédiée à l'usage interne tout en liant l'autre aux contrôleurs chargés de recevoir les entrées de l'utilisateur final. En alternative, vous pouvez mettre en œuvre votre propre logique en configurant sa propriété `value`.

6.3.10 Travail avec des données relationnelles

En plus de travailler avec des tables de base de données individuelles, l'enregistrement actif permet aussi de rassembler des données en relation, les rendant ainsi immédiatement accessibles via les données primaires. Par exemple, la donnée client est en relation avec les données commandes parce qu'un client peut avoir passé une ou plusieurs commandes. Avec les déclarations appropriées de cette relation, vous serez capable d'accéder aux commandes d'un client en utilisant l'expression `$customer->orders` qui vous renvoie les informations sur les commandes du client en terme de tableau d'instances `Order` (Commande) d'enregistrement actif.

Déclaration de relations

Pour travailler avec des données relationnelles en utilisant l'enregistrement actif, vous devez d'abord déclarer les relations dans les classes d'enregistrement actif. La tâche est aussi simple que de déclarer une *méthode de relation* pour chacune des relations concernées, comme ceci :

```php
class Customer extends ActiveRecord
{
    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}

class Order extends ActiveRecord
{
    // ...

    public function getCustomer()
    {
        return $this->hasOne(Customer::class, ['id' => 'customer_id']);
    }
}
```

Dans le code ci-dessus, nous avons déclaré une relation `orders` (commandes) pour la classe `Customer` (client), et une relation `customer` (client) pour la classe `Order` (commande).

Chacune des méthodes de relation doit être nommée sous la forme `getXyz`. Nous appelons `xyz` (la première lettre est en bas de casse) le *nom de la relation*. Notez que les noms de relation sont *sensibles à la casse*.

En déclarant une relation, vous devez spécifier les informations suivantes :

— la multiplicité de la relation : spécifiée en appelant soit la méthode `hasMany()`, soit la méthode `hasOne()`. Dans l'exemple ci-dessus vous pouvez facilement déduire en lisant la déclaration des relations qu'un client a beaucoup de commandes, tandis qu'une commande n'a qu'un client.

— le nom de la classe d'enregistrement actif : spécifié comme le premier paramètre de `hasMany()` ou de `hasOne()`. Une pratique conseillée est d'appeler `Xyz::class` pour obtenir la chaîne de caractères représentant le nom de la classe de manière à bénéficier de l'auto-complètement de l'EDI et de la détection d'erreur dans l'étape de compilation.

— Le lien entre les deux types de données : spécifie le(s) colonne(s) via lesquelles les deux types de données sont en relation. Les valeurs du tableau sont les colonnes des données primaires (représentées par la classe d'enregistrement actif dont vous déclarez les relations), tandis que les clés sont les colonnes des données en relation.

Une règle simple pour vous rappeler cela est, comme vous le voyez dans l'exemple ci-dessus, d'écrire la colonne qui appartient à l'enregistrement actif en relation juste à coté de lui. Vous voyez là que l'identifiant du client (`customer_id`) est une propriété de `Order` et `id` est une propriété de `Customer`.

Accès aux données relationnelles

Après avoir déclaré des relations, vous pouvez accéder aux données relationnelles via le nom des relations. Tout se passe comme si vous accédiez à une propriété d'un objet défini par la méthode de relation. Pour cette raison, nous appelons cette propriété *propriété de relation*. Par exemple :

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
// $orders est un tableau d'objets Order
$orders = $customer->orders;
```

> Info : lorsque vous déclarez une relation nommée `xyz` via une méthode d'obtention `getXyz()`, vous êtes capable d'accéder à `xyz` comme à un objet property. Notez que le nom est sensible à la casse.

Si une relation est déclarée avec la méthode `hasMany()`, l'accès à cette propriété de relation retourne un tableau des instances de l'enregistrement actif en relation ; si une relation est déclarée avec la méthode `hasOne()`, l'accès à la propriété de relation retourne l'instance de l'enregistrement actif en relation, ou `null` si aucune donnée en relation n'est trouvée.

Lorsque vous accédez à une propriété de relation pour la première fois, une instruction SQL est exécutée comme le montre l'exemple précédent. Si la même propriété fait l'objet d'un nouvel accès, le résultat précédent est retourné sans exécuter à nouveau l'instruction SQL. Pour forcer l'exécution à nouveau de l'instruction SQL, vous devez d'abord annuler la définition de la propriété de relation : `unset($customer->orders)`.

> Note : bien que ce concept semble similaire à la fonctionnalité propriété d'objet, il y a une différence importante. Pour les propriétés normales d'objet, la valeur est du même type que la méthode d'obtention de définition. Une méthode de relation cependant retourne toujours une instance d'`yii\db\ActiveRecord` ou un tableau de telles instances.
>
> ```
> $customer->orders; // est un tableau d'objets `Order`
> $customer->getOrders(); // retourne une instance d'ActiveQuery
> ```
>
> Cela est utile for créer des requêtes personnalisées, ce qui est décrit dans la section suivante.

Requête relationnelle dynamique

Parce qu'une méthode de relation retourne une instance d'`yii\db\ActiveQuery`, vous pouvez continuer à construire cette requête en utilisant les méthodes de construction avant de l'exécuter. Par exemple :

```php
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 200
ORDER BY `id`
$orders = $customer->getOrders()
    ->where(['>', 'subtotal', 200])
    ->orderBy('id')
    ->all();
```

Contrairement à l'accès à une propriété de relation, chaque fois que vous effectuez une requête relationnelle dynamique via une méthode de relation, une instruction SQL est exécutée, même si la même requête relationnelle dynamique a été effectuée auparavant.

Parfois, vous voulez peut-être paramétrer une déclaration de relation de manière à ce que vous puissiez effectuer des requêtes relationnelles dynamiques plus facilement. Par exemple, vous pouvez déclarer une relation `bigOrders` comme ceci :,

```php
class Customer extends ActiveRecord
{
    public function getBigOrders($threshold = 100)
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id'])
            ->where('subtotal > :threshold', [':threshold' => $threshold])
            ->orderBy('id');
    }
}
```

Par la suite, vous serez en mesure d'effectuer les requêtes relationnelles suivantes :

```php
// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 200
ORDER BY `id`
$orders = $customer->getBigOrders(200)->all();

// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 100
ORDER BY `id`
$orders = $customer->bigOrders;
```

Relations via une table de jointure

Dans la modélisation de base de données, lorsque la multiplicité entre deux tables en relation est *many-to-many* (de plusieurs à plusieurs), une table de jointure[28] est en général introduite. Par exemple, la table `order` (commande) et la table `item` peuvent être en relation via une table de jointure nommée `order_item` (item_de_commande). Une commande correspond ensuite à de multiples items de commande, tandis qu'un item de produit correspond lui-aussi à de multiples items de commande (*order items*).

28. https://en.wikipedia.org/wiki/Junction_table

Lors de la déclaration de telles relations, vous devez appeler soit `via()`, soit `viaTable()`, pour spécifier la table de jointure. La différence entre `via()` et `viaTable()` est que la première spécifie la table de jointure en termes de noms de relation existante, tandis que la deuxième utilise directement la table de jointure. Par exemple :

```
class Order extends ActiveRecord
{
    public function getItems()
    {
        return $this->hasMany(Item::class, ['id' => 'item_id'])
            ->viaTable('order_item', ['order_id' => 'id']);
    }
}
```

ou autrement,

```
class Order extends ActiveRecord
{
    public function getOrderItems()
    {
        return $this->hasMany(OrderItem::class, ['order_id' => 'id']);
    }

    public function getItems()
    {
        return $this->hasMany(Item::class, ['id' => 'item_id'])
            ->via('orderItems');
    }
}
```

L'utilisation de relations déclarées avec une table de jointure est la même que celle de relations normales. Par exemple :

```
// SELECT * FROM `order` WHERE `id` = 100
$order = Order::findOne(100);

// SELECT * FROM `order_item` WHERE `order_id` = 100
// SELECT * FROM `item` WHERE `item_id` IN (...)
// retourne un tableau d'objets Item
$items = $order->items;
```

Chaînage de définitions de relation via de multiples tables

Il est de plus possible de définir des relations via de multiples tables en chaînant les définitions de relation en utilisant `via()`. En reprenant l'exemple ci-dessus, nous avons les classes `Customer`, `Order` et `Item`. Nous pouvons ajouter une relation à la classe `Customer` qui liste tous les items de tous les commandes qu'ils ont passées, et la nommer `getPurchasedItems()`, le chaînage de relations est présenté dans l'exemple de code suivant :

```php
class Customer extends ActiveRecord
{
    // ...

    public function getPurchasedItems()
    {
        // items de clients pour lesquels la colonne 'id' de `Item`
        correspond à  'item_id' dans OrderItem
        return $this->hasMany(Item::class, ['id' => 'item_id'])
                    ->via('orderItems');
    }

    public function getOrderItems()
    {
        // items de commandes clients pour lesquels, la colonne 'id' de
        `Order` correspond à 'order_id' dans OrderItem
        return $this->hasMany(OrderItem::class, ['order_id' => 'id'])
                    ->via('orders');
    }

    public function getOrders()
    {
        // idem à ci-dessus
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}
```

Chargement paresseux et chargement précoce

Dans la sous-section Accès aux données relationnelles, nous avons expliqué que vous pouvez accéder à une propriété de relation d'une instance d'enregistrement actif comme si vous accédiez à une propriété normale d'objet. Une instruction SQL est exécutée seulement lorsque vous accédez à cette propriété pour la première fois. Nous appelons une telle méthode d'accès à des données relationnelles, *chargement paresseux*. Par exemple :

```php
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$orders = $customer->orders;

// pas de SQL exécuté
$orders2 = $customer->orders;
```

Le chargement paresseux est très pratique à utiliser. Néanmoins, il peut souffrir d'un problème de performance lorsque vous avez besoin d'accéder à la même propriété de relation sur de multiples instances d'enregistrement actif. Examinons l'exemple de code suivant. Combien d'instruction SQL sont-elles exécutées ?

```php
// SELECT * FROM `customer` LIMIT 100
$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {
    // SELECT * FROM `order` WHERE `customer_id` = ...
    $orders = $customer->orders;
}
```

Comme vous pouvez le constater dans le fragment de code ci-dessus, 101 instruction SQL sont exécutées ! Cela tient au fait que, à chaque fois que vous accédez à la propriété de relation `orders` d'un objet client différent dans la boucle for, une instruction SQL est exécutée.

Pour résoudre ce problème de performance, vous pouvez utiliser ce qu'on appelle le *chargement précoce* comme montré ci-dessous :

```php
// SELECT * FROM `customer` LIMIT 100;
// SELECT * FROM `orders` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->with('orders')
    ->limit(100)
    ->all();

foreach ($customers as $customer) {
    // aucune instruction SQL exécutée
    $orders = $customer->orders;
}
```

En appelant `yii\db\ActiveQuery::with()`, vous donner comme instruction à l'enregistrement actif de rapporter les commandes (*orders*) pour les 100 premiers clients (*customers*) en une seule instruction SQL. En conséquence, vous réduisez le nombre d'instructions SQL de 101 à 2 !

Vous pouvez charger précocement une ou plusieurs relations. Vous pouvez même charger précocement des *relations imbriquées*. Une relation imbriquée est une relation qui est déclarée dans une classe d'enregistrement actif. Par exemple, `Customer` est en relation avec `Order` via la relation `orders`, et `Order` est en relation avec `Item` via la relation `items`. Lorsque vous effectuez une requête pour `Customer`, vous pouvez charger précocement `items` en utilisant la notation de relation imbriquée `orders.items`.

Le code suivant montre différentes utilisations de `with()`. Nous supposons que la classe `Customer` possède deux relations `orders` (commandes) et `country` (pays), tandis que la classe `Order` possède une relation `items`.

```php
// chargement précoce à la fois de "orders" et de "country"
$customers = Customer::find()->with('orders', 'country')->all();
// équivalent au tableau de syntaxe ci-dessous
$customers = Customer::find()->with(['orders', 'country'])->all();
// aucune instruction SQL exécutée
$orders= $customers[0]->orders;
// aucune instruction SQL exécutée
```

```php
$country = $customers[0]->country;

// chargement précoce de "orders" et de la relation imbriquée
"orders.items"
$customers = Customer::find()->with('orders.items')->all();
// accés aux items de la première commande du premier client
// aucune instruction SQL exécutée
$items = $customers[0]->orders[0]->items;
```

Vous pouvez charger précocement des relations imbriquées en profondeur, telles que `a.b.c.d`. Toutes les relations parentes sont chargées précocement. C'est à dire, que lorsque vous appelez `with()` en utilisant `a.b.c.d`, vous chargez précocement `a`, `a.b`, `a.b.c` et `a.b.c.d`.

> Info : en général, lors du chargement précoce de `N` relations parmi lesquelles `M` relations sont définies par une table de jointure, `N+M+1` instructions SQL sont exécutées au total. Notez qu'une relation imbriquée `a.b.c.d` possède 4 relations.

Lorsque vous chargez précocement une relation, vous pouvez personnaliser le requête relationnelle correspondante en utilisant une fonction anonyme. Par exemple :

```php
// trouve les clients et rapporte leur pays et leurs commandes actives
// SELECT * FROM `customer`
// SELECT * FROM `country` WHERE `id` IN (...)
// SELECT * FROM `order` WHERE `customer_id` IN (...) AND `status` = 1
$customers = Customer::find()->with([
    'country',
    'orders' => function ($query) {
        $query->andWhere(['status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

Lors de la personnalisation de la requête relationnelle pour une relation, vous devez spécifier le nom de la relation comme une clé de tableau et utiliser une fonction anonyme comme valeur de tableau correspondante. La fonction anonyme accepte une paramètre `$query` qui représente l'objet `yii\db\ActiveQuery` utilisé pour effectuer la requête relationnelle pour la relation. Dans le code ci-dessus, nous modifions la requête relationnelle en ajoutant une condition additionnelle à propos de l'état de la commande (*order*).

> Note : si vous appelez `select()` tout en chargeant précocement les relations, vous devez vous assurer que les colonnes référencées dans la déclaration de la relation sont sélectionnées. Autrement, les modèles en relation peuvent ne pas être chargés correctement. Par exemple :

```
$orders = Order::find()->select(['id',
'amount'])->with('customer')->all();
// $orders[0]->customer est toujours nul. Pour régler le problème,
vous devez faire ce qui suit :
$orders = Order::find()->select(['id', 'amount',
'customer_id'])->with('customer')->all();
```

Jointure avec des relations

> Note : le contenu décrit dans cette sous-section ne s'applique qu'aux bases de données relationnelles, telles que MySQL, PostgreSQL, etc.

Les requêtes relationnelles que nous avons décrites jusqu'à présent ne font référence qu'aux colonnes de table primaires lorsque nous faisons une requête des données primaires. En réalité, nous avons souvent besoin de faire référence à des colonnes dans les tables en relation. Par exemple, vous désirez peut-être rapporter les clients qui ont au moins une commande active. Pour résoudre ce problème, nous pouvons construire une requête avec jointure comme suit :

```
// SELECT `customer`.* FROM `customer`
// LEFT JOIN `order` ON `order`.`customer_id` = `customer`.`id`
// WHERE `order`.`status` = 1
//
// SELECT * FROM `order` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->select('customer.*')
    ->leftJoin('order', '`order`.`customer_id` = `customer`.`id`')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->with('orders')
    ->all();
```

> Note : il est important de supprimer les ambiguïtés sur les noms de colonnes lorsque vous construisez les requêtes relationnelles faisant appel à des instructions SQL JOIN. Une pratique courante est de préfixer les noms de colonnes par le nom des tables correspondantes.

Néanmoins, une meilleure approche consiste à exploiter les déclarations de relations existantes en appelant `yii\db\ActiveQuery::joinWith()` :

```
$customers = Customer::find()
    ->joinWith('orders')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->all();
```

Les deux approches exécutent le même jeu d'instructions SQL. La deuxième approche est plus propre et plus légère cependant.

Par défaut, `joinWith()` utilise `LEFT JOIN` pour joindre la table primaire avec les tables en relation. Vous pouvez spécifier une jointure différente (p .ex.

`RIGHT JOIN`) via son troisième paramètre `$joinType`. Si le type de jointure que vous désirez est `INNER JOIN`, vous pouvez simplement appeler `innerJoinWith()`, à la place.

L'appel de `joinWith()` charge précocement les données en relation par défaut. Si vous ne voulez pas charger les données en relation, vous pouvez spécifier son deuxième paramètre `$eagerLoading` comme étant `false`.

> Note : même en utilisant `joinWith()` ou `innerJoinWith()` avec le chargement précoce activé les données en relation ne sont **pas** peuplées à partir du résultat de la requête `JOIN`. C'est pourquoi il y a toujours une requête supplémetaire pour chacune des relations jointes comme expliqué à la section chargement précoce.

Comme avec `with()`, vous pouvez joindre une ou plusieurs relations ; vous pouvez personnaliser les requêtes de relation à la volée ; vous pouvez joindre des relations imbriquées ; et vous pouvez mélanger l'utilisation de `with()` et celle de `joinWith()`. Par exemple :

```
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->andWhere(['>', 'subtotal', 100]);
    },
])->with('country')
    ->all();
```

Parfois, en joignant deux tables, vous désirez peut-être spécifier quelques conditions supplémentaires dans la partie `ON` de la requête JOIN. Cela peut être réalisé en appelant la méthode `yii\db\ActiveQuery::onCondition()` comme ceci :

```
// SELECT `customer`.* FROM `customer`
// LEFT JOIN `order` ON `order`.`customer_id` = `customer`.`id` AND
`order`.`status` = 1
//
// SELECT * FROM `order` WHERE `customer_id` IN (...)
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->onCondition(['order.status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

La requête ci-dessus retourne *tous* les clients, et pour chacun des clients, toutes les commandes actives. Notez que cela est différent de notre exemple précédent qui ne retourne que les clients qui ont au moins une commande active.

> Info : quand `yii\db\ActiveQuery` est spécifiée avec une condition via une jointure `onCondition()`, la condition est placée dans la partie `ON` si la requête fait appel à une requête JOIN. Si la

requête ne fait pas appel à JOIN, la *on-condition* est automatiquement ajoutée à la partie `WHERE` de la requête. Par conséquent elle peut ne contenir que des conditions incluant des colonnes de la table en relation.

Alias de table de relation Comme noté auparavant, lorsque vous utilisez une requête JOIN, vous devez lever les ambiguïtés sur le nom des colonnes. Pour cela, un alias est souvent défini pour une table. Définir un alias pour la requête relationnelle serait possible en personnalisant le requête de relation de la manière suivante :

```
$query->joinWith([
    'orders' => function ($q) {
        $q->from(['o' => Order::tableName()]);
    },
])
```

Cela paraît cependant très compliqué et implique soit de coder en dur les noms de tables des objets en relation, soit d'appeler `Order::tableName()`. Depuis la version 2.0.7, Yii fournit un raccourci pour cela. Vous pouvez maintenant définir et utiliser l'alias pour la table de relation comme ceci :

```
// joint la relation orders et trie les résultats par orders.id
$query->joinWith(['orders o'])->orderBy('o.id');
```

La syntaxe ci-dessus ne fonctionne que pour des relations simples. Si vous avez besoin d'un alias pour une table intermédiaire lors de la jointure via des relations imbriquées, p. ex. `$query->joinWith(['orders.product'])`, vous devez imbriquer les appels joinWith comme le montre l'exemple suivant :

```
$query->joinWith(['orders o' => function($q) {
        $q->joinWith('product p');
    }])
    ->where('o.amount > 100');
```

Relations inverses

Les déclarations de relations sont souvent réciproques entre deux classes d'enregistrement actif. Par exemple, `Customer` est en relation avec `Order` via la relation `orders`, et `Order` est en relation inverse avec `Customer` via la relation `customer`.

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
```

```
}

class Order extends ActiveRecord
{
    public function getCustomer()
    {
        return $this->hasOne(Customer::class, ['id' => 'customer_id']);
    }
}
```

Considérons maintenant ce fragment de code :

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// SELECT * FROM `customer` WHERE `id` = 123
$customer2 = $order->customer;

// displays "not the same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

On aurait tendance à penser que `$customer` et `$customer2` sont identiques, mais ils ne le sont pas ! En réalité, ils contiennent les mêmes données de client, mais ce sont des objets différents. En accédant à `$order->customer`, une instruction SQL supplémentaire est exécutée pour remplir un nouvel objet `$customer2`.

Pour éviter l'exécution redondante de la dernière instruction SQL dans l'exemple ci-dessus, nous devons dire à Yii que `customer` est une *relation inverse* de `orders` en appelant la méthode `inverseOf()` comme ci-après :

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' =>
        'id'])->inverseOf('customer');
    }
}
```

Avec cette déclaration de relation modifiée, nous avons :

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// aucune instruction SQL n'est exécutée
$customer2 = $order->customer;

// affiche "same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

> Note : les relations inverses ne peuvent être définies pour des relations faisant appel à une table de jointure. C'est à dire que, si une relation est définie avec `via()` ou avec `viaTable()`, vous ne devez pas appeler `inverseOf()` ensuite.

6.3.11 Sauvegarde des relations

En travaillant avec des données relationnelles, vous avez souvent besoin d'établir de créer des relations entre différentes données ou de supprimer des relations existantes. Cela requiert de définir les valeurs appropriées pour les colonnes qui définissent ces relations. En utilisant l'enregistrement actif, vous pouvez vous retrouver en train d'écrire le code de la façon suivante :

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

// défninition de l'attribut qui définit la relation "customer" dans Order
$order->customer_id = $customer->id;
$order->save();
```

L'enregistrement actif fournit la méthode `link()` qui vous permet d'accomplir cette tâche plus élégamment :

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

$order->link('customer', $customer);
```

La méthode `link()` requiert que vous spécifiiez le nom de la relation et l'instance d'enregistrement actif cible avec laquelle le relation doit être établie. La méthode modifie les valeurs des attributs qui lient deux instances d'enregistrement actif et les sauvegardes dans la base de données. Dans l'exemple ci-dessus, elle définit l'attribut `customer_id` de l'instance `Order` comme étant la valeur de l'attribut `id` de l'instance `Customer` et le sauvegarde ensuite dans la base de données.

> Note : vous ne pouvez pas lier deux instances d'enregistrement actif nouvellement créées.

L'avantage d'utiliser `link()` est même plus évident lorsqu'une relation est définie via une table de jointure. Par exemple, vous pouvez utiliser le code suivant pour lier une instance de `Order` à une instance de `Item` :

```
$order->link('items', $item);
```

Le code ci-dessus insère automatiquement une ligne dans la table de jointure `order_item` pour mettre la commande en relation avec l'item.

> Info : la méthode `link()` n'effectue AUCUNE validation de données lors de la sauvegarde de l'instance d'enregistrement actif affectée. Il est de votre responsabilité de valider toutes les données entrées avant d'appeler cette méthode.

L'opération opposée à `link()` est `unlink()` qui casse une relation existante entre deux instances d'enregistrement actif. Par exemple :

```
$customer = Customer::find()->with('orders')->where(['id' => 123])->one();
$customer->unlink('orders', $customer->orders[0]);
```

Par défaut, la méthode `unlink()` définit la valeur de la (des) clé(s) étrangères qui spécifie(nt) la relation existante à `null`. Vous pouvez cependant, choisir de supprimer la ligne de la table qui contient la valeur de clé étrangère en passant à la méthode la valeur `true` pour le paramètre `$delete`.

Lorsqu'une table de jointure est impliquée dans une relation, appeler `unlink()` provoque l'effacement des clés étrangères dans la table de jointure, ou l'effacement de la ligne correspondante dans la table de jointure si `#delete` vaut `true`.

6.3.12 Relations inter bases de données

L'enregistrement actif vous permet de déclarer des relations entre les classes d'enregistrement actif qui sont mise en œuvre par différentes bases de données. Les bases de données peuvent être de types différents (p. ex. MySQL and PostgreSQL, ou MS SQL et MongoDB), et elles peuvent s'exécuter sur des serveurs différents. Vous pouvez utiliser la même syntaxe pour effectuer des requêtes relationnelles. Par exemple :

```
// Customer est associé à la table "customer" dans la base de données
relationnelle (e.g. MySQL)
class Customer extends \yii\db\ActiveRecord
{
    public static function tableName()
    {
        return 'customer';
    }

    public function getComments()
    {
        // a customer has many comments
        return $this->hasMany(Comment::class, ['customer_id' => 'id']);
    }
}

// Comment est associé à la collection "comment" dans une base de données
MongoDB
```

```php
class Comment extends \yii\mongodb\ActiveRecord
{
    public static function collectionName()
    {
        return 'comment';
    }

    public function getCustomer()
    {
        // un commentaire (comment) a un client (customer)
        return $this->hasOne(Customer::class, ['id' => 'customer_id']);
    }
}

$customers = Customer::find()->with('comments')->all();
```

Vous pouvez utiliser la plupart des fonctionnalités de requêtes relationnelles qui ont été décrites dans cette section.

> Note : l'utilisation de `joinWith()` est limitée aux bases de données qui permettent les requête JOIN inter bases. Pour cette raison, vous ne pouvez pas utiliser cette méthode dans l'exemple ci-dessus car MongoDB ne prend pas JOIN en charge.

6.3.13 Personnalisation des classes de requête

Par défaut, toutes les requêtes d'enregistrement actif sont prises en charge par `yii\db\ActiveQuery`. Pour utiliser une classe de requête personnalisée dans une classe d'enregistrement actif, vous devez redéfinir la méthode `yii\db\ActiveRecord::find()` et retourner une instance de votre classe de requête personnalisée . Par exemple :

```php
namespace app\models;

use yii\db\ActiveRecord;
use yii\db\ActiveQuery;

class Comment extends ActiveRecord
{
    public static function find()
    {
        return new CommentQuery(get_called_class());
    }
}
```

Désormais, à chaque fois que vous effectuez une requête (p. ex. `find()`, `findOne()`) ou définissez une relation (p. ex. `hasOne()`) avec `Comment`, vous travaillez avec une instance de `CommentQuery` au lieu d'une instance d'`ActiveQuery`.

Vous devez maintenant définir la classe `CommentQuery`, qui peut être personnalisée de maintes manières créatives pour améliorer votre expérience de la construction de requête. Par exemple :

```php
// fichier CommentQuery.php
namespace app\models;

use yii\db\ActiveQuery;

class CommentQuery extends ActiveQuery
{
    // conditions ajoutées par défaut (peut être sauté)
    public function init()
    {
        $this->andOnCondition(['deleted' => false]);
        parent::init();
    }

    // ... ajoutez vos méthodes de requêtes personnalisées ici ...

    public function active($state = true)
    {
        return $this->andOnCondition(['active' => $state]);
    }
}
```

> Note : au lieu d'appeler `onCondition()`, vous devez généralement appeler `andOnCondition()` ou `orOnCondition()` pour ajouter des conditions supplémentaires lors de la définition de nouvelles méthodes de requête de façon à ce que aucune condition existante en soit redéfinie.

Cela vous permet d'écrire le code de construction de requête comme suit :

```php
$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();
```

> Astuce : dans les gros projets, il est recommandé que vous utilisiez des classes de requête personnalisées pour contenir la majeure partie de code relatif aux requêtes de manière à ce que les classe d'enregistrement actif puissent être maintenues propres.

Vous pouvez aussi utiliser les méthodes de construction de requêtes en définissant des relations avec `Comment` ou en effectuant une requête relationnelle :

```php
class Customer extends \yii\db\ActiveRecord
{
    public function getActiveComments()
    {
        return $this->hasMany(Comment::class, ['customer_id' =>
        'id'])->active();
    }
}

$customers = Customer::find()->joinWith('activeComments')->all();
```

```php
// ou en alternative
class Customer extends \yii\db\ActiveRecord
{
    public function getComments()
    {
        return $this->hasMany(Comment::class, ['customer_id' => 'id']);
    }
}

$customers = Customer::find()->joinWith([
    'comments' => function($q) {
        $q->active();
    }
])->all();
```

> Info : dans Yii 1.1, il existe un concept appelé *scope*. Scope n'est plus pris en charge directement par Yii 2.0, et vous devez utiliser des classes de requête personnalisées et des méthodes de requêtes pour remplir le même objectif.

6.3.14 Sélection de champs supplémentaires

Quand un enregistrement actif est rempli avec les résultats d'une requête, ses attributs sont remplis par les valeurs des colonnes correspondantes du jeu de données reçu.

Il vous est possible d'aller chercher des colonnes ou des valeurs additionnelles à partir d'une requête et des les stocker dans l'enregistrement actif. Par exemple, supposons que nous ayons une table nommée `room`, qui contient des informations sur les chambres (rooms) disponibles dans l'hôtel. Chacune des chambres stocke des informations sur ses dimensions géométriques en utilisant des champs `length` (longueur), `width` (largeur) , `height` (hauteur). Imaginons que vous ayez besoin de retrouver une liste des chambres disponibles en les classant par volume décroissant. Vous ne pouvez pas calculer le volume en PHP parce que vous avez besoin de trier les enregistrements par cette valeur, mais vous voulez peut-être aussi que `volume` soit affiché dans la liste. Pour atteindre ce but, vous devez déclarer un champ supplémentaire dans la classe d'enregistrement actif `Room` qui contiendra la valeur de `volume` :

```php
class Room extends \yii\db\ActiveRecord
{
    public $volume;

    // ...
}
```

Ensuite, vous devez composer une requête qui calcule le volume de la chambre et effectue le tri :

```php
$rooms = Room::find()
    ->select([
        '{{room}}.*', // selectionne toutes les colonnes
        '([[length]] * [[width]] * [[height]]) AS volume', // calcule un
        volume
    ])
    ->orderBy('volume DESC') // applique le tri
    ->all();

foreach ($rooms as $room) {
    echo $room->volume; // contient la valeur calculée par SQL
}
```

La possibilité de sélectionner des champs supplémentaires peut être exceptionnellement utile pour l'agrégation de requêtes. Supposons que vous ayez besoin d'afficher une liste des clients avec le nombre total de commandes qu'ils ont passées. Tout d'abord, vous devez déclarer une classe `Customer` avec une relation `orders` et un champ supplémentaire pour le stockage du nombre de commandes :

```php
class Customer extends \yii\db\ActiveRecord
{
    public $ordersCount;

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}
```

Ensuite vous pouvez composer une requête qui joint les commandes et calcule leur nombre :

```php
$customers = Customer::find()
    ->select([
        '{{customer}}.*', // selectionne tous les champs de customer
        'COUNT({{order}}.id) AS ordersCount' // calcule le nombre de
        commandes (orders)
    ])
    ->joinWith('orders') // garantit la jointure de la table
    ->groupBy('{{customer}}.id') // groupe les résultats pour garantir que
    la fonction d'agrégation fonctionne
    ->all();
```

Un inconvénient de l'utilisation de cette méthode est que si l'information n'est pas chargée dans la requête SQL, elle doit être calculée séparément. Par conséquent, si vous avez trouvé un enregistrement particulier via une requête régulière sans instruction de sélection supplémentaire, il ne pourra retourner la valeur réelle du champ supplémentaire. La même chose se produit pour l'enregistrement nouvellement sauvegardé.

```php
$room = new Room();
$room->length = 100;
$room->width = 50;
$room->height = 2;

$room->volume; // cette valeur est `null` puisqu'elle n'a pas encore été
déclarée
```

En utilisant les méthodes magiques `__get()` et `__set()` nous pouvons émuler le comportement d'une propriété :

```php
class Room extends \yii\db\ActiveRecord
{
    private $_volume;

    public function setVolume($volume)
    {
        $this->_volume = (float) $volume;
    }

    public function getVolume()
    {
        if (empty($this->length) || empty($this->width) ||
        empty($this->height)) {
            return null;
        }

        if ($this->_volume === null) {
            $this->setVolume(
                $this->length * $this->width * $this->height
            );
        }

        return $this->_volume;
    }

    // ...
}
```

Lorsque la requête *select* ne fournit pas le volume, le modèle est capable de le calculer automatiquement en utilisant les attributs du modèle.

Vous pouvez aussi bien calculer les champs agrégés en utilisant les relations définies :

```php
class Customer extends \yii\db\ActiveRecord
{
    private $_ordersCount;

    public function setOrdersCount($count)
    {
        $this->_ordersCount = (int) $count;
    }
```

```
    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // cela évite d'appeler une requête pour chercher
            une clé primaire nulle
        }

        if ($this->_ordersCount === null) {
            $this->setOrdersCount($this->getOrders()->count()); // calcule
            l'aggrégation à la demande à partir de la relation
        }

        return $this->_ordersCount;
    }

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}
```

Avec ce code, dans le cas où 'ordersCount' est présent dans l'instruction 'select' - `Customer::ordersCount` est peuplé par les résultats de la requête, autrement il est calculé à la demande en utilisant la relation `Customer::orders`.

Cette approche peut aussi bien être utilisée pour la création de raccourcis pour certaines données relationnelles, en particulier pour l'aggrégation. Par exemple :

```
class Customer extends \yii\db\ActiveRecord
{
    /**
     * Definit une propriété en lecture seule pour les données agrégées.
     */
    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // ceci évite l'appel d'une requête pour cherche
            une clé primaire nulle
        }

        return empty($this->ordersAggregation) ? 0 :
        $this->ordersAggregation[0]['counted'];
    }

    /**
     * Déclere une relation 'orders' normale.
     */
    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
```

```
    /**
     * Déclare une nouvelle relation basée sur 'orders', qui fournit
l'agrégation.
     */
    public function getOrdersAggregation()
    {
        return $this->getOrders()
            ->select(['customer_id', 'counted' => 'count(*)'])
            ->groupBy('customer_id')
            ->asArray(true);
    }

    // ...
}

foreach (Customer::find()->with('ordersAggregation')->all() as $customer) {
    echo $customer->ordersCount; // fournit les données agrégées à partir de
    la relation sans requête supplémentaire grâce au chargement précoce.
}

$customer = Customer::findOne($pk);
$customer->ordersCount; // fournit les données agrégées à partir de la
relation paresseuse chargée
```

6.4 Migrations de base de données

Durant la période de développement et de maintenance d'une application s'appuyant sur une base de données, la structure de la base de données évolue tout comme le code source. Par exemple, durant développement une nouvelle table peut devenir nécessaire ; après que l'application est déployée en production, on peut s'apercevoir qu'un index doit être créé pour améliorer la performance des requêtes ; et ainsi de suite. Comme un changement dans la base de données nécessite souvent des changements dans le code, Yii prend en charge une fonctionnalité qu'on appelle *migrations de base de données*. Cette fonctionnalité permet de conserver la trace des changements de la base de données en termes de *migrations de base de données* dont les versions sont contrôlées avec celles du code.

Les étapes suivantes montrent comment des migrations de base de données peuvent être utilisées par une équipe durant la phase de développement :

1. Tim crée une nouvelle migration (p. ex. créer une nouvelle table, changer la définition d'une colonne, etc.).
2. Tim entérine (commit) la nouvelle migration dans le système de contrôle de version (p. ex. Git, Mercurial).
3. Doug met à jour son dépôt depuis le système de contrôle de version et reçoit la nouvelle migration.

4. Doug applique la migration à sa base de données de développement locale, et ce faisant synchronise sa base de données pour refléter les changements que Tim a faits.

Les étapes suivantes montrent comment déployer une nouvelle version avec les migrations de base de données en production :

1. Scott crée une balise de version pour le dépôt du projet qui contient quelques nouvelles migrations de base de données.
2. Scott met à jour le code source sur le serveur de production à la version balisée.
3. Scott applique toutes les migrations accumulées à la base de données de production.

Yii fournit un jeu de commandes de migration en ligne de commande qui vous permet de :

— créer de nouvelles migrations ;
— appliquer les migrations ;
— défaire les migrations ;
— ré-appliquer les migrations ;
— montrer l'historique de l'état des migrations.

Tous ces outils sont accessibles via la commande `yii migrate`. Dans cette section nous décrivons en détails comment accomplir des tâches variées en utilisant ces outils. Vous pouvez aussi obtenir les conseils d'utilisation de chacun des outils via la commande d'aide `yii help migrate`.

> Astuce : les migrations peuvent non seulement affecter le schéma de base de données mais aussi ajuster les données existantes pour s'adapter au nouveau schéma, créer la hiérarchie RBAC (Role Based Acces Control - Contrôle d'accès basé sur les rôles), ou vider le cache.

> Note : lors de la manipulation de données utilisant une migration, vous pouvez trouver qu'utiliser vos classes Active Record pour cela peut s'avérer utile parce qu'une partie de la logique y est déjà mise en œuvre. Soyez cependant conscient que, contrairement au code écrit dans les migrations, dont la nature est de rester constant à jamais, la logique d'application est sujette à modification. C'est pourquoi, lorsque vous utilisez des classes ActiveRecord dans le code d'une migration, des modifications de la logique de l'ActiveRecord peuvent accidentellement casser des migrations existantes. Pour cette raison, le code des migrations devrait être conservé indépendant d'autres logiques d'application telles que celles des classes ActiveRecord.

6.4.1 Création de migrations

Pour créer une nouvelle migration, exécutez la commande suivante :

```
yii migrate/create <name>
```

L'argument `name` requis donne une brève description de la nouvelle migration. Par exemple, si la création concerne la création d'une nouvelle table nommée *news*, vous pouvez utiliser le nom `create_news_table` et exécuter la commande suivante :

```
yii migrate/create create_news_table
```

> **Note :** comme l'argument `name` est utilisé comme partie du nom de la classe migration générée, il ne doit contenir que des lettres, des chiffre et/ou des caractères *souligné*.

La commande ci-dessus crée une nouvelle classe PHP nommée `m150101_185401_create_news_table.php` dans le dossier `@app/migrations`. Le fichier contient le code suivant qui déclare principalement une classe de migration `m150101_185401_create_news_table` avec le squelette de code suivant :

```php
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {

    }

    public function down()
    {
        echo "m101129_185401_create_news_table cannot be reverted.\n";

        return false;
    }

    /*
    // Use safeUp/safeDown to run migration code within a transaction
    public function safeUp()
    {
    }

    public function safeDown()
    {
    }
    */
}
```

Chaque migration de base de données est définie sous forme de classe PHP étendant la classe `yii\db\Migration`. Le nom de la classe de migration est généré automatiquement dans le format `m<YYMMDD_HHMMSS>_<Name>`, dans lequel :

— `<YYMMDD_HHMMSS>` fait référence à l'horodate UTC à laquelle la commande de création de la migration a été exécutée.
— `<Name>` est le même que la valeur que vous donnez à l'argument `name` dans la commande.

Dans la classe de migration, vous devez écrire du code dans la méthode `up()` qui effectue les modifications dans la structure de la base de données. Vous désirez peut-être écrire du code dans la méthode `down()` pour défaire les changements apportés par `up()`. La méthode `up()` est invoquée lorsque vous mettez à jour la base de données avec la migration, tandis que la méthode `down()` est invoquée lorsque vous ramenez la base de données à l'état antérieur. Le code qui suit montre comment mettre en œuvre la classe de migration pour créer une table `news` :

```
<?php

use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => Schema::TYPE_PK,
            'title' => Schema::TYPE_STRING . ' NOT NULL',
            'content' => Schema::TYPE_TEXT,
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

> **Info :** toutes les migrations ne sont pas réversibles. Par exemple, si la méthode `up()` supprime une ligne dans une table, il se peut que vous soyez incapable de récupérer cette ligne dans la méthode `down()`. Parfois, vous pouvez simplement être trop paresseux pour implémenter la méthode `down`, parce que défaire une migration de base de données n'est pas chose courante. Dans ce cas, vous devriez retourner `false` dans la méthode `down()` pour indiquer que la migration n'est pas réversible.

La classe de migration de base `yii\db\Migration` expose une connexion à une base de données via la propriété `db`. Vous pouvez utiliser cette connexion

pour manipuler le schéma en utilisant les méthodes décrites dans la sous-section Travail avec le schéma de base de données.

Plutôt que d'utiliser des types physiques, lors de la création d'une table ou d'une colonne, vous devez utiliser des *types abstraits* afin que vos migrations soient indépendantes d'un système de gestion de base de données en particulier. La classe `yii\db\Schema` définit une jeu de constantes pour représenter les types abstraits pris en charge. Ces constantes sont nommées dans le format `TYPE_<Name>`. Par exemple, `TYPE_PK` fait référence au type clé primaire à auto-incrémentation ; `TYPE_STRING` fait référence au type chaîne de caractères. Lorsqu'une migration est appliquée à une base de données particulière, le type abstrait est converti dans le type physique correspondant. Dans le cas de MySQL, `TYPE_PK` est transformé en `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY`, tandis que `TYPE_STRING` est transformé en `varchar(255)`.

Vous pouvez ajouter des contraintes additionnelles lors de l'utilisation des types abstraits. Dans l'exemple ci-dessus, `NOT NULL` est ajouté à `Schema::TYPE_STRING` pour spécifier que la colonne ne peut être `null` (nulle).

> **Info :** la mise en correspondance entre les types abstraits et les types physiques est spécifiée par la propriété `$typeMap` dans chacune des classes `QueryBuilder` concrètes.

Depuis la version 2.0.6, vous pouvez utiliser le constructeur de schéma récemment introduit qui procure un moyen plus pratique de définir le schéma d'une colonne. Ainsi, la migration ci-dessus pourrait s'écrire comme ceci :

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

Une liste de toutes les méthodes disponibles pour définir les types de colonne est disponible dans la documentation de l'API de `yii\db\SchemaBuilderTrait`.

6.4.2 Génération des migrations

Depuis la version 2.0.7, la commande de migration procure un moyen pratique de créer des migrations.

Si le nom de la migration est d'une forme spéciale, par exemple, `create_xxx_table` ou `drop_xxx_table` alors le fichier de la migration générée contient du code supplémentaire, dans ce cas pour créer/supprimer des tables. Dans ce qui suit, toutes les variantes de cette fonctionnalité sont décrites.

Création d'une table

```
yii migrate/create create_post_table
```

génère

```
/**
 * prend en charge la création de la table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey()
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

Pour créer les champs de table tout de suite, spécifiez les via l'option `--fields`.

```
yii migrate/create create_post_table --fields="title:string,body:text"
```

génère

```
/**
 * prend en charge la création de la table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
```

```
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

Vous pouvez spécifier plus de paramètres de champs.

```
yii migrate/create create_post_table
--fields="title:string(12):notNull:unique,body:text"
```

génère

```
/**
 * prend en charge la création de la table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

> **Note :** par défaut, une clé primaire nommée `id` est ajoutée automatiquement. Si vous voulez utiliser un autre nom, vous devez le spécifier explicitement comme dans `--fields="name:primaryKey"`.

Clés étrangères Depuis 2.0.8 le générateur prend en charge les clés étrangères en utilisant le mot clé `foreignKey`.

```
yii migrate/create create_post_table
--fields="author_id:integer:notNull:foreignKey(user),category_id:integer:defaultValue(1):foreignK
```

génère

```php
/**
 * prend en charge la création de la table `post`.
 * possède des clés étrangères vers les tables
 *
 * - `user`
 * - `category`
 */
class m160328_040430_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'author_id' => $this->integer()->notNull(),
            'category_id' => $this->integer()->defaultValue(1),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);

        // crée un index pour la colonne `author_id`
        $this->createIndex(
            'idx-post-author_id',
            'post',
            'author_id'
        );

        // ajoute une clé étrangère vers la table `user`
        $this->addForeignKey(
            'fk-post-author_id',
            'post',
            'author_id',
            'user',
            'id',
            'CASCADE'
        );

        // crée un index pour la colonne `category_id`
        $this->createIndex(
            'idx-post-category_id',
            'post',
            'category_id'
        );
```

```
        // ajoute une clé étrangère vers la table `category`
        $this->addForeignKey(
            'fk-post-category_id',
            'post',
            'category_id',
            'category',
            'id',
            'CASCADE'
        );
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        // supprime la clé étrangère vers la table `user`
        $this->dropForeignKey(
            'fk-post-author_id',
            'post'
        );

        // supprime l'index pour la colonne `author_id`
        $this->dropIndex(
            'idx-post-author_id',
            'post'
        );

        // supprime la clé étrangère vers la table `category`
        $this->dropForeignKey(
            'fk-post-category_id',
            'post'
        );

        // supprime l'index pour la colonne `category_id`
        $this->dropIndex(
            'idx-post-category_id',
            'post'
        );

        $this->dropTable('post');
    }
}
```

La position du mot clé `foreignKey` dans la description de la colonne ne change pas le code généré. Ce qui signifie que les expressions :

- `author_id:integer:notNull:foreignKey(user)`
- `author_id:integer:foreignKey(user):notNull`
- `author_id:foreignKey(user):integer:notNull`

génèrent toutes le même code.

Le mot clé `foreignKey` accepte un paramètre entre parenthèses qui est le nom de la table en relation pour la clé étrangère générée. Si aucun paramètre

n'est passé, le nom de table est déduit du nom de la colonne.

Dans l'exemple ci-dessus `author_id:integer:notNull:foreignKey(user)` génère une colonne nommée `author_id` avec une clé étrangère pointant sur la table `user`, tandis que `category_id:integer:defaultValue(1):foreignKey` génère une colonne `category_id` avec une clé étrangère pointant sur la table `category`.

Depuis la version 2.0.11, le mot clé `foreignKey` accepte un second paramètre, séparé par une espace. Il accepte le nom de la colonne en relation pour la clé étrangère générée. Si aucun second paramètre n'est passé, le nom de la colonne est retrouvé dans le schéma de table. Si aucun schéma n'existe, la clé primaire n'est pas définie ou est composite, le nom par défaut `id` est utilisé.

Suppression de tables

```
yii migrate/create drop_post_table
--fields="title:string(12):notNull:unique,body:text"
```

génère

```
class m150811_220037_drop_post_table extends Migration
{
    public function up()
    {
        $this->dropTable('post');
    }

    public function down()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }
}
```

Ajout de colonnes

Si le nom de la migration est de la forme `add_xxx_column_to_yyy_table` alors le fichier doit contenir les instructions `addColumn` et `dropColumn` nécessaires.

Pour ajouter une colonne :

```
yii migrate/create add_position_column_to_post_table
--fields="position:integer"
```

génère

```
class m150811_220037_add_position_column_to_post_table extends Migration
{
    public function up()
```

```
    {
        $this->addColumn('post', 'position', $this->integer());
    }

    public function down()
    {
        $this->dropColumn('post', 'position');
    }
}
```

Vous pouvez spécifier de multiples colonnes comme suit :

```
yii migrate/create add_xxx_column_yyy_column_to_zzz_table
--fields="xxx:integer,yyy:text"
```

Supprimer une colonne

Si le nom de la migration est de la forme `drop_xxx_column_from_yyy_table` alors le fichier doit contenir les instructions `addColumn` et `dropColumn` néessaires.

```
yii migrate/create drop_position_column_from_post_table
--fields="position:integer"
```

génère

```
class m150811_220037_drop_position_column_from_post_table extends Migration
{
    public function up()
    {
        $this->dropColumn('post', 'position');
    }

    public function down()
    {
        $this->addColumn('post', 'position', $this->integer());
    }
}
```

Ajout d'une table de jointure

Si le nom de la migration est de la forme `create_junction_table_for_xxx_and_yyy_tables` ou `create_junction_xxx_and_yyy_tables`, alors le code nécessaire à la création de la table de jointure est généré.

```
yii migrate/create create_junction_table_for_post_and_tag_tables
--fields="created_at:dateTime"
```

génère

```
/**
 * prend en charge la création de la table `post_tag`.
 * possède des clés étrangères vers les tables:
```

```
 *
 * - `post`
 * - `tag`
 */
class m160328_041642_create_junction_table_for_post_and_tag_tables extends
Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post_tag', [
            'post_id' => $this->integer(),
            'tag_id' => $this->integer(),
            'created_at' => $this->dateTime(),
            'PRIMARY KEY(post_id, tag_id)',
        ]);

        // crée un index pour la colonne`post_id`
        $this->createIndex(
            'idx-post_tag-post_id',
            'post_tag',
            'post_id'
        );

        // ajoute un clé étrangère vers la table `post`
        $this->addForeignKey(
            'fk-post_tag-post_id',
            'post_tag',
            'post_id',
            'post',
            'id',
            'CASCADE'
        );

        // crée un index pour la colonne `tag_id`
        $this->createIndex(
            'idx-post_tag-tag_id',
            'post_tag',
            'tag_id'
        );

        // ajoute une clé étrangère vers la table `tag`
        $this->addForeignKey(
            'fk-post_tag-tag_id',
            'post_tag',
            'tag_id',
            'tag',
            'id',
            'CASCADE'
        );
    }
```

```
    /**
     * {@inheritdoc}
     */
    public function down()
    {
        // supprime la clé étrangère vers la table `post`
        $this->dropForeignKey(
            'fk-post_tag-post_id',
            'post_tag'
        );

        // supprime l'index pour la colonne `post_id`
        $this->dropIndex(
            'idx-post_tag-post_id',
            'post_tag'
        );

        // supprime la clé étrangère vers la table `tag`
        $this->dropForeignKey(
            'fk-post_tag-tag_id',
            'post_tag'
        );

        // supprime l'index pour la column `tag_id`
        $this->dropIndex(
            'idx-post_tag-tag_id',
            'post_tag'
        );

        $this->dropTable('post_tag');
    }
}
```

Depuis la version 2.0.1, les noms de colonne des clés étrangères pour les tables de jonction sont recherchées dans le schéma de table. Dans le cas où la table n'est pas définie dans le schéma, ou quand la clé primaire n'est pas définie ou est composite, le nom par défaut `id` est utilisé.

Migrations transactionnelles

En effectuant des migration de base de données complexes, il est important de garantir que chacune des migrations soit réussisse, soit échoue dans son ensemble, de manière à ce que la base de données reste cohérente et intègre. Pour atteindre ce but, il est recommandé que vous englobiez les opérations de base de données de chacune des migrations dans une transaction.

Une manière encore plus aisée pour mettre en œuvre des migrations transactionnelles est de placer le code de migration dans les méthodes `safeUp()` et `safeDown()`. Ces deux méthodes diffèrent de `up()` et `down()` par le fait qu'elles sont implicitement englobées dans une transaction. En conséquence,

si n'importe quelle opération de ces méthodes échoue, toutes les opérations antérieures à elle sont automatiquement défaites.

Dans l'exemple suivant, en plus de créer la table `news`, nous insérons une ligne initiale dans cette table.

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function safeUp()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);

        $this->insert('news', [
            'title' => 'test 1',
            'content' => 'content 1',
        ]);
    }

    public function safeDown()
    {
        $this->delete('news', ['id' => 1]);
        $this->dropTable('news');
    }
}
```

Notez que, généralement, si vous effectuez de multiples opérations de base de données dans `safeUp()`, vous devriez les défaire dans `safeDown()`. Dans l'exemple ci-dessus, dans `safeUp()`, nous créons d'abord la table puis nous insérons une ligne, tandis que, dans `safeDown`, nous commençons par supprimer la ligne, puis nous supprimons la table.

> **Note :** tous les systèmes de gestion de bases de données NE prennent PAS en charge les transactions. De plus, quelques requêtes de base de données ne peuvent être placées dans une transaction. Pour quelques exemples, reportez-vous à entérinement implicite [29]. Si c'est le cas, vous devez simplement mettre en œuvre `up()` et`down()`, à la place.

Méthodes d'accès aux bases de données

La classe de base de migration `yii\db\Migration` fournit un jeu de méthodes pour vous permettre d'accéder aux bases de données et de les

29. https://dev.mysql.com/doc/refman/5.7/en/implicit-commit.html

manipuler. Vous vous apercevrez que ces méthodes sont nommées de façon similaires aux méthodes d'objets d'accès aux données fournies par la classe `yii\db\Command`. Par exemple, la méthode `yii\db\Migration::createTable()` vous permet de créer une nouvelle table, tout comme `yii\db\Command::createTable()`.

L'avantage d'utiliser les méthodes fournies par `yii\db\Migration` est que vous n'avez pas besoin de créer explicitement des instances de `yii\db\Command` et que l'exécution de chacune des méthodes affiche automatiquement des messages utiles vous indiquant que les opérations de base de données sont effectuées et combien de temps ces opérations ont pris.

Ci-dessous, nous présentons la liste de toutes les méthodes d'accès aux bases de données :

- `execute()` : exécute une instruction SQL
- `insert()` : insère une unique ligne
- `batchInsert()` : insère de multiples lignes
- `update()` : met à jour des lignes
- `delete()` : supprime des lignes
- `createTable()` : crée une table
- `renameTable()` : renomme une table
- `dropTable()` : supprime une table
- `truncateTable()` : supprime toutes les lignes d'une table
- `addColumn()` : ajoute une colonne
- `renameColumn()` : renomme une colonne
- `dropColumn()` : supprime une colonne
- `alterColumn()` : modifie une colonne
- `addPrimaryKey()` : ajoute une clé primaire
- `dropPrimaryKey()` : supprime une clé primaire
- `addForeignKey()` : ajoute une clé étrangère
- `dropForeignKey()` : supprime une clé étrangère
- `createIndex()` : crée un index
- `dropIndex()` : supprime un index
- `addCommentOnColumn()` : ajoute un commentaire à une colonne
- `dropCommentFromColumn()` : supprime un commentaire d'une colonne
- `addCommentOnTable()` : ajoute un commentaire à une table
- `dropCommentFromTable()` : supprime un commentaire d'une table

> **Info :** `yii\db\Migration` ne fournit pas une méthode de requête de base de données. C'est parce que, normalement, vous n'avez pas besoin d'afficher de messages supplémentaire à propos de l'extraction de données dans une base de données. C'est aussi parce que vous pouvez utiliser le puissant constructeur de requêtes pour construire et exécuter des requêtes complexes. L'utilisation du constructeur de requêtes dans une migration ressemble à ceci :

```
// update status field for all users
```

```php
foreach((new Query)->from('users')->each() as $user) {
    $this->update('users', ['status' => 1], ['id' => $user['id']]);
}
```

6.4.3 Application des migrations

Pour mettre une base de données à jour à sa dernière structure, vous devez appliquer toutes les nouvelles migrations disponibles en utilisant la commande suivante :

```
yii migrate
```

Cette commande liste toutes les migrations qui n'ont pas encore été appliquées. Si vous confirmez que vous voulez appliquer ces migrations, cela provoque l'exécution des méthodes `up()` ou `safeUp()` de chacune des nouvelles migrations, l'une après l'autre, dans l'ordre de leur horodate. Si l'une de ces migrations échoue, la commande se termine sans appliquer les migrations qui restent.

> Astuce : dans le cas où votre serveur ne vous offre pas de ligne de commande, vous pouvez essayer Web shell [30].

Pour chaque migration qui n'a pas été appliqué avec succès, la commande insère une ligne dans une table de base de données nommée `migration` pour enregistrer les applications réussies de la migration. Cela permet à l'outil de migration d'identifier les migrations qui ont été appliquées et celles qui ne l'ont pas été.

> **Info :** l'outil de migration crée automatiquement la table `migration` dans la base de données spécifiée par l'option `db` de la commande. Par défaut, la base de données est spécifiée dans le composant d'application `db`.

Parfois, vous désirez peut-être appliquer une ou quelques migrations plutôt que toutes les migrations disponibles. Vous pouvez le faire en spécifiant le nombre de migrations que vous voulez appliquer en exécutant la commande. Par exemple, la commande suivante essaye d'appliquer les trois prochaines migrations disponibles :

```
yii migrate 3
```

Vous pouvez également spécifier explicitement une migration particulière à laquelle la base de données doit être amenée en utilisant la commande `migrate/to` dans l'un des formats suivants :

30. https://github.com/samdark/yii2-webshell

```
yii migrate/to 150101_185401                      # utiliser l'horodatage
pour spécifier la migration
yii migrate/to "2015-01-01 18:54:01"              # utilise une chaîne de
caractères qui peut être analysée par strtotime()
yii migrate/to m150101_185401_create_news_table   # utilise le nom complet
yii migrate/to 1392853618                         # utilise un horodatage
UNIX
```

S'il existe des migrations non appliquée antérieures à celle spécifiée, elles sont toutes appliquées avant que la migration spécifiée ne le soit.

Si la migration spécifiée a déjà été appliquée auparavant, toutes les migrations postérieures qui ont été appliquées sont défaites.

6.4.4 Défaire des migrations

Pour défaire une ou plusieurs migrations que ont été appliquées auparavant, vous pouvez exécuter la commande suivante :

```
yii migrate/down     # défait la migration appliquée le plus récemment
yii migrate/down 3   # défait les 3 migrations appliquées le plus récemment
```

> **Note :** toutes les migrations ne sont PAS réversibles. Essayer de défaire de telles migrations provoque une erreur et arrête tout le processus de retour à l'état initial.

6.4.5 Refaire des migrations

Refaire (ré-appliquer) des migrations signifie d'abord défaire les migrations spécifiées puis les appliquer à nouveau. Cela peut être fait comme suit :

```
yii migrate/redo        # refait la dernière migration appliquée
yii migrate/redo 3      # refait les 3 dernière migrations appliquées
```

> **Note :** si une migration n'est pas réversible, vous ne serez pas en mesure de la refaire.

6.4.6 Rafraîchir des Migrations

Deepuis la version 2.0.13, vous pouvez supprimer toutes les tables et clés étrangères de la base de données et ré-appliquer toutes les migrations depuis le début. ` yii migrate/fresh # Tronçonne la base de données et applique toutes les migrations depuis le début

```
## Lister des migrations <span id="listing-migrations"></span>

Pour lister quelles migrations ont été appliquées et quelles migrations ne
l'ont pas été, vous pouvez utiliser les commandes suivantes :
```

yii migrate/history # montre les 10 dernières migrations appliquées yii migrate/history 5 # montre les 5 dernières migrations appliquées yii migrate/history all # montre toutes les migrations appliquées

yii migrate/new # montre les 10 premières nouvelles migrations yii migrate/new 5 # montre les 5 premières nouvelles migrations yii migrate/new all # montre toutes les nouvelles migrations `

6.4.7 Modification de l'historique des migrations

Au lieu d'appliquer ou défaire réellement des migrations, parfois, vous voulez peut-être simplement marquer que votre base de données a été portée à une certaine migration. Cela arrive souvent lorsque vous changer manuellement la base de données pour l'amener à un état particulier et que vous ne voulez pas que la migration correspondant à ce changement soit appliquée de nouveau par la suite. Vous pouvez faire cela avec la commande suivante : `yii migrate/mark 150101_185401 # utilise un horodatage pour spécifier la migration yii migrate/mark "2015-01-01 18:54:01" # utilise une chaîne de caractères qui peut être analysée par strtotime() yii migrate/mark m150101_185401_creat # utilise le nom complet yii migrate/mark 1392853618 # utilise un horodatage UNIX`

La commande modifie la table `migration` en ajoutant ou en supprimant certaines lignes pour indiquer que la base de données s'est vue appliquer toutes les migrations jusqu'à celle spécifiée. Aucune migration n'est appliquée ou défaite par cette commande.

6.4.8 Personnalisation des migrations

Il y a plusieurs manières de personnaliser la commande de migration.

Utilisation des options de ligne de commande

La commande de migration possède quelques options en ligne de commande qui peuvent être utilisées pour personnaliser son comportement :

- `interactive` : boolean (valeur par défaut `true`), spécifie si la migration doit être effectuées en mode interactif. Lorsque cette option est `true`, l'utilisateur reçoit un message avant que la commande n'effectue certaines actions. Vous désirez peut-être définir cette valeur à `false` si la commande s'exécute en arrière plan.
- `migrationPath` : string (valeur par défaut `@app/migrations`), spécifie le dossier qui stocke tous les fichiers de classe de migration. Cela peut être spécifié soit comme un chemin de dossier, soit comme un alias de chemin. Notez que le dossier doit exister sinon la commande déclenche une erreur.
- `migrationTable` : string (valeur par défaut `migration`), spécifie le nom de la table de base de données pour stocker l'historique de migration. La

table est créée automatiquement par la commande si elle n'existe pas encore. Vous pouvez aussi la créer à la main en utilisant la structure `version varchar(255) primary key, apply_time integer.`

— `db` : string (valeur par défaut `db`), spécifie l'identifiant du composant d'application base de données. Il représente la base de données à laquelle les migrations sont appliquées avec cette commande.

— `templateFile` : string (valeur par défaut `@yii/views/migration.php`), spécifie le chemin vers le fichier modèle qui est utilisé pour générer le squelette des fichiers de classe de migration. Cela peut être spécifié soit sous forme de chemin de fichier, soit sous forme d'alias de chemin. Le fichier modèle est un script PHP dans lequel vous pouvez utiliser une variable prédéfinie nommée `$className` pour obtenir le nom de la classe de migration.

— `generatorTemplateFiles` : array (valeur par défaut '[

```
'create_table' => '@yii/views/createTableMigration.php',
'drop_table' => '@yii/views/dropTableMigration.php',
'add_column' => '@yii/views/addColumnMigration.php',
'drop_column' => '@yii/views/dropColumnMigration.php',
'create_junction' => '@yii/views/createTableMigration.php'
```

]'), spécifie les fichiers modèles pour générer le code de migration. Voir "Génération des migrations" pour plus de détails.

— `fields` : array (tableau) de chaîne de caractères de définition de colonnes utilisées pour créer le code de migration. Valeur par défaut `[]`. Le format de chacune des définitions est `COLUMN_NAME:COLUMN_TYPE:COLUMN_DECORATOR`. Par exemple, `--fields=name:string(12):notNull` produit une colonne chaîne de caractères de taille 12 qui n'est pas `null` (nulle).

L'exemple suivant montre comment vous pouvez utiliser ces options.

Par exemple, si vous voulez appliquer des migrations à un module `forum` dont les fichiers de migration sont situés dans le dossier `migrations` du module, vous pouvez utiliser la commande suivante :

```
# Appliquer les migrations d'un module forum sans interactivité
yii migrate --migrationPath=@app/modules/forum/migrations --interactive=0
```

Configuration globale des commandes

Au lieu de répéter les mêmes valeurs d'option à chaque fois que vous exécutez une commande de migration, vous pouvez la configurer une fois pour toute dans la configuration de l'application comme c'est montré ci-après :

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationTable' => 'backend_migration',
```

```
        ],
    ],
];
```

Avec la configuration ci-dessus, à chaque fois que vous exécutez la commande de migration, la table `backend_migration` est utilisée pour enregistrer l'historique de migration. Vous n'avez plus besoin de le spécifier via l'option en ligne de commande `migrationTable`.

Migrations avec espaces de noms

Depuis la version 2.0.10, vous pouvez utiliser les espaces de noms pour les classes de migration. Vous pouvez spécifier la liste des espaces de noms des migrations via `migrationNamespaces`. L'utilisation des espaces de noms pour les classes de migration vous permet l'utilisation de plusieurs emplacement pour les sources des migrations. Par exemple :

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => null, // désactive les migration sans espace
            de noms si app\migrations est listé ci-dessous
            'migrationNamespaces' => [
                'app\migrations', // Migration ordinaires pour l'ensemble de
                l'application
                'module\migrations', // Migrations pour le module de projet
                spécifique
                'some\extension\migrations', // Migrations pour l'extension
                spécifique
            ],
        ],
    ],
];
```

> Note : les migrations appliquées appartenant à des espaces de noms différent créent un historique de migration **unique**, p. ex. vous pouvez être incapable d'appliquer ou d'inverser des migrations d'un espace de noms particulier seulement.

Lors des opérations sur les migrations avec espaces de noms : la création, l'inversion, etc. vous devez spécifier l'espace de nom complet avant le nom de la migration. Notez que le caractère barre oblique inversée (\) est en général considéré comme un caractère spécial dans l'interprète de commandes, c'est pourquoi vous devez l'échapper correctement pour éviter des erreurs d'interprète de commandes ou des comportements incorrects. Par exemple :

```
yii migrate/create 'app\\migrations\\createUserTable'
```

> Note : les migrations spécifiées via `migrationPath` ne peuvent pas contenir un espace de noms, les migrations avec espaces de noms peuvent être appliquée via la propriété `yii\console\controllers\MigrateController::$migrationNamespaces`.

Depuis la version 2.0.12, la propriété `migrationPath` accepte également un tableau pour spécifier de multiples dossiers contenant des migrations sans espaces de noms. Cela a été ajouté principalement pour être utilisé dans des projets existants qui utilisent des migrations provenant de différents emplacements. Ces migrations viennent principalement de sources externes, comme les extensions à Yii développées par d'autres développeurs, qui ne peuvent être facilement modifiées pour utiliser les espaces de noms lors du démarrage avec la nouvelle approche.

Migrations séparées

Parfois, l'utilisation d'un historique unique de migration pour toutes les migrations du projet n'est pas souhaité. Par exemple : vous pouvez installer une extension 'blog', qui contient des fonctionnalités complètement séparées et contient ses propres migrations, qui ne devraient pas affecter celles dédiées aux fonctionnalités principales du projet. Si vous voulez qui plusieurs migrations soient appliquées et complétement tracées séparément l'une de l'autre, vous pouvez configurer de multiples commandes de migration qui utilisent des espaces de noms différents et des tables d'historique de migration différentes :

```
return [
    'controllerMap' => [
        // Common migrations for the whole application
        'migrate-app' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['app\migrations'],
            'migrationTable' => 'migration_app',
            'migrationPath' => null,
        ],
        // Migrations for the specific project's module
        'migrate-module' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['module\migrations'],
            'migrationTable' => 'migration_module',
            'migrationPath' => null,
        ],
        // Migrations for the specific extension
        'migrate-rbac' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => '@yii/rbac/migrations',
            'migrationTable' => 'migration_rbac',
        ],
    ],
];
```

Notez que pour synchroniser la base de données vous devez maintenant exécuter plusieurs commandes au lieu d'une seule :

```
yii migrate-app
yii migrate-module
yii migrate-rbac
```

6.4.9 Migration de multiples base de données

Par défaut, les migrations sont appliquées à la même base de données spécifiée par le composant d'application `db`. Si vous voulez que celles-ci soient appliquées à des bases de données différentes, vous pouvez spécifier l'option en ligne de commande `db` comme indiqué ci-dessous :

```
yii migrate --db=db2
```

La commande ci-dessus applique les migration à la base de données `db2`.

Parfois, il est possible que vous vouliez appliquer *quelques unes* des migrations à une base de données, et *quelques autres* à une autre base de données. Pour y parvenir, lorsque vous implémentez une classe de migration, vous devez spécifier explicitement l'identifiant du composant base de données que la migration doit utiliser, comme ceci :

```php
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function init()
    {
        $this->db = 'db2';
        parent::init();
    }
}
```

La migration ci-dessus est appliquée à `db2`, même si vous spécifiez une autre base via l'option en ligne de commande `db`. Notez que l'historique de migration est toujours enregistré dans la base de données spécifiée par l'option en ligne de commande `db`.

Si vous avez de multiples migrations qui utilisent la même base de données, il est recommandé que vous créiez une classe de migration de base avec le code `init()` ci-dessus. Ensuite, chaque classe de migration peut étendre cette classe de base.

> Astuce : en plus de définir la propriété `db`, vous pouvez aussi opérer sur différentes bases de données en créant de nouvelles connexions à ces bases de données dans vos classes de migration. Ensuite,vous utilisez les méthodes des objets d'accès aux bases

de données avec ces connexions pour manipuler différentes bases de données.

Une autre stratégie que vous pouvez adopter pour appliquer des migrations à de multiples bases de données est de tenir ces migrations de différentes bases de données dans des chemins différents. Ensuite vous pouvez appliquer les migrations à ces bases de données dans des commandes séparées comme ceci :

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1
yii migrate --migrationPath=@app/migrations/db2 --db=db2
...
```

La première commande applique les migrations dans `@app/migrations/db1` à la base de données `db1`, la seconde commande applique les migrations dans `@app/migrations/db2` à `db2`, et ainsi de suite.

Error : not existing file : db-sphinx.md

Error : not existing file : db-redis.md

Error : not existing file : db-mongodb.md

Error : not existing file : db-elastic-search.md

Chapitre 7

Getting Data from Users

7.1 Création de formulaires

La manière primaire d'utiliser des formulaires dans Yii de faire appel aux `yii\widgets\ActiveForm`. Cette approche doit être privilégiée lorsque le formulaire est basé sur un modèle. En plus, il existe quelques méthodes utiles dans `yii\helpers\Html` qui sont typiquement utilisées pour ajouter des boutons et des textes d'aides de toute forme.

Un formulaire, qui est affiché du côté client, possède dans la plupart des cas, un modèle correspondant qui est utilisé pour valider ses entrées du côté serveur (lisez la section Validation des entrées pour plus de détails sur la validation). Lors de la création de formulaires basés sur un modèle, la première étape est de définir le modèle lui-même. Le modèle peut être soit basé sur une classe d'enregistrement actif représentant quelques données de la base de données, soit sur une classe de modèle générique qui étend la classe `yii\base\Model`) pour capturer des entrées arbitraires, par exemple un formulaire de connexion. Dans l'exemple suivant, nous montrons comment utiliser un modèle générique pour un formulaire de connexion :

```
<?php

class LoginForm extends \yii\base\Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // les règles de validation sont définies ici
        ];
    }
}
```

Dans le contrôleur, nous passons une instance de ce modèle à la vue, dans

laquelle le composant graphique `ActiveForm` est utilisé pour afficher le formulaire :

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'login-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>

    <div class="form-group">
        <div class="col-lg-offset-1 col-lg-11">
            <?= Html::submitButton('Login', ['class' => 'btn btn-primary'])
?>
        </div>
    </div>
<?php ActiveForm::end() ?>
```

Dans le code précédent, `ActiveForm::begin()` ne crée pas seulement une instance de formulaire, mais il marque également le début du formulaire. Tout le contenu placé entre `ActiveForm::begin()` et `ActiveForm::end()` sera enveloppé dans la balise HTML `<form>`. Comme avec tout composant graphique, vous pouvez spécifier quelques options sur la façon dont le composant graphique est configuré en passant un tableau à la méthode `begin`. Dans ce cas précis, une classe CSS supplémentaire et un identifiant sont passés pour être utilisés dans l'ouverture de balise `<form>`. Pour connaître toutes les options disponibles, reportez-vous à la documentation de l'API de `yii\widgets\ActiveForm`.

Afin de créer un élément *form* dans le formulaire, avec l'élément *label* et toute validation JavaScript applicable, la méthode `ActiveForm::field()` est appelée. Elle retourne une instance de `yii\widgets\ActiveField`. Lorsque le résultat de cette méthode est renvoyé en écho directement, le résultat est un champ de saisie de texte régulier. Pour personnaliser la sortie, vous pouvez enchaîner des méthodes additionnelles de `ActiveField` à cet appel :

```
// un champ de saisie du mot de passe
<?= $form->field($model, 'password')->passwordInput() ?>
// ajoute une invite et une étiquette personnalisée
<?= $form->field($model, 'username')->textInput()->hint('Please enter your
name')->label('Name') ?>
// crée un élément HTML5 de saisie d'une adresse de courriel
<?= $form->field($model, 'email')->input('email') ?>
```

Cela crée toutes les balises `<label>`, `<input>` et autres, selon le `modèle` défini par le champ de formulaire. Le nom du champ de saisie est déterminé automatiquement à partir du `nom de formulaire` du modèle et du nom d'attribut. Par exemple, le nom du champ de saisie de l'attribut `username` dans

l'exemple ci-dessus est `LoginForm[username]`. Cette règle de nommage aboutit à un tableau de tous les attributs du formulaire de connexion dans `$_POST['LoginForm']` côté serveur.

> **Conseil :** si vous avez seulement un modèle dans un formulaire et que vous voulez simplifier le nom des champs de saisie, vous pouvez sauter la partie tableau en redéfinissant la méthode `formName()` du modèle pour qu'elle retourne une chaîne vide. Cela peut s'avérer utile pour les modèles de filtres utilisés dans le composant graphique GridView pour créer des URL plus élégantes.

Spécifier l'attribut de modèle peut se faire de façon plus sophistiquée. Par exemple, lorsqu'un attribut peut prendre une valeur de tableau lors du chargement sur le serveur de multiples fichiers ou lors de la sélection de multiples items, vous pouvez le spécifier en ajoutant `[]` au nom d'attribut :

```
// permet à de multiples fichiers d'être chargés sur le serveur :
echo $form->field($model,
'uploadFile[]')->fileInput(['multiple'=>'multiple']);

// permet à de multiples items d'être cochés :
echo $form->field($model, 'items[]')->checkboxList(['a' => 'Item A', 'b' =>
'Item B', 'c' => 'Item C']);
```

Soyez prudent lorsque vous nommez des éléments de formulaire tels que des boutons de soumission. Selon la documentation de jQuery [1], certains noms sont réservés car ils peuvent créer des conflits :

> Les éléments *forms* et leurs éléments enfants ne devraient par utiliser des noms de champ de saisie, ou des identifiants que entrent en conflit avec les propriétés d'un élément de *form*, tels que `submit`, `length`, ou `method`. Les conflits de noms peuvent créer des échecs troublants. Pour une liste complètes des règles et pour vérifier votre code HTML à propos de ces problèmes, reportez-vous à DOMLint [2].

Des balises additionnelles HTML peuvent être ajoutées au formulaire en utilisant du HTML simple ou en utilisant les méthodes de la classe `Html`-helper comme cela est fait dans l'exemple ci-dessus avec le `bouton de soumission`.

> **Conseil :** si vous utilisez la base structurée *Twitter Bootstrap CSS* dans votre application, vous désirez peut-être utiliser `yii\bootstrap\ActiveForm` à la place de `yii\widgets\ActiveForm`. La première étend la deuxième et utilise les styles propres à Bootstrap lors de la génération des champs de saisie du formulaire.

1. https://api.jquery.com/submit/
2. https://kangax.github.io/domlint/

Conseil : afin de styler les champs requis avec une astérisque, vous pouvez utiliser le CSS suivant :

```
div.required label.control-label:after {
    content: " *";
    color: red;
}
```

7.1.1 Création d'une liste déroulante

Vous pouvez utiliser la méthode dropDownList() [3] de ActiveForm pour créer une liste déroulante :

```
use app\models\ProductCategory;

/* @var $this yii\web\View */
/* @var $form yii\widgets\ActiveForm */
/* @var $model app\models\Product */

echo $form->field($model, 'product_category')->dropdownList(
    ProductCategory::find()->select(['category_name',
    'id'])->indexBy('id')->column(),
    ['prompt'=>'Select Category']
);
```

La valeur du champ de saisie de votre modèle est automatiquement pré-selectionnée

7.1.2 Travail avec Pjax

Le composant graphique `Pjax` vous permet de mettre à jour une certaine section d'une page plutôt que de recharger la page entière. Vous pouvez l'utiliser pour mettre à jour seulement le formulaire et remplacer son contenu après la soumission.

Vous pouvez configurer `$formSelector` pour spécifier quelles soumissions de formulaire peuvent déclencher pjax. Si cette propriété n'est pas définie, tous les formulaires avec l'attribut `data-pjax` dans le contenu englobé par Pjax déclenchent des requêtes pjax.

```
use yii\widgets\Pjax;
use yii\widgets\ActiveForm;

Pjax::begin([
    // Pjax options
]);
    $form = ActiveForm::begin([
        'options' => ['data' => ['pjax' => true]],
```

3. https://www.yiiframework.com/doc-2.0/yii-widgets-activefield.html#dropDownList()-detail

```
        // plus d'options d'ActiveForm
    ]);

        // contenu de ActiveForm

    ActiveForm::end();
Pjax::end();
```

> **Conseil :** soyez prudent avec les liens à l'intérieur du composant graphique `Pjax` car la réponse est également rendue dans le composant graphique. Pour éviter cela, utilisez l'attribut HTML `data-pjax="0"`.

Valeurs dans les boutons de soumission et dans les chargement de fichiers sur le serveur Il y a des problèmes connus avec l'utilisation de `jQuery.serializeArray()` lorsqu'on manipule des fichiers [4] et des valeurs de boutons de soumission [5] qui ne peuvent être résolus et sont plutôt rendus obsolète en faveur de la classe `FormData` introduite en HTML5.

Cela siginifie que la seule prise en charge officielle pour les fichiers et les valeurs de boutons de soumission avec ajax, ou en utilisant le composant graphique `Pjax`, dépend de la prise en charge par le navigateur [6] de la classe `FormData`.

7.1.3 Lectures d'approfondissement

La section suivante, Validation des entrées prend en charge la validation des données soumises par le formulaire du côté serveur ainsi que la validation ajax et du côté client.

Pour en apprendre plus sur les utilisations complexes de formulaires, vous pouvez lire les sections suivantes :

- Collecte des champs de saisie tabulaires, pour collecter des données à destination de multiples modèles du même genre.
- Obtention de données pour de multiples modèles, pour manipuler plusieurs modèles différents dans le même formulaire.
- Chargement de fichiers sur le serveur, sur la manière d'utiliser les formulaires pour charger des fichiers sur le serveur.

7.2 Validation des entrées utilisateur

En général, vous ne devriez jamais faire confiance aux données entrées par l'utilisateur et devriez toujours les valider avant de les utiliser.,

4. https://github.com/jquery/jquery/issues/2321
5. https://github.com/jquery/jquery/issues/2321
6. https://developer.mozilla.org/fr/docs/Web/API/FormData#compatibilit%C3%A9_des_navigateurs

Étant donné un modèle rempli par les données entrées par l'utilisateur, il est possible de valider ces entrées en appelant la méthode `yii\base\Model::validate()`. La méthode retourne une valeur booléenne qui indique si la validation a réussi ou pas. Si ce n'est pas le cas, vous pouvez obtenir les messages d'erreur depuis la propriété `yii\base\Model::$errors`. Par exemple :

```
$model = new \app\models\ContactForm();

// remplit les attributs du modèle avec les entrées de l'utilisateur
$model->load(\Yii::$app->request->post());
// ce qui est équivalent à :
// $model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // toutes les entrées sont valides
} else {
    // la validation a échoué: $errors est un tableau contenant les messages
    d'erreur
    $errors = $model->errors;
}
```

7.2.1 Déclaration de règles

Pour que `validate()` fonctionne réellement, vous devez déclarer des règles de validation pour les attributs que vous envisagez de valider. Cela peut être réalisé en redéfinissant la méthode `yii\base\Model::rules()`. L'exemple suivant montre comment les règles de validation pour le modèle `ContactForm` sont déclarées :

```
public function rules()
{
    return [
        // les attributs  name, email, subject et  body sont à saisir
        obligatoirement
        [['name', 'email', 'subject', 'body'], 'required'],

        // l'attribut email doit être une adresse de courriel valide
        ['email', 'email'],
    ];
}
```

La méthode `rules()` doit retourner un tableau de règles, dont chacune est un tableau dans le format suivant :

```
[
    // obligatoire, spécifie quels attributs doivent être validés par cette
    règle.
    // Pour un attribut unique, vous pouvez utiliser le nom de l'attribut
    directement
    // sans le mettre dans un tableau
```

```
    ['attribute1', 'attribute2', ...],

    // obligatoire, spécifier le type de cette règle.
    // Il peut s'agir d'un nom de classe, d'un alias de validateur ou du nom
    d'une méthode de validation
    'validator',

    // facultatif, spécifie dans quel(s) scénario(s) cette règle doit être
    appliquée
    // si absent, cela signifie que la règle s'applique à tous les
    scénarios
    // Vous pouvez aussi configurer l'option "except" si vous voulez que la
    règle
    // s'applique à tous les scénarios sauf à ceux qui sont listés
    'on' => ['scenario1', 'scenario2', ...],

    // facultatif, spécifie des configurations additionnelles pour l'objet
    validateur
    'property1' => 'value1', 'property2' => 'value2', ...
]
```

Pour chacune des règles vous devez spécifier au moins à quels attributs la règle s'applique et quel est le type de cette règle. Vous pouvez spécifier le type de la règle sous l'une des formes suivantes :

— l'alias d'un validateur du noyau, comme `required`, `in`, `date`, etc. Reportez-vous à la sous-section Validateurs du noyau pour une liste complète des validateurs du noyau.
— le nom d'une méthode de validation dans la classe du modèle, ou une fonction anonyme. Reportez-vous à la sous-section Inline Validators pour plus de détails.
— un nom de classe de validateur pleinement qualifié. Reportez-vous à la sous-section Validateurs autonomes pour plus de détails.

Une règle peut être utilisée pour valider un ou plusieurs attributs, et un attribut peut être validé par une ou plusieurs règles. Une règle peut s'appliquer dans certains scenarios seulement en spécifiant l'option `on`. Si vous ne spécifiez pas l'option `on`, la règle s'applique à tous les scénarios.

Quand la méthode `validate()` est appelée, elle suit les étapes suivantes pour effectuer l'examen de validation :

1. Détermine quels attributs doivent être validés en obtenant la liste des attributs de `yii\base\Model::scenarios()` en utilisant le `scenario` courant. Ces attributs sont appelés *attributs actifs*.
2. Détermine quelles règles de validation doivent être appliquées en obtenant la liste des règles de `yii\base\Model::rules()` en utilisant le `scenario` courant. Ces règles sont appelées *règles actives*.
3. Utilise chacune des règles actives pour valider chacun des attributs qui sont associés à cette règle. Les règles sont évaluées dans l'ordre dans lequel elles sont listées.

Selon les étapes de validation décrites ci-dessus, un attribut est validé si, et seulement si, il est un attribut actif déclaré dans `scenarios()` et est associé à une ou plusieurs règles actives déclarées dans `rules()`.

> **Note :** il est pratique le nommer les règles, c.-à-d.

```
public function rules()
{
    return [
        // ...
        'password' => [['password'], 'string', 'max' => 60],
    ];
}
```

Vous pouvez l'utiliser dans un modèle enfant :

```
public function rules()
{
    $rules = parent::rules();
    unset($rules['password']);
    return $rules;
}
```

Personnalisation des messages d'erreur

La plupart des validateurs possèdent des messages d'erreurs qui sont ajoutés au modèle en cours de validation lorsque ses attributs ne passent pas la validation. Par exemple, le validateur `required` ajoute le message "Username cannot be blank." (Le nom d'utilisateur ne peut être vide.) au modèle lorsque l'attribut `username` ne passe pas la règle de validation utilisant ce validateur.

Vous pouvez personnaliser le message d'erreur d'une règle en spécifiant la propriété `message` lors de la déclaration de la règle, comme ceci :

```
public function rules()
{
    return [
        ['username', 'required', 'message' => 'Please choose a username.'],
    ];
}
```

Quelques validateurs peuvent prendre en charge des messages d'erreur additionnels pour décrire précisément les différentes causes de non validation. Par exemple, le validateur `number` prend en charge`tooBig (trop grand)` et `tooSmall (trop petit)` pour décrire la cause de non validation lorsque la valeur à valider est trop grande ou trop petite, respectivement. Vous pouvez configurer ces messages d'erreur comme vous configureriez d'autres propriétés de validateurs dans une règle de validation.

Événement de validation

Losque la méthode `yii\base\Model::validate()` est appelée, elle appelle deux méthodes que vous pouvez redéfinir pour personnaliser le processus de validation :

— `yii\base\Model::beforeValidate()` : la mise en œuvre par défaut déclenche un événement `yii\base\Model::EVENT_BEFORE_VALIDATE`. Vous pouvez, soit redéfinir cette méthode, soit répondre à cet événement pour accomplir un travail de pré-traitement (p. ex. normaliser les données entrées) avant que l'examen de validation n'ait lieu. La méthode retourne une valeur booléenne indiquant si l'examen de validation doit avoir lieu ou pas.

— `yii\base\Model::afterValidate()` : la mise en œuvre par défaut déclenche un événement `yii\base\Model::EVENT_AFTER_VALIDATE`. Vous pouvez, soit redéfinir cette méthode, soit répondre à cet événement pour accomplir un travail de post-traitement après que l'examen de validation a eu lieu.

Validation conditionnelle

Pour valider des attributs seulement lorsque certaines conditions sont réalisées, p. ex. la validation d'un attribut dépend de la valeur d'un autre attribut, vous pouvez utiliser la propriété `when` pour définir de telles conditions. Par exemple :

```
['state', 'required', 'when' => function($model) {
    return $model->country == 'USA';
}]
```

La propriété `when` accepte une fonction de rappel PHP avec la signature suivante :

```
/**
 * @param Model $model le modèle en cours de validation
 * @param string $attribute l'attribut en cours de validation
 * @return bool `true` si la règle doit être appliqué, `false` si non
 */
function ($model, $attribute)
```

Si vous avez aussi besoin de la prise en charge côté client de la validation conditionnelle, vous devez configurer la propriété `whenClient` qui accepte une chaîne de caractères représentant une fonction JavaScript dont la valeur de retour détermine si la règles doit être appliquée ou pas. Par exemple :

```
['state', 'required', 'when' => function ($model) {
    return $model->country == 'USA';
}, 'whenClient' => "function (attribute, value) {
    return $('#country').val() == 'USA';
}"]
```

Filtrage des données

Les entrées utilisateur nécessitent souvent d'être filtrées ou pré-traitées. Par exemple, vous désirez peut-être vous débarrasser des espaces devant et derrière l'entrée `username`. Vous pouvez utiliser les règles de validation pour le faire.

Les exemples suivants montrent comment se débarrasser des espaces dans les entrées et transformer des entrées vides en `null`s en utilisant les validateurs du noyau trim et default :

```
return [
    [['username', 'email'], 'trim'],
    [['username', 'email'], 'default'],
];
```

Vous pouvez également utiliser le validateur plus général filter pour accomplir un filtrage plus complexe des données.

Comme vous le voyez, ces règles de validation ne pratiquent pas un examen de validation proprement dit. Plus exactement, elles traitent les valeurs et les sauvegardent dans les attributs en cours de validation.

Gestion des entrées vides

Lorsque les entrées sont soumises par des formulaires HTML, vous devez souvent assigner des valeurs par défaut aux entrées si elles restent vides. Vous pouvez le faire en utilisant le validateur default. Par exemple :

```
return [
    // définit "username" et "email" comme *null* si elles sont vides
    [['username', 'email'], 'default'],

    // définit "level" à 1 si elle est vide
    ['level', 'default', 'value' => 1],
];
```

Par défaut, une entrée est considérée vide si sa valeur est une chaîne de caractères vide, un tableau vide ou un `null`. Vous pouvez personnaliser la logique de détection de vide en configurant la propriété `yii\validators\Validator::isEmpty()` avec une fonction de rappel PHP. Par exemple :

```
['agree', 'required', 'isEmpty' => function ($value) {
    return empty($value);
}]
```

> **Note :** la plupart des validateurs ne traitent pas les entrées vides si leur propriété `yii\validators\Validator::$skipOnEmpty` prend la valeur par défaut `true` (vrai). Ils sont simplement sautés lors de l'examen de validation si leurs attributs associés reçoivent des

entrées vides. Parmi les validateurs de noyau, seuls les validateurs `captcha`, `default`, `filter`, `required`, et `trim` traitent les entrées vides.

7.2.2 Validation ad hoc

Parfois vous avez besoin de faire une *validation ad hoc* pour des valeurs qui ne sont pas liées à un modèle.

Si vous n'avez besoin d'effectuer qu'un seul type de validation (p. ex. valider une adresse de courriel), vous pouvez appeler la méthode `validate()` du validateur désiré, comme ceci :

```
$email = 'test@example.com';
$validator = new yii\validators\EmailValidator();

if ($validator->validate($email, $error)) {
    echo 'Email is valid.';
} else {
    echo $error;
}
```

> **Note :** tous les validateurs ne prennent pas en charge ce type de validation. Le validateur du noyau unique, qui est conçu pour travailler avec un modèle uniquement, en est un exemple.

Si vous avez besoin de validations multiples pour plusieurs valeurs, vous pouvez utiliser `yii\base\DynamicModel` qui prend en charge, à la fois les attributs et les règles à la volée. Son utilisation ressemble à ce qui suit :

```
public function actionSearch($name, $email)
{
    $model = DynamicModel::validateData(compact('name', 'email'), [
        [['name', 'email'], 'string', 'max' => 128],
        ['email', 'email'],
    ]);

    if ($model->hasErrors()) {
        // validation fails
    } else {
        // validation succeeds
    }
}
```

La méthode `yii\base\DynamicModel::validateData()` crée une instance de `DynamicModel`, définit les attributs utilisant les données fournies (`name` et `email` dans cet exemple), puis appelle `yii\base\Model::validate()` avec les règles données.

En alternative, vous pouvez utiliser la syntaxe plus *classique* suivante pour effectuer la validation ad hoc :

```php
public function actionSearch($name, $email)
{
    $model = new DynamicModel(compact('name', 'email'));
    $model->addRule(['name', 'email'], 'string', ['max' => 128])
        ->addRule('email', 'email')
        ->validate();

    if ($model->hasErrors()) {
        // la validation a échoué
    } else {
        // la validation a réussi
    }
}
```

Après l'examen de validation, vous pouvez vérifier si la validation a réussi ou pas en appelant la méthode `hasErrors()` et obtenir les erreurs de validation de la propriété `errors`, comme vous le feriez avec un modèle normal. Vous pouvez aussi accéder aux attributs dynamiques définis via l'instance de modèle, p. ex. `$model->name` et `$model->email`.

7.2.3 Création de validateurs

En plus de pouvoir utiliser les validateurs du noyau inclus dans les versions publiées de Yii, vous pouvez également créer vos propres validateurs. Vous pouvez créer des validateurs en ligne et des validateurs autonomes.

Validateurs en ligne

Un validateur en ligne est un validateur défini sous forme de méthode de modèle ou de fonction anonyme. La signature de la méthode/fonction est :

```php
/**
 * @param string $attribute l'attribut en cours de validation
 * @param mixed $params la valeur des *paramètres* donnés dans la règle
 */
function ($attribute, $params)
```

Si un attribut ne réussit pas l'examen de validation, la méthode/fonction doit appeler `yii\base\Model::addError()` pour sauvegarder le message d'erreur dans le modèle de manière à ce qu'il puisse être retrouvé plus tard pour être présenté à l'utilisateur.

Voici quelques exemples :

```php
use yii\base\Model;

class MyForm extends Model
{
    public $country;
    public $token;
```

```
    public function rules()
    {
        return [
            // un validateur en ligne défini sous forme de méthode de modèle
            validateCountry()
            ['country', 'validateCountry'],

            // un validateur en ligne défini sous forme de fonction anonyme
            ['token', function ($attribute, $params) {
                if (!ctype_alnum($this->$attribute)) {
                    $this->addError($attribute, 'The token must contain
                    letters or digits.');
                }
            }],
        ];
    }

    public function validateCountry($attribute, $params)
    {
        if (!in_array($this->$attribute, ['USA', 'Web'])) {
            $this->addError($attribute, 'The country must be either "USA" or
            "Web".');
        }
    }
}
```

> **Note :** Par défaut, les validateurs en ligne ne sont pas appliqués si leurs attributs associés reçoivent des entrées vides ou s'ils ont déjà échoué à des examen de validation selon certaines règles. Si vous voulez être sûr qu'une règle sera toujours appliquée, vous devez configurer les propriétés `skipOnEmpty` et/ou `skipOnError` à `false` (faux) dans les déclarations des règles. Par exemple :
>
> ```
> [
>     ['country', 'validateCountry', 'skipOnEmpty' => false,
>     'skipOnError' => false],
> ]
> ```

Validateurs autonomes

Un validateur autonome est une classe qui étend la classe `yii\validators\Validator` ou une de ses classe filles. Vous pouvez mettre en œuvre sa logique de validation en redéfinissant la méthode `yii\validators\Validator::validateAttribute()`. Si un attribut ne réussit pas l'exament de validation, appellez `yii\base\Model::addError()` pour sauvegarder le message d'erreur dans le modèle, comme vous le feriez avec des validateurs en ligne.

Par exemple, le validateur en ligne ci-dessus peut être transformé en une nouvelle classe [[components/validators/CountryValidator]].

```
namespace app\components;
```

```php
use yii\validators\Validator;

class CountryValidator extends Validator
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['USA', 'Web'])) {
            $this->addError($model, $attribute, 'The country must be either
            "USA" or "Web".');
        }
    }
}
```

Si vous voulez que votre validateur prennent en charge la validation d'une valeur sans modèle, vous devez redéfinir la méthode `yii\validators\Validator::validate()`. Vous pouvez aussi redéfinir `yii\validators\Validator::validateValue()` au lieu de `validateAttribute()` et `validate()`, parce que, par défaut, les deux dernières méthodes sont appelées en appelant `validateValue()`.

Ci-dessous, nous présentons un exemple de comment utiliser la classe de validateur précédente dans votre modèle.

```php
namespace app\models;

use Yii;
use yii\base\Model;
use app\components\validators\CountryValidator;

class EntryForm extends Model
{
    public $name;
    public $email;
    public $country;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['country', CountryValidator::class],
            ['email', 'email'],
        ];
    }
}
```

7.2.4 Validation côté client

La validation côté client basée sur JavaScript est souhaitable lorsque l'utilisateur fournit les entrées via des formulaires HTML, parce que cela permet à l'utilisateur de détecter plus vite les erreurs et lui apporte ainsi un meilleur ressenti. Vous pouvez utiliser ou implémenter un validateur qui prend en charge la validation côté client *en plus* de la validation côté serveur.

> **Info :** bien que la validation côté client soit souhaitable, ce n'est pas une obligation. Son but principal est d'apporter un meilleur ressenti à l'utilisateur. Comme pour les données venant de l'utilisateur, vous ne devriez jamais faire confiance à la validation côté client. Pour cette raison, vous devez toujours effectuer la validation côté serveur en appelant `yii\base\Model::validate()`, comme nous l'avons décrit dans les sous-sections précédentes.

Utilisation de la validation côté client

Beaucoup de validateurs du noyau prennent en charge la validation côté client directement. Tout ce que vous avez à faire c'est utiliser `yii\widgets\ActiveForm` pour construire vos formulaires HTML. Par exemple, `LoginForm` ci-dessous déclare deux règles : l'une utilise le validateur du noyau required qui est pris en charge à la fois côté serveur et côté client ; l'autre utilise le validateur en ligne `validatePassword` qui ne prend pas en charge la validation côté client.

```
namespace app\models;

use yii\base\Model;
use app\models\User;

class LoginForm extends Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // username et password sont tous deux obligatoires
            [['username', 'password'], 'required'],

            // password est validé par validatePassword()
            ['password', 'validatePassword'],
        ];
    }

    public function validatePassword()
    {
        $user = User::findByUsername($this->username);

        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError('password', 'Incorrect username or password.');
        }
    }
}
```

Le formulaire HTML construit par le code suivant contient deux champs de saisie `username` et `password`. Si vous soumettez le formulaire sans rien saisir,

vous recevrez directement les messages d'erreur vous demandant d'entrer quelque chose sans qu'aucune communication avec le serveur n'ait lieu.

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('Login') ?>
<?php yii\widgets\ActiveForm::end(); ?>
```

En arrière plan, `yii\widgets\ActiveForm` lit les règles de validation déclarées dans le modèle et génère le code JavaScript approprié pour la prise en charge de la validation côté client. Lorsqu'un utilisateur modifie la valeur d'un champ de saisie ou soumet le formulaire, le code JavaScript est appelé.

Si vous désirez inhiber la validation côté client complètement, vous pouvez configurer la propriété `yii\widgets\ActiveForm::$enableClientValidation` à `false` (faux). Vous pouvez aussi inhiber la validation côté client pour des champs de saisie individuels en configurant leur propriété `yii\widgets\ActiveField::$enableClientValidation` à `false`. Lorsque `enableClientValidation` est configurée à la fois au niveau du champ et au niveau du formulaire, c'est la première configuration qui prévaut.

Mise en œuvre de la validation côté client

Pour créer un validateur qui prend en charge la validation côté client, vous devez implémenter la méthode `yii\validators\Validator::clientValidateAttribute()` qui retourne un morceau de code JavaScript propre à effectuer l'examen de validation côté client. Dans ce code JavaScript, vous pouvez utiliser les variables prédéfinies suivantes :

- `attribute` : le nom de l'attribut en cours de validation ;
- `value` : la valeur en cours de validation ;
- `messages` : un tableau utilisé pour contenir les messages d'erreurs pour l'attribut ;
- `deferred` : un tableau dans lequel les objets différés peuvent être poussés (explication dans la prochaine sous-section).

Dans l'exemple suivant, nous créons un `StatusValidator` qui valide une entrée si elle représente l'identifiant d'une donnée existante ayant un état valide. Le validateur prend en charge à la fois la validation côté serveur et la validation côté client.

```
namespace app\components;

use yii\validators\Validator;
use app\models\Status;

class StatusValidator extends Validator
{
    public function init()
```

```
    {
        parent::init();
        $this->message = 'Invalid status input.';
    }

    public function validateAttribute($model, $attribute)
    {
        $value = $model->$attribute;
        if (!Status::find()->where(['id' => $value])->exists()) {
            $model->addError($attribute, $this->message);
        }
    }

    public function clientValidateAttribute($model, $attribute, $view)
    {
        $statuses =
        json_encode(Status::find()->select('id')->asArray()->column());
        $message = json_encode($this->message, JSON_UNESCAPED_SLASHES |
        JSON_UNESCAPED_UNICODE);
        return <<<JS
if ($.inArray(value, $statuses) === -1) {
    messages.push($message);
}
JS;
    }
}
```

> **Conseil :** le code ci-dessus est donné essentiellement pour démontrer comment prendre en charge la validation côté client. En pratique, vous pouvez utiliser le validateur du noyau in pour arriver au même résultat. Vous pouvez écrire la règle de validation comme suit :
>
> ```
> [
>     ['status', 'in', 'range' =>
>     Status::find()->select('id')->asArray()->column()],
> ]
> ```
>
> **Conseil :** si vous avez besoin de travailler à la main avec la validation côté client, c.-à-d. ajouter des champs dynamiquement ou effectuer quelque logique d'interface utilisateur, reportez-vous à Travail avec ActiveForm via JavaScript [7] dans le *Cookbook* de Yii 2.0 .

Validation différée

Si vous devez effectuer une validation asynchrone côté client, vous pouvez créer des objets différés [8]. Par exemple, pour effectuer une validation AJAX

7. https://github.com/samdark/yii2-cookbook/blob/master/book/forms-activeform-js.md
8. https://api.jquery.com/category/deferred-object/

personnalisée, vous pouvez utiliser le code suivant :

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.push($.get("/check", {value: value}).done(function(data) {
            if ('' !== data) {
                messages.push(data);
            }
        }));
JS;
}
```

Dans ce qui précède, la variable `deferred` est fournie par Yii, et représente un tableau de d'objets différés. La méthode `$.get()` crée un objet différé qui est poussé dans le tableau `deferred`.

Vous pouvez aussi créer explicitement un objet différé et appeler sa méthode `resolve()` lorsque la fonction de rappel asynchrone est activée . L'exemple suivant montre comment valider les dimensions d'une image à charger sur le serveur du côté client.

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        var def = $.Deferred();
        var img = new Image();
        img.onload = function() {
            if (this.width > 150) {
                messages.push('Image too wide!!');
            }
            def.resolve();
        }
        var reader = new FileReader();
        reader.onloadend = function() {
            img.src = reader.result;
        }
        reader.readAsDataURL(file);

        deferred.push(def);
JS;
}
```

> **Note :** La méthode `resolve()` doit être appelée après que l'attribut a été validé. Autrement la validation principale du formulaire ne se terminera pas.

Pour faire simple, le tableau `deferred` est doté d'une méthode raccourci `add()` qui crée automatiquement un objet différé et l'ajoute au tableau `deferred`. En utilisant cette méthode, vous pouvez simplifier l'exemple ci-dessus comme suit :

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.add(function(def) {
            var img = new Image();
            img.onload = function() {
                if (this.width > 150) {
                    messages.push('Image too wide!!');
                }
                def.resolve();
            }
            var reader = new FileReader();
            reader.onloadend = function() {
                img.src = reader.result;
            }
            reader.readAsDataURL(file);
        });
JS;
}
```

Validation AJAX

Quelques validations ne peuvent avoir lieu que côté serveur, parce que seul le serveur dispose des informations nécessaires. Par exemple, pour valider l'unicité d'un nom d'utilisateur, il est nécessaire de consulter la table des utilisateurs côté serveur. Vous pouvez utiliser la validation basée sur AJAX dans ce cas. Elle provoquera une requête AJAX en arrière plan pour exécuter l'examen de validation tout en laissant à l'utilisateur le même ressenti que lors d'une validation côté client normale.

Pour activer la validation AJAX pour un unique champ de saisie, configurez la propriété `enableAjaxValidation` de ce champ à `true` et spécifiez un `identifiant` unique de formulaire :

```
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'registration-form',
]);

echo $form->field($model, 'username', ['enableAjaxValidation' => true]);

// ...

ActiveForm::end();
```

Pour étendre la validation AJAX à tout le formulaire, configurez la propriété `enableAjaxValidation` à `true` au niveau du formulaire :

```
$form = ActiveForm::begin([
    'id' => 'contact-form',
    'enableAjaxValidation' => true,
]);
```

> **Note :** lorsque la propriété `enableAjaxValidation` est configurée à la fois au niveau du champ et au niveau du formulaire, la première configuration prévaut.

Vous devez aussi préparer le serveur de façon à ce qu'il puisse prendre en charge les requêtes de validation AJAX . Cela peut se faire à l'aide d'un fragment de code comme celui qui suit dans les actions de contrôleur :

```
if (Yii::$app->request->isAjax && $model->load(Yii::$app->request->post()))
{
    Yii::$app->response->format = Response::FORMAT_JSON;
    return ActiveForm::validate($model);
}
```

Le code ci-dessus vérifie si la requête courante est une requête AJAX. Si oui, il répond à la requête en exécutant l'examen de validation et en retournant les erreurs au format JSON.

> **Info :** vous pouvez aussi utiliser la validation différée pour effectuer une validation AJAX. Néanmoins, la fonctionnalité de validation AJAX décrite ici est plus systématique et nécessite moins de codage.

Quand, à la fois `enableClientValidation` et `enableAjaxValidation` sont définies à `true`, la requête de validation AJAX est déclenchée seulement après une validation réussie côté client.

7.3 Chargement de fichiers sur le serveur

Le chargement de fichiers sur le serveur dans Yii est ordinairement effectué avec l'aide de `yii\web\UploadedFile` qui encapsule chaque fichier chargé dans un objet `UploadedFile`. Combiné avec les `yii\widgets\ActiveForm` et les modèles, vous pouvez aisément mettre en œuvre un mécanisme sûr de chargement de fichiers sur le serveur.

7.3.1 Création de modèles

Comme on le ferait avec des entrées de texte simple, pour charger un unique fichier sur le serveur, vous devez créer une classe de modèle et utliser un attribut du modèle pour conserver un instance du fichier chargé. Vous devez également déclarer une règle de validation pour valider le fichier chargé. Par exemple :

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;
```

```
class UploadForm extends Model
{
    /**
     * @var UploadedFile
     */
    public $imageFile;

    public function rules()
    {
        return [
            [['imageFile'], 'file', 'skipOnEmpty' => false, 'extensions' =>
            'png, jpg'],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            $this->imageFile->saveAs('uploads/' . $this->imageFile->baseName
            . '.' . $this->imageFile->extension);
            return true;
        } else {
            return false;
        }
    }
}
```

Dans le code ci-dessus, l'attribut `imageFile` est utilisé pur conserver une instance du fichier chargé. Il est associé à une règle de validation de fichier (`file`) qui utilise `yii\validators\FileValidator` pour garantir que l'extension du nom de fichier chargé est `png` ou `jpg`. La méthode `upload()` effectue l'examen de validation et sauvegarde le fichier sur le serveur.

Le validateur `file` vous permet de vérifier l'extension du fichier, sa taille, son type MIME, etc. Reportez-vous à la section Validateurs de noyau pour plus de détails.

> **Conseil :** si vous chargez une image sur le serveur, vous pouvez envisager l'utilisation du validateur `image` au lieu de `file`. Le validateur `image` est mis en œuvre via `yii\validators\ImageValidator` qui vérifie si un attribut a reçu une image valide qui peut être, soit sauvegardée, soit traitée en utilisant l'extension Imagine [9].

7.3.2 Rendu d'une entrée de fichier

Ensuite, créez une entrée de fichier dans une vue :

```
<?php
use yii\widgets\ActiveForm;
?>
```

9. https://github.com/yiisoft/yii2-imagine

```
<?php $form = ActiveForm::begin(['options' => ['enctype' =>
'multipart/form-data']]) ?>

    <?= $form->field($model, 'imageFile')->fileInput() ?>

    <button>Submit</button>

<?php ActiveForm::end() ?>
```

Il est important de se rappeler que vous devez ajouter l'option `enctype` au formulaire afin que le fichier soit proprement chargé sur le serveur. L'appel de `fileInput()` rend une balise `<input type="file">` qui permet à l'utilisateur de sélectionner un fichier à charger sur le serveur.

> **Conseil :** depuis la version 2.0.8, `fileInput` ajoute l'option `enctype` au formulaire automatiquement lorsqu'un champ d'entrée de fichier est utilisé.

7.3.3 Câblage

Maintenant dans une action de contrôleur, écrivez le code de câblage entre le modèle et la vue pour mettre en œuvre le chargement sur le serveur :

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFile = UploadedFile::getInstance($model,
            'imageFile');
            if ($model->upload()) {
                // le fichier a été chargé avec succès sur le serveur
                return;
            }
        }

        return $this->render('upload', ['model' => $model]);
    }
}
```

Dans le code ci-dessus, lorsque le formulaire est soumis, la méthode `yii\web\UploadedFile::getInstance()` est appelée pour représenter le fichier

chargé sous forme d'instance de `UploadedFile`. Nous comptons ensuite sur la validation du modèle pour garantir que le fichier chargé est valide et le sauvegarder sur le serveur.

7.3.4 Chargement sur le serveur de plusieurs fichiers

Vous pouvez également charger sur le serveur plusieurs fichiers à la fois, avec quelques ajustements au code présenté dans les sous-sections précédentes.

Tout d'abord, vous devez ajuster la classe du modèle en ajoutant l'option `maxFiles` dans la règle de validation de `file` pour limiter le nombre maximum de fichiers à charger simultanément. Définir `maxFiles` à `0` signifie que ce nombre n'est pas limité. Le nombre maximal de fichiers que l'on peut charger simultanément est aussi limité par la directive PHP `max_file_uploads` [10], dont la valeur par défaut est 20. La méthode `upload()` doit aussi être modifiée pour permettre la sauvegarde des fichiers un à un.

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile[]
     */
    public $imageFiles;

    public function rules()
    {
        return [
            [['imageFiles'], 'file', 'skipOnEmpty' => false, 'extensions' =>
            'png, jpg', 'maxFiles' => 4],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            foreach ($this->imageFiles as $file) {
                $file->saveAs('uploads/' . $file->baseName . '.' .
                $file->extension);
            }
            return true;
        } else {
            return false;
        }
    }
}
```

10. https://www.php.net/manual/fr/ini.core.php#ini.max-file-uploads

Dans le fichier de vue, vous devez ajouter l'option `multiple` à l'appel de `fileInput()` afin que le champ d'entrée puisse recevoir plusieurs fichiers :

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' =>
'multipart/form-data']]) ?>

    <?= $form->field($model, 'imageFiles[]')->fileInput(['multiple' =>
true, 'accept' => 'image/*']) ?>

    <button>Submit</button>

<?php ActiveForm::end() ?>
```

Pour finir, dans l'action du contrôleur, vous devez appeler `UploadedFile::getInstances()` au lieu de `UploadedFile::getInstance()` pour assigner un tableau d'instances de `UploadedFile` à `UploadForm::imageFiles`.

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFiles = UploadedFile::getInstances($model,
            'imageFiles');
            if ($model->upload()) {
                // file is uploaded successfully
                return;
            }
        }

        return $this->render('upload', ['model' => $model]);
    }
}
```

7.4 Obtenir des données pour plusieurs modèles

Lorsque vous avez affaire à des données complexes, il est possible que vous ayez besoin d'utiliser plusieurs modèles différents pour collecter des saisies de

l'utilisateur. Par exemple, en supposant que les informations de connexion de l'utilisateur sont stockées dans la table `user` tandis que les informations de son profil sont stockées dans la table `profil`, vous désirez peut-être collecter les données de l'utilisateur via un modèle `User` et un modèle `Profile`. Avec la prise en charge par Yii des modèles et des formulaires, vous pouvez résoudre ce problème d'une façon qui ne diffère qu'assez peu de celle consistant à utiliser un modèle unique.

Dans ce qui suit, nous montrons comment créer un formulaire que permet la collecte de données pour les deux modèles, `User` et `Profile`, à la fois.

Tout d'abord, l'action de contrôleur pour la collecte des données de connexion (user) et des données de profil, peut être écrite comme suit :

```
namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use app\models\User;
use app\models\Profile;

class UserController extends Controller
{
    public function actionUpdate($id)
    {
        $user = User::findOne($id);
        if (!$user) {
            throw new NotFoundHttpException("The user was not found.");
        }

        $profile = Profile::findOne($user->profile_id);

        if (!$profile) {
            throw new NotFoundHttpException("The user has no profile.");
        }

        $user->scenario = 'update';
        $profile->scenario = 'update';

        if ($user->load(Yii::$app->request->post()) &&
        $profile->load(Yii::$app->request->post())) {
            $isValid = $user->validate();
            $isValid = $profile->validate() && $isValid;
            if ($isValid) {
                $user->save(false);
                $profile->save(false);
                return $this->redirect(['user/view', 'id' => $id]);
            }
        }

        return $this->render('update', [
            'user' => $user,
```

```
            'profile' => $profile,
        ]);
    }
}
```

Dans l'action `update`, nous commençons par charger les données des modèles, `$user` et `$profile`, à mettre à jour dans la base de données. Puis nous appelons `yii\base\Model::load()` pour remplir les deux modèles avec les entrées de l'utilisateur. Si tout se passe bien, nous validons les deux modèles et les sauvegardons. Autrement, nous rendons la vue `update` avec le contenu suivant :

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'user-update-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($user, 'username') ?>

    ...autres champs de saisie...

    <?= $form->field($profile, 'website') ?>

    <?= Html::submitButton('Update', ['class' => 'btn btn-primary']) ?>
<?php ActiveForm::end() ?>
```

Comme vous le voyez, la vue `update` rend les champs de saisie de deux modèles `$user` et `$profile`.

Chapitre 8

Afficher les données

8.1 Formatage des données

Pour afficher des données dans un format plus facile à lire par les utilisateurs, vous pouvez les formater en utilisant le composant d'application `formatter`. Par défaut, le formateur est mis en œuvre par `yii\i18n\Formatter` qui fournit un jeu de méthodes pour formater des données telles que des dates, des temps, des nombres, des monnaies et autres données couramment utilisées. Vous pouvez utiliser le formateur de la manière indiquée ci-dessous :

```
$formatter = \Yii::$app->formatter;

// affiche : January 1, 2014
echo $formatter->asDate('2014-01-01', 'long');

// affiche : 12.50%
echo $formatter->asPercent(0.125, 2);

// affiche : <a href="mailto:cebe@example.com">cebe@example.com</a>
echo $formatter->asEmail('cebe@example.com');

// affiche : Yes
echo $formatter->asBoolean(true);
// il prend aussi en charge l'affichage de valeurs nulles :

// affiche : (Not set)
echo $formatter->asDate(null);
```

Comme vous pouvez le voir, ces trois méthodes sont nommées selon le format suivant `asXyz()`, où `Xyz` représente un format pris en charge. En alternative, vous pouvez formater les données en utilisant la méthode générique `format()`, qui vous permet de contrôler le format désiré par programmation et qui est communément utilisé par les composants graphiques tels que `yii\grid\GridView` et `yii\widgets\DetailView`. Par exemple :

```
// affiche : January 1, 2014
echo Yii::$app->formatter->format('2014-01-01', 'date');
```

```
// vous pouvez aussi utiliser un tableau pour spécifier les paramètres de
votre méthode de formatage :
// `2` est la valeur du paramètre `$decimals` (nombre de décimales) pour la
méthode asPercent().
// affiche : 12.50%
echo Yii::$app->formatter->format(0.125, ['percent', 2]);
```

> **Note :** le composant de formatage est conçu pour formater des valeurs à présenter à l'utilisateur. Si vous voulez convertir des entrées utilisateur en un format lisible par la machine, ou simplement formater une date dans un format lisible par la machine, le formateur n'est pas l'outil adapté à cela. Pour convertir une entrée utilisateur pour une date et un temps, vous pouvez utiliser `yii\validators\DateValidator` et `yii\validators\NumberValidator` respectivement. Pour une simple conversion entre les formats lisibles par la machine de date et de temps, la fonction PHP date() [1] suffit.

8.1.1 Configuration du formateur

Vous pouvez configurer les règles de formatage en configurant le composant `formatter` dans la configuration de l'application. Par exemple :

```
return [
    'components' => [
        'formatter' => [
            'dateFormat' => 'dd.MM.yyyy',
            'decimalSeparator' => ',',
            'thousandSeparator' => ' ',
            'currencyCode' => 'EUR',
        ],
    ],
];
```

Reportez-vous à la classe `yii\i18n\Formatter` pour connaître les propriétés qui peuvent être configurées.

8.1.2 Formatage de valeurs de dates et de temps

Le formateur prend en charge les formats de sortie suivants en relation avec les dates et les temps :

- `date` : la valeur est formatée sous la forme d'une date, p. ex. `January 01, 2014`.
- `time` : la valeur est formatée sous la forme d'un temps, p. ex. `14:23`.
- `datetime` : la valeur est formatée sous la forme d'une date et d'un temps, p. ex. `January 01, 2014 14:23`.

1. https://www.php.net/manual/fr/function.date.php

— `timestamp` : la valeur est formatée sous la forme d'un horodatage unix[2], p. ex. `1412609982`.
— `relativeTime` : la valeur est formatée sous la forme d'un intervalle de temps entre un temps et le temps actuel dans une forme lisible par l'homme, p.ex. `1 hour ago`.
— `duration` : la valeur est formatée comme une durée dans un format lisible par l'homme, p. ex. `1 day, 2 minutes`.

Les formats par défaut pour les dates et les temps utilisés pour les méthodes `date`, `time`, et `datetime` peuvent être configurés globalement en configurant `dateFormat`, `timeFormat`, et `datetimeFormat`.

Vous pouvez spécifier les formats de date et de temps en utilisant la syntaxe ICU[3]. Vous pouvez aussi utiliser la syntaxe date() de PHP[4] avec le préfixe `php:` pour la différentier de la syntaxe ICU. Par exemple :

```
// format ICU
echo Yii::$app->formatter->asDate('now', 'yyyy-MM-dd'); // 2014-10-06

// format date() de PHP
echo Yii::$app->formatter->asDate('now', 'php:Y-m-d'); // 2014-10-06
```

Lorsque vous travaillez avec des applications qui requièrent une prise en charge de plusieurs langues, vous devez souvent spécifier différents formats de dates et de temps pour différentes locales. Pour simplifier cette tâche, vous pouvez utiliser les raccourcis de formats (p. ex. `long`, `short`), à la place. Le formateur transforme un raccourci de formats en un format approprié en prenant en compte la `locale` courante. Les raccourcis de formats suivants sont pris en charge (les exemples supposent que `en_GB` est la locale courante) :

— `short` : affiche `06/10/2014` pour une date et `15:58` pour un temps ;
— `medium` : affiche `6 Oct 2014` et `15:58:42` ;
— `long` : affiche `6 October 2014` et `15:58:42 GMT` ;
— `full` : affiche `Monday, 6 October 2014` et `15:58:42 GMT`.

Depuis la version 2.0.7, il est aussi possible de formater les dates dans différents systèmes calendaires. Reportez-vous à la documentation de l'API pour la propriété `$calendar` des formateurs pour savoir comment définir un autre système calendaire.

Fuseaux horaires

Lors du formatage des dates et des temps, Yii les convertit dans le `fuseau horaire` cible. La valeur à formater est supposée être donnée en UTC, sauf si un fuseau horaire est explicitement défini ou si vous avez configuré `yii\i18n\Formatter::$defaultTimeZone`.

2. https://fr.wikipedia.org/wiki/Heure_Unix
3. https://unicode-org.github.io/icu/userguide/format_parse/datetime/
4. https://www.php.net/manual/fr/function.date.php

Dans les exemples qui suivent, nous supposons que la cible `fuseau horaire` est définie à `Europe/Berlin`.

```
// formatage d'un horodatage UNIX comme un temps
echo Yii::$app->formatter->asTime(1412599260); // 14:41:00

// formatage d'une chaîne de caractère date-temps (en UTC) comme un temps
echo Yii::$app->formatter->asTime('2014-10-06 12:41:00'); // 14:41:00

// formatage d'une chaîne de caractères date-temps (en CEST) comme un temps
echo Yii::$app->formatter->asTime('2014-10-06 14:41:00 CEST'); // 14:41:00
```

> **Note :** comme les fuseaux horaires sont assujettis à des règles fixées par les gouvernements du monde entier, et que ces règles peuvent varier fréquemment, il est vraisemblable que vous n'ayez pas la dernière information dans la base de données des fuseaux horaires installée sur votre système. Vous pouvez vous reporter au manuel d'ICU [5] pour des informations sur la manière de mettre cette base de données à jour. Reportez-vous aussi au tutoriel Configurer votre environnement PHP pour l'internationalisation.

8.1.3 Formatage des nombres

Pour les nombres, le formateur prend en charge les formats de sortie suivants :

- `integer` : la valeur est formatée comme un entier, p. ex. `42`.
- `decimal` : la valeur est formatée comme un nombre décimal en portant attention aux décimales et aux séparateurs de milliers, p. ex. `2,542.123` ou `2.542,123`.
- `percent` : la valeur est formatée comme un pourcentage p. ex. `42%`.
- `scientific` : la valeur est formatée comme un nombre dans le format scientifique p. ex. `4.2E4`.
- `currency` : la valeur est formatée comme une valeur monétaire, p. ex. `£420.00`. Notez que pour que cette fonction fonctionne correctement, la locale doit inclure la partie correspondant au pays p. ex. `en_GB` ou `en_US` parce que la partie langue seulement reste ambigüe dans ce cas.
- `size` : la valeur, qui est un nombre d'octets est formatée sous une forme lisible par l'homme, p. ex. `410 kibibytes`.
- `shortSize` : est la version courte de `size`, e.g. `410 KiB`.

Le format pour un nombre peut être ajusté en utilisant `decimalSeparator (séparateur de décimales)` et `thousandSeparator (séparateur de milliers)` , qui prennent tous les deux les valeurs par défaut déterminées par la `locale` courante.

5. https://unicode-org.github.io/icu/userguide/datetime/timezone/#updating-the-time-zone-data

Pour une configuration plus avancée, `yii\i18n\Formatter::$numberFormatterOptions` et `yii\i18n\Formatter::$numberFormatterTextOptions` peuvent être utilisés pour configurer la classe NumberFormater (formateur de nombres) [6] utilisée en interne pour implémenter le formateur. Par exemple, pour ajuster la valeur minimum et maximum des chiffres fractionnaires, vous pouvez configurer la propriété `yii\i18n\Formatter::$numberFormatterOptions` comme ceci :

```
'numberFormatterOptions' => [
    NumberFormatter::MIN_FRACTION_DIGITS => 0,
    NumberFormatter::MAX_FRACTION_DIGITS => 2,
]
```

8.1.4 Autres formats

En plus des formats de date, temps et nombre, Yii prend aussi en charge les autres formats communément utilisés, y compris :

- `raw` : la valeur est affichée telle quelle, il s'agit d'un pseudo-formateur qui n'a pas d'effet, à l'exception des valeurs `null` qui sont affichées en utilisant la propriété `nullDisplay`.
- `text` : la valeur est encodée HTML. C'est le format par défaut utilisé par les données des colonnes du widget GridView.
- `ntext` : la valeur est formatée comme un texte simple encodé HTML avec conversion des retours à la ligne en balise break.
- `paragraphs` : la valeur est formatée comme un paragraphe de texte encodé HTML à l'intérieur d'une balise `<p>`.
- `html` : la valeur est purifiée en utilisant `HtmlPurifier` pour éviter les attaques XSS. Vous pouvez passer les options additionnelles telles que `['html', ['Attr.AllowedFrameTargets' => ['_blank']]]`.
- `email` : la valeur est encodé comme un lien `mailto`.
- `image` : la valeur est formatée comme une balise image.
- `url` : la valeur est formatée comme un hyperlien.
- `boolean` : la valeur est formatée comme une valeur booléenne. Par défaut `true` est rendu par `Yes` et `false` par `No`, traduit dans la langue courante de l'application. Vous pouvez ajuster cela en configurant la propriété `yii\i18n\Formatter::$booleanFormat`.

8.1.5 Valeurs nulles (null)

Les valeurs *null* sont formatées spécialement. Au lieu d'afficher une chaîne de caractères vide, le formateur la convertit en une chaîne de caractères prédéfinie dont la valeur par défaut est `(not set)` traduite dans la langue courante de l'application. Vous pouvez configurer la propriété `nullDisplay` pour personnaliser cette chaîne de caractères.

6. https://www.php.net/manual/fr/class.numberformatter.php

8.1.6 Localisation des formats de données

Comme nous l'avons mentionné précédemment, le formateur utilise la `locale` courante pour déterminer comment formater une valeur qui soit convenable dans la cible pays/région. Par exemple, la même valeur de date est formatée différemment pour différentes locales :

```
Yii::$app->formatter->locale = 'en-US';
echo Yii::$app->formatter->asDate('2014-01-01'); // affiche : January 1,
2014

Yii::$app->formatter->locale = 'de-DE';
echo Yii::$app->formatter->asDate('2014-01-01'); // affiche : 1. Januar
2014

Yii::$app->formatter->locale = 'ru-RU';
echo Yii::$app->formatter->asDate('2014-01-01'); // affiche : 1 января 2014
г.
```

Par défaut, la `locale` est déterminée par la valeur de `yii\base\Application::$language`. Vous pouvez la redéfinir en définissant la propriété `yii\i18n\Formatter::$locale` explicitement.

> **Note :** le formateur de Yii a besoin de l'extension intl de PHP [7] pour prendre en charge la localisation des formats de données. Parce que différentes versions de la bibliothèque ICU compilées par PHP produisent des résultats de formatage différents, il est recommandé que vous utilisiez la même version de la bibliothèque ICU pour tous vos environnements. Pour plus de détails, reportez-vous au tutoriel Configuration de votre environnement PHP pour l'internationalisation.
>
> Si l'extension intl extension n'est pas installée, les données ne sont pas localisées.
>
> Notez que pour les valeurs de dates qui sont antérieures à l'année 1901, ou postérieures à 2038, la localisation n'est pas faite sur les systèmes 32 bits, même si l'extension intl est installée. Cela est dû au fait que, dans ce cas, ICU utilise des horodatages UNIX 32 bits pour les valeurs de date.

8.2 Pagination

Lorsqu'il y a trop de données à afficher sur une seule page, une stratégie courante consiste à les afficher en de multiples pages, et sur chacune des pages, à n'afficher qu'une fraction réduite des données. Cette stratégie est connue sous le nom de *pagination*.

7. https://www.php.net/manual/fr/book.intl.php

Yii utilise un objet `yii\data\Pagination` pour représenter les informations d'un schéma de pagination. En particulier :

— `nombre total (` spécifie le nombre total d'items de données. Notez que cela est ordinairement beaucoup plus élevé que le nombre d'items de données que l'on a besoin d'afficher sur une unique page.

— `taille de la page (` spécifie combien d'items de données chaque page contient. La valeur par défaut est 20.

— `page courante (` donne la numéro de la page courante (qui commence à zéro). La valeur par défaut est 0, ce qui indique la première page.

Avec un objet `yii\data\Pagination` pleinement spécifié, vous pouvez retrouver et afficher partiellement des données. Par exemple, si vous allez chercher des données dans une base de données, vous pouvez spécifier les clauses `OFFSET` et `LIMIT` de la requête de base de données avec les valeurs correspondantes fournies par l'objet pagination. Un exemple est présenté ci-dessous.

```
use yii\data\Pagination;

// construit une requêt de base de données pour obtenir tous les articles
dont le *status* vaut 1
$query = Article::find()->where(['status' => 1]);

// obtient le nombre total d'articles (mais ne va pas chercher les données
articles pour le moment)
$count = $query->count();

// crée un objet pagination en lui passant le nombre total d'items
$pagination = new Pagination(['totalCount' => $count]);

// limite la requête en utilisant l'objet pagination et va chercher les
articles
$articles = $query->offset($pagination->offset)
    ->limit($pagination->limit)
    ->all();
```

Mais quelle page d'article est retournée par l'exemple ci-dessus ? Cela dépend d'un paramètre de la requête nommé `page`. Par défaut, l'objet pagination essaye de définir le paramètre `page` avec la valeur de la `page courante (`. Si le paramètre n'est pas fourni, il prend la valeur par défaut `0`.

Pour faciliter la construction des élément de l'interface utilisateur qui prennent en charge la pagination, Yii fournit le composant graphique `yii\widgets\LinkPager` qui affiche une liste de boutons de page sur lesquels l'utilisateur peut cliquer pour préciser quelle page de données doit être affichée. Ce composant graphique accepte en paramètre un objet pagination afin de savoir quelle est la page courante et combien de boutons de page afficher. Par exemple :

```
use yii\widgets\LinkPager;
```

```
echo LinkPager::widget([
    'pagination' => $pagination,
]);
```

Si vous voulez construire des éléments d'interface graphique à la main, vous pouvez utiliser `yii\data\Pagination::createUrl()` pour créer des URL qui conduisent à différentes pages. La méthode requiert un paramètre de page et crée une URL formatée correctement qui contient le paramètre de page. Par exemple :

```
// spécifie la route que l'URL à créer doit utiliser,
// si vous ne la spécifiez pas, la route actuellement requise est utilisée
$pagination->route = 'article/index';

// affiche : /index.php?r=article%2Findex&page=100
echo $pagination->createUrl(100);

// affiche : /index.php?r=article%2Findex&page=101
echo $pagination->createUrl(101);
```

> **Conseil :** vous pouvez personnaliser le nom du paramètre de requête `page` en configurant la propriété `pageParam` lors de la création de l'objet pagination.

8.3 Tri

Lors de l'affichage de multiples lignes de données, on a souvent besoin des trier les données en fonction des valeurs de certaines colonnes spécifiées par l'utilisateur. Yii utilise l'objet `yii\data\Sort` pour représenter les information sur le schéma de triage. En particulier :

- `attributes` spécifie les *attributs* grâce auxquels les données peuvent être triées. Un attribut peut être aussi simple qu'un attribut de modèle. Il peut aussi être un composite combinant le multiples attributs de modèles ou de colonnes de base de données. Nous apportons des informations plus détaillées dans la suite de cette page.
- `attributeOrders` fournit la direction de l'ordre de tri pour chacun des attributs.
- `orders` fournit les directions de tri en terme de colonnes de bas niveau.

Pour utiliser `yii\data\Sort`, commencez par déclarer quels attributs peuvent être triés. Puis retrouvez les informations d'ordre de tri courantes de `attributeOrders` ou `orders`, et utilisez-les pour personnaliser votre requête de données. Par exemple :

```
use yii\data\Sort;

$sort = new Sort([
```

```
    'attributes' => [
        'age',
        'name' => [
            'asc' => ['first_name' => SORT_ASC, 'last_name' => SORT_ASC],
            'desc' => ['first_name' => SORT_DESC, 'last_name' => SORT_DESC],
            'default' => SORT_DESC,
            'label' => 'Name',
        ],
    ],
]);

$articles = Article::find()
    ->where(['status' => 1])
    ->orderBy($sort->orders)
    ->all();
```

Dans l'exemple qui précède, deux attributs sont déclarés pour l'objet `Sort` : `age` et `name`.

L'attribut `age` est un attribut *simple* correspondant à l'attribut `age` de la classe d'enregistrement actif `Article`. Il équivaut à la déclaration suivante :

```
'age' => [
    'asc' => ['age' => SORT_ASC],
    'desc' => ['age' => SORT_DESC],
    'default' => SORT_ASC,
    'label' => Inflector::camel2words('age'),
]
```

L'attribut `name` est un attribut *composite* défini par `first_name` et `last_name` de la classe `Article`. Il est déclaré en utilisant la structure de tableau suivante :

— Les éléments `asc` et `desc` spécifient comment trier selon l'attribut dans la direction croissante et décroissante, respectivement. Leurs valeurs représentent les colonnes réelles et les directions dans lesquelles les données sont triées. Vous pouvez spécifier une ou plusieurs colonnes pour préciser un tri simple ou un tri composite.

— L'élément `default` spécifie la direction dans laquelle l'attribut doit être trié lorsqu'il est initialement requis. Sa valeur par défaut est l'ordre croissant, ce qui signifie que si les données n'ont pas été triées auparavant et que vous demandez leur tri par cet attribut, elles sont triées par cette attribut dans la direction croissante.

— L'élément `label` spécifie quelle étiquette doit être utilisée lors de l'appel de `yii\data\Sort::link()` pour créer un lien de tri. Si cet élément n'est pas spécifié, la fonction `yii\helpers\Inflector::camel2words()` est appelée pour générer une étiquette à partir du nom de l'attribut. Notez que cette étiquette n'est pas encodée HTML.

  Info : vous pouvez fournir la valeur de `orders` à la requête de base de données pour construire sa clause `ORDER BY`. N'utilisez pas `attributeOrders`, parce que certains attributs peuvent être

composites et ne peuvent pas être reconnus par la requête de base de données.

Vous pouvez appeler `yii\data\Sort::link()` pour générer un hyperlien sur lequel l'utilisateur peut cliquer pour demander le tri des données selon l'attribut spécifié. Vous pouvez aussi appeler `yii\data\Sort::createUrl()` pour créer une URL susceptible d'être triée. Par exemple :

```
// spécifie la route que l'URL à créer doit utiliser,
// si vous ne la spécifiez pas, la route couramment requise est utilisée
$sort->route = 'article/index';

// affiche des liens conduisant à trier par *name* (nom) et *age*,
respectivement
echo $sort->link('name') . ' | ' . $sort->link('age');

// affiche : /index.php?r=article%2Findex&sort=age
echo $sort->createUrl('age');
```

`yii\data\Sort` vérifie le paramètre `sort` pour savoir quels attributs sont requis pour le tri. Vous pouvez spécifier un ordre de tri par défaut via `yii\data\Sort::$defaultOrder` lorsque le paramètre de requête est absent. Vous pouvez aussi personnaliser le nom du paramètre de requête en configurant la porpriété `sortParam`.

8.4 Fournisseurs de données

Dans les sections Pagination et Tri, nous avons décrit comment permettre à l'utilisateur de choisir une page particulière de données à afficher et de trier ces données en fonction de certaines colonnes. Comme les tâches de pagination et de tri sont très courantes, Yii met à votre disposition un jeu de classes *fournisseurs de données* pour les encapsuler.

Un fournisseur de données est une classe qui implémente l'interface `yii\data\DataProviderInterface`. Il prend en essentiellement en charge l'extraction de données paginées et triées. Il fonctionne ordinairement avec des composants graphiques de données pour que l'utilisateur final puisse paginer et trier les données de manière interactive.

Les classes fournisseurs de données suivantes sont incluses dans les versions publiées de Yii :

— `yii\data\ActiveDataProvider` : utilise `yii\db\Query` ou `yii\db\ActiveQuery` pour demander des données à des bases de données et les retourner sous forme de tableaux ou d'instances d'enregistrement actif.
— `yii\data\SqlDataProvider` : exécute une instruction SQL et retourne les données sous forme de tableaux.
— `yii\data\ArrayDataProvider` : prend un gros tableau et en retourne une tranche en se basant sur les spécifications de pagination et de tri.

Tous ces fournisseurs de données sont utilisés selon un schéma commun :

```
// créer le fournisseur de données en configurant ses propriétés de
pagination et de tri
$provider = new XyzDataProvider([
    'pagination' => [...],
    'sort' => [...],
]);

// retrouver les données paginées et triées
$models = $provider->getModels();

// obtenir le nombre d'items de données dans la page courante
$count = $provider->getCount();

// obtenir le nombre total d'items de données de l'ensemble des pages
$totalCount = $provider->getTotalCount();
```

Vous spécifiez les comportements de pagination et de tri d'un fournisseur de données en configurant ses propriétés `pagination` et `sort (tri)` qui correspondent aux configurations de `yii\data\Pagination` et `yii\data\Sort`, respectivement. Vous pouvez également les configurer à `false` pour désactiver la pagination et/ou le tri.

Les composants graphiques de données, tels que `yii\grid\GridView`, disposent d'une propriété nommée `dataProvider` qui accepte une instance de fournisseur de données et affiche les données qu'il fournit. Par exemple :

```
echo yii\grid\GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

Ces fournisseurs de données varient essentiellement en fonction de la manière dont la source de données est spécifiée. Dans les sections qui suivent, nous expliquons l'utilisation détaillée de chacun des ces fournisseurs de données.

8.4.1 Fournisseur de données actif

Pour utiliser le `fournisseur de données actif (classe` , vous devez configurer sa propriété `query`. Elle accepte soit un objet `yii\db\Query`, soit un objet `yii\db\ActiveQuery`. Avec le premier, les données peuvent être soit des tableaux, soit des instances d'enregistrement actif. Par exemple :

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
    'pagination' => [
        'pageSize' => 10,
```

```
    ],
    'sort' => [
        'defaultOrder' => [
            'created_at' => SORT_DESC,
            'title' => SORT_ASC,
        ]
    ],
]);

// retourne un tableau d'objets Post
$posts = $provider->getModels();
```

Si la requête `$query` de l'exemple ci-dessus est créée en utilisant le code suivant, alors le fournisseur de données retourne des tableaux bruts.

```
use yii\db\Query;

$query = (new Query())->from('post')->where(['status' => 1]);
```

> **Note :** si une requête spécifie déjà la clause `orderBy`, les nouvelles instructions de tri données par l'utilisateur final (via la configuration `sort`) sont ajoutées à la clause `orderBy` existante. Toute clause `limit` et `offset` existante est écrasée par la requête de pagination de l'utilisateur final (via la configuration `pagination`). Par défaut, `yii\data\ActiveDataProvider` utilise le composant d'application `db` comme connexion à la base de données. Vous pouvez utiliser une connexion différente en configurant la propriété `yii\data\ActiveDataProvider::$db`.

8.4.2 Fournisseur de données SQL

`yii\data\SqlDataProvider` travaille avec des instructions SQL brutes pour aller chercher les données. Selon les spécifications de `sort` et de `pagination`, le fournisseur ajuste les clauses `ORDER BY` et `LIMIT` de l'instruction SQL en conséquence pour n'aller chercher que la page de données requise dans l'ordre désiré.

Pour utiliser `yii\data\SqlDataProvider`, vous devez spécifier la propriété `sql`, ainsi que la propriété `totalCount`. Par exemple :

```
use yii\data\SqlDataProvider;

$count = Yii::$app->db->createCommand('
    SELECT COUNT(*) FROM post WHERE status=:status
', [':status' => 1])->queryScalar();

$provider = new SqlDataProvider([
    'sql' => 'SELECT * FROM post WHERE status=:status',
    'params' => [':status' => 1],
    'totalCount' => $count,
```

```
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => [
            'title',
            'view_count',
            'created_at',
        ],
    ],
]);

// retourne un tableau de lignes de données
$models = $provider->getModels();
```

> **Info :** la propriété `totalCount` est requise seulement si vous avez besoin de paginer les données.Cela est dû au fait que l'instruction SQL spécifiée via `sql` est modifiée par le fournisseur pour ne retourner que la page de données couramment requise. Le fournisseur a donc besoin de connaître le nombre total d'items de données pour calculer correctement le nombre de pages disponibles.

8.4.3 Fournisseur de données tableau

L'utilisation de `yii\data\ArrayDataProvider` est préférable lorsque vous travaillez avec un grand tableau. Le fournisseur vous permet de retourner une page des données du tableau, triées selon une ou plusieurs colonnes. Pour utiliser `yii\data\ArrayDataProvider`, vous devez spécifier la propriété `allModels` comme un grand tableau. Les éléments dans le grand tableau peuvent être, soit des tableaux associatifs (p. ex. des résultats de requête d'objets d'accès aux données (DAO)) ou des objets (p. ex. les instances d'Active Record. Par exemple :

```
use yii\data\ArrayDataProvider;

$data = [
    ['id' => 1, 'name' => 'name 1', ...],
    ['id' => 2, 'name' => 'name 2', ...],
    ...
    ['id' => 100, 'name' => 'name 100', ...],
];

$provider = new ArrayDataProvider([
    'allModels' => $data,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => ['id', 'name'],
```

```
    ],
]);

// obtient les lignes de la page couramment requise
$rows = $provider->getModels();
```

> **Note :** comparé au fournisseur de données actif et au fournisseur de données SQL](#sql-data-provider), le fournisseur de données tableau est moins efficient car il requiert de charger *toutes* les données en mémoire.

8.4.4 Travail avec les clés de données

Lorsque vous utilisez les items de données retournés par le fournisseur de données, vous avez souvent besoin d'identifier chacun des items de données par une clé unique. Par exemple, si les items de donnés représentent des informations sur un client, vous désirez peut-être utiliser l'identifiant du client en tant que clé pour chacun de lots d'informations sur un client. Les fournisseurs de données peuvent retourner une liste de telles clés correspondant aux items de données retournés par `yii\data\DataProviderInterface::getModels()`. Par exemple :

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
]);

// retourne un tableau d'objets Post
$posts = $provider->getModels();

// retourne les valeurs des clés primaires correspondant à $posts
```

Dans l'exemple ci-dessus, comme vous fournissez un objet `yii\db\ActiveQuery` à `yii\data\ActiveDataProvider`. Il est suffisamment intelligent pour retourner les valeurs de la clé primaire en tant que clés. Vous pouvez aussi spécifier comment les valeurs de la clé sont calculées en configurant `yii\data\ActiveDataProvider::$key` avec un nom de colonne ou une fonction de rappel qui calcule les valeurs de la clé. Par exemple :

```
// utilise la colonne "slug" comme valeurs de la clé
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => 'slug',
]);

// utilise le résultat de md5(id) comme valeurs de la clé
```

```
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => function ($model) {
        return md5($model->id);
    }
]);
```

8.4.5 Création d'un fournisseur de données personnalisé

Pour créer votre fournisseur de données personnalisé, vous devez implémenter `yii\data\DataProviderInterface`. Une manière plus facile est d'étendre `yii\data\BaseDataProvider`,ce qui vous permet de vous concentrer sur la logique centrale du fournisseur de données. En particulier, vous devez essentiellement implémenter les méthodes suivantes :

— `prepareModels()` : prépare les modèles de données qui seront disponibles dans la page courante et les retourne sous forme de tableau.
— `prepareKeys()` : accepte un tableau de modèles de données couramment disponibles et retourne les clés qui leur sont associés.
— `prepareTotalCount` : retourne une valeur indiquant le nombre total de modèles de données dans le fournisseur.

Nous présentons ci-dessous un exemple de fournisseur de données que lit des données CSV efficacement :

```
<?php
use yii\data\BaseDataProvider;

class CsvDataProvider extends BaseDataProvider
{
    /**
     * @var string name of the CSV file to read
     */
    public $filename;

    /**
     * @var string|callable nom de la colonne clé ou fonction de rappel la
retournant
     */
    public $key;

    /**
     * @var SplFileObject
     */
    protected $fileObject; // SplFileObject est très pratique pour
    rechercher une ligne particulière dans un fichier


    /**
     * {@inheritdoc}
     */
    public function init()
    {
```

```
        parent::init();

        // open file
        $this->fileObject = new SplFileObject($this->filename);
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareModels()
    {
        $models = [];
        $pagination = $this->getPagination();

        if ($pagination === false) {
            // dans le cas où il n'y a pas de pagination, lit toutes les
            lignes
            while (!$this->fileObject->eof()) {
                $models[] = $this->fileObject->fgetcsv();
                $this->fileObject->next();
            }
        } else {
            // s'il y a une pagination, ne lit qu'une seule page
            $pagination->totalCount = $this->getTotalCount();
            $this->fileObject->seek($pagination->getOffset());
            $limit = $pagination->getLimit();

            for ($count = 0; $count < $limit; ++$count) {
                $models[] = $this->fileObject->fgetcsv();
                $this->fileObject->next();
            }
        }

        return $models;
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareKeys($models)
    {
        if ($this->key !== null) {
            $keys = [];

            foreach ($models as $model) {
                if (is_string($this->key)) {
                    $keys[] = $model[$this->key];
                } else {
                    $keys[] = call_user_func($this->key, $model);
                }
            }

            return $keys;
        } else {
```

```
            return array_keys($models);
        }
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareTotalCount()
    {
        $count = 0;

        while (!$this->fileObject->eof()) {
            $this->fileObject->next();
            ++$count;
        }

        return $count;
    }
}
```

8.5 Composants graphiques d'affichage de données

Yii fournit un jeu de composants graphiques utilisables pour afficher des données. Tandis que le componsant graphique DetailView (vue détaillée) peut être utilisé pour afficher un enregistrement unique, les composants graphiques ListView (vue en liste) et GridView (vue en grille) peuvent être utilisés pour afficher plusieurs enregistrements en liste ou en grille assortis de fonctionnalités telles que la pagination, le tri et le filtrage.

8.5.1 Vue détaillée (classe *DetailView*)

Le composant graphique `DetailView` (vue détaillée) affiche les détails d'un `modèle` de données unique.

Il est le plus adapté à l'affichage d'un modèle dans un format courant (p. ex. chacun des attributs du modèle est affiché en tant que ligne d'une grille). Le modèle peut être, soit une instance, ou une classe fille, de `yii\base\Model` telle que la classe ActiveRecord, soit un tableau associatif.

DetailView utilise la propriété `$attributes` pour déterminer quels attributs du modèle doivent être affichés et comment ils doivent être formatés. Reportez-vous à la section formatage des données pour des informations sur les options de formatage.

Une utilisation typique de *DetailView* ressemble à ce qui suit :

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        'title',              // attribut title (en texte simple)
```

```
        'description:html',    // attribut description formaté en HTML
        [                      // le nom du propriétaire du modèle
            'label' => 'Owner',
            'value' => $model->owner->name,
        ],
        'created_at:datetime', // date de création formaté comme une
        date/temps
    ],
]);
```

8.5.2 Vue en liste (class *ListView*)

Le composant graphique `ListView` (vue en liste) est utilisé pour afficher des données issues d'un fournisseur de données. Chacun des modèles est rendu en utilisant le composant `ListView` (vue en liste) spécifié. Comme ce composant fournit des fonctionnalités telles que la pagination, le tri et le filtrage de base, il est pratique, à la fois pour afficher des informations et pour créer des interfaces utilisateur de gestion des données.

Typiquement, on l'utilise comme ceci :

```
use yii\widgets\ListView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
]);
```

Le fichier de vue, `_post`, contient ce qui suit :

```
<?php
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
?>
<div class="post">
    <h2><?= Html::encode($model->title) ?></h2>

    <?= HtmlPurifier::process($model->text) ?>
</div>
```

Dans le fichier ci-dessus, le modèle de données courant est disponible comme `$model`. En outre, les variables suivantes sont disponibles :

- `$key` : mixed, la valeur de la clé associée à l'item de données
- `$index` : integer, l'index commençant à zéro de l'item de données dans le tableau d'items retourné par le fournisseur de données.

— `$widget` : ListView, l'instance de ce composant graphique.

Si vous avez besoin de passer des données additionnelles à chacune des vues, vous pouvez utiliser la propriété `$viewParams` pour passer des paires clé valeur, comme ceci :

```
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
    'viewParams' => [
        'fullView' => true,
        'context' => 'main-page',
        // ...
    ],
]);
```

Celles-ci sont alors disponibles aussi dans la vue en tant que variables.

8.5.3 Vue en grille (classe *GridView*)

La vue en grille, ou composant `GridView`, est un des composants les plus puissants de Yii. Ce composant est extrêmement utile si vous devez rapidement construire l'interface d'administration du système. Il accepte des données d'un fournisseur de données et rend chacune des lignes en utilisant un jeu de `columns` (colonnes), présentant ainsi l'ensemble des données sous forme d'une grille.

Chacune des lignes de la grille représente un item unique de données, et une colonne représente ordinairement un attribut de l'item (quelques colonnes peuvent correspondre à des expressions complexes utilisant les attributs ou à un texte statique).

Le code minimal pour utiliser le composant *GridView* se présente comme suit :

```
use yii\grid\GridView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

Le code précédent crée un fournisseur de données, puis utilise le composant *GridView* pour afficher chacun des attributs dans une ligne en le prélevant dans le fournisseur de données. La grille affichée est doté de fonctionnalités de pagination et de tri sans autre intervention.

Colonnes de la grille

Les colonnes de la grille sont exprimées en terme de classe `yii\grid\Column`, qui sont configurées dans la propriété `columns` (colonnes) de la configuration du composant *GridView*. En fonction du type de colonne et des réglages, celles-ci sont en mesure de présenter les données différemment. La classe par défaut est `yii\grid\DataColumn` (colonne de données), qui représente un attribut de modèle et peut être triée et filtrée.

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        // colonnes simples définies par les données contenues dans le
        fournisseur de données
        // les données de la colonne du modèle sont utilisées
        'id',
        'username',
        // un exemple plus complexe
        [
            'class' => 'yii\grid\DataColumn', // peut être omis car c'est la
            valeur par défaut
            'value' => function ($data) {
                return $data->name; // $data['name'] pour une donnée tableau
                p. ex. en utilisant SqlDataProvider.
            },
        ],
    ],
]);
```

Notez que si la partie `columns` de la configuration n'est pas spécifiée, Yii essaye de montrer toutes les colonnes possibles du modèle du fournisseur de données.

Classes de colonne

Les colonnes du composant *GridView* peuvent être personnalisées en utilisant différentes classes de colonnes :

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\SerialColumn', // <-- ici
            // vous pouvez configurer des propriété additionnelles ici
        ],
```

En plus des classes de colonne fournies par Yii que nous allons passer en revue ci-après, vous pouvez créer vos propres classes de colonne.

Chacune des classes de colonne étend la classe `yii\grid\Column` afin que quelques options communes soient disponibles lors de la configuration des colonnes.

— `header` permet de définir une ligne d'entête
— `footer` permet de définir le contenu d'une ligne de pied de grille
— `visible` définit si la colonne doit être visible.
— `content` vous permet de passer une fonction de rappel PHP valide qui retourne les données d'une ligne. Le format est le suivant :

```
function ($model, $key, $index, $column) {
    return 'a string';
}
```

Vous pouvez spécifier différentes options HTML de conteneurs en passant des tableaux à :

— `headerOptions`
— `footerOptions`
— `filterOptions`
— `contentOptions`

Colonne de données (*DataColumn*) La classe `DataColumn` (colonne de données) est utilisée pour afficher et trier des données. C'est le type de colonne par défaut, c'est pourquoi la spécification de la classe peut être omise.

Le réglage principal de la colonne de données est celui de sa propriété `format`. Ses valeurs correspondent aux méthodes du composant d'application `formatter` qui est de classe `Formatter` par défaut :

```
echo GridView::widget([
    'columns' => [
        [
            'attribute' => 'name',
            'format' => 'text'
        ],
        [
            'attribute' => 'birthday',
            'format' => ['date', 'php:Y-m-d']
        ],
    ],
]);La valeur de la colonne est passée en tant que premier argument
```

Dans cet exemple, `text` correspond à la méthode `yii\i18n\Formatter::asText()`. La valeur de la colonne est passée en tant que premier argument. Dans la deuxième définition de colonne, `date` correspond à la méthode `yii\i18n\Formatter::asDate()`. La valeur de la colonne est passée en tant que premier argument tandis que 'php :Y-m-d' est utilisé en tant que valeur du deuxième argument.

Pour une liste complète de tous les formateurs, reportez-vous à la section Formatage des données.

Pour configurer des colonnes de données, il y a aussi un format raccourci qui est décrit dans la documentation de l'API de `columns`.

Colonne d'actions (*ActionColumn*) La classe `ActionColumn` (colonne d'action) affiche des boutons d'action tels que mise à jour ou supprimer pour chacune des lignes.

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\ActionColumn',
            // vous pouvez configurer des propriétés additionnelles ici
        ],
```

Les propriétés additionnelles configurables sont :

- `controller` qui est l'identifiant du contrôleur qui prend en charge l'action. Si cette propriété n'est pas définie, le contrôleur courant est utilisé.
- `template` qui définit le modèle utilisé pour composer chacune des cellules dans la colonne d'actions. Les marqueurs (textes à l'intérieur d'accolades) sont traités comme des identifiants d'action (aussi appelé *noms de bouton* dans le contexte d'une colonne d'actions. Il sont remplacés par les fonctions de rappel correspondantes spécifiées dans la propriété `buttons`. Par exemple, le marqueur `{view}` sera remplacé par le résultat de la fonction de rappel `buttons['view']`. Si une fonction de rappel n'est pas trouvée, le texte est remplacé par une chaîne vide. Les marqueurs par défaut sont `{view} {update} et {delete}`.
- `buttons` est un tableau de fonctions de rappel pour le rendu des boutons. Les clés du tableau sont les noms des boutons (sans les accolades), et les valeurs sont les fonctions de rappel de rendu des boutons. Les fonctions de rappel ont la signature suivante :

  ```
  function ($url, $model, $key) {
      // retourne le code HTML du bouton
  }
  ```

  dans le code qui précède, `$url` est l'URL que la colonne crée pour le bouton, `$model` est l'objet modèle qui est en train d'être rendu pour la ligne courante, et `$key` est la clé du modèle dans le tableau du fournisseur de données.
- `urlCreator` est une fonction de rappel qui crée une URL de bouton en utilisant les informations spécifiées sur le modèle. La signature de la fonction de rappel doit être le même que celle de `yii\grid\ActionColumn::createUrl()`. Si cette propriété n'est pas définie, les URL de bouton sont créées en utilisant `yii\grid\ActionColumn::createUrl()`.
- `visibleButtons` est un tableau des conditions de visibilité pour chacun des boutons. Les clés du tableau sont les noms des boutons (sans les accolades), et les valeurs sont les valeurs booleénnes `true` ou `false` (vrai ou faux) ou la fonction anonyme. Lorsque le nom du bouton n'est pas

spécifié dans ce tableau, il est montré par défaut. Les fonctions de rappel utilisent la signature suivante :

```
function ($model, $key, $index) {
    return $model->status === 'editable';
}
```

Ou vous pouvez passer une valeur booléenne :

```
[
    'update' => \Yii::$app->user->can('update')
]
```

Colonne boîte à cocher (*CheckboxColumn*) La classe `CheckboxColumn` (colonne de boîtes à cocher) affiche une colonne de boîtes à cocher.

Pour ajouter une colonne de boîtes à cocher à la vue en grille (*GridView*), ajoutez la configuration de `columns` comme ceci :

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        // ...
        [
            'class' => 'yii\grid\CheckboxColumn',
            // vous pouvez configurer des propriétés additionnelles ici
        ],
    ],
```

L'utilisateur peut cliquer sur les boîtes à cocher pour sélectionner des lignes dans la grille. Les lignes sélectionnées peuvent être obtenues en appelant le code JavaScript suivant :

```
var keys = $('#grid').yiiGridView('getSelectedRows');
// keys est un tableau constitué des clés associées aux lignes
sélectionnées.
```

Colonne série (*SerialColumn*) La classe `SerialColumn` (colonne série) rend les numéros de ligne en commençant à `1` et en continuant.

L'utilisation est aussi simple que ce que nous présentons ci-après :

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'], // <-- ici
        // ...
```

Tri des données

> **Note :** cette section est en cours de développement.
>
> — https://github.com/yiisoft/yii2/issues/1576

Filtrage des données

Pour filtrer les données, la vue en grille (*GridView*) requiert un modèle qui représente le critère de recherche qui est ordinairement pris dans les champs du filtre dans la vue en grille. Une pratique courante lorsqu'on utilise des enregistrements actifs est de créer une classe modèle de recherche qui fournit les fonctionnalités nécessaires (elle peut être générée pour vous par Gii). Cette classe définit les règles de validation pour la recherche et fournit une méthode `search()` (recherche) qui retourne le fournisseur de données avec une requête ajustée qui respecte les critères de recherche.

Pour ajouter la fonctionnalité de recherche au modèle `Post`, nous pouvons créer un modèle `PostSearch` comme celui de l'exemple suivant :

```
<?php

namespace app\models;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;

class PostSearch extends Post
{
    public function rules()
    {
        // seuls les champs dans rules() peuvent être recherchés
        return [
            [['id'], 'integer'],
            [['title', 'creation_date'], 'safe'],
        ];
    }

    public function scenarios()
    {
        // bypasse l'implémentation de scenarios() dans la classe parent
        return Model::scenarios();
    }

    public function search($params)
    {
        $query = Post::find();

        $dataProvider = new ActiveDataProvider([
            'query' => $query,
        ]);

        // charge les données du formulaire de recherche et valide
        if (!($this->load($params) && $this->validate())) {
            return $dataProvider;
        }

        // ajuste la requête en ajoutant les filtres
```

```
        $query->andFilterWhere(['id' => $this->id]);
        $query->andFilterWhere(['like', 'title', $this->title])
              ->andFilterWhere(['like', 'creation_date',
              $this->creation_date]);

        return $dataProvider;
    }
}
```

> **Conseil :** reportez-vous au Constructeur de requêtes (*Query Builder*) et en particulier aux conditions de filtrage pour savoir comment construire la requête de filtrage.

Vous pouvez utiliser cette fonction dans le contrôleur pour obtenir le fournisseur de données de la vue en grille :

```
$searchModel = new PostSearch();
$dataProvider = $searchModel->search(Yii::$app->request->get());

return $this->render('myview', [
    'dataProvider' => $dataProvider,
    'searchModel' => $searchModel,
]);
```

Et dans la vue, vous assignez ensuite le fournisseur de données (`$dataProvider`) et le modèle de recherche (`$searchModel`) à la vue en grille (*GridView*) :

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        // ...
    ],
]);
```

Formulaire de filtrage séparé

La plupart du temps, utiliser les filtres de l'entête de la vue en grille suffit, mais dans le cas où vous avez besoin d'un formulaire de filtrage séparé, vous pouvez facilement l'ajouter aussi. Vous pouvez créer une vue partielle `_search.php` avec le contenu suivant :

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model app\models\PostSearch */
/* @var $form yii\widgets\ActiveForm */
?>
```

```
<div class="post-search">
    <?php $form = ActiveForm::begin([
        'action' => ['index'],
        'method' => 'get',
    ]); ?>

    <?= $form->field($model, 'title') ?>

    <?= $form->field($model, 'creation_date') ?>

    <div class="form-group">
        <?= Html::submitButton('Search', ['class' => 'btn btn-primary']) ?>
        <?= Html::submitButton('Reset', ['class' => 'btn btn-default']) ?>
    </div>

    <?php ActiveForm::end(); ?>
</div>
```

et l'inclure dans la vue `index.php`, ainsi :

```
<?= $this->render('_search', ['model' => $searchModel]) ?>
```

> **Note :** si vous utilisez Gii pour générer le code des méthodes CRUD, le formulaire de filtrage séparé (`_search.php`) est généré par défaut, mais est commenté dans la vue `index.php`. Il vous suffit de supprimer la marque du commentaire pour l'utiliser !

Un formulaire de filtrage séparé est utile quand vous avez besoin de filtrer selon des champs qui ne sont pas visibles dans la vue en grille, ou pour des conditions particulières de filtrage, telles qu'une plage de dates. Pour filtrer selon une plage de dates, nous pouvons ajouter les attributs non DB `createdFrom` et`createdTo` au modèle de recherche :

```
class PostSearch extends Post
{
    /**
     * @var string
     */
    public $createdFrom;

    /**
     * @var string
     */
    public $createdTo;
}
```

Étendez les conditions de la requête dans la méthode `search()` comme ceci :

```
$query->andFilterWhere(['>=', 'creation_date', $this->createdFrom])
      ->andFilterWhere(['<=', 'creation_date', $this->createdTo]);
```

Et ajoutez les champs représentatifs au formulaire de filtrage :

```
<?= $form->field($model, 'creationFrom') ?>

<?= $form->field($model, 'creationTo') ?>
```

Travail avec des relations entre modèles

Lorsque vous affichez des enregistrements actifs dans la vue en grille, vous pouvez rencontrer le cas où vous affichez des valeurs de colonne en relation telles que le nom de l'auteur de l'article (post) au lieu d'afficher simplement son identifiant (`id`). Vous pouvez le faire en définissant le nom de l'attribut dans `yii\grid\GridView::$columns` comme étant `author.name` lorsque le modèle de l'article (`Post`) possède une relation nommée `author` (auteur) et que le modèle possède un attribut nommé `name` (nom). La vue en grille affiche alors le nom de l'auteur mais le tri et le filtrage ne sont pas actifs par défaut. Vous devez ajuster le modèle `PostSearch` que nous avons introduit dans la section précédente pour y ajouter cette fonctionnalité.

Pour activer le tri sur une colonne en relation, vous devez joindre la table en relation et ajouter la règle de tri au composant *Sort* du fournisseur de données :

```
$query = Post::find();
$dataProvider = new ActiveDataProvider([
    'query' => $query,
]);

// joingnez avec la relation nommée `author` qui est une relation avec la
table `users`
// et définissez l'alias à `author`
$query->joinWith(['author' => function($query) { $query->from(['author' =>
'users']); }]);
// depuis la version 2.0.7, l'écriture ci-dessus peut être simplifiée en
$query->joinWith('author AS author');
// active le tri pour la colonne en relation
$dataProvider->sort->attributes['author.name'] = [
    'asc' => ['author.name' => SORT_ASC],
    'desc' => ['author.name' => SORT_DESC],
];

// ...
```

Le filtrage nécessite aussi l'appel de la fonction *joinWith* ci-dessus. Vous devez également autoriser la recherche sur la colonne dans les attributs et les règles comme ceci :

```
public function attributes()
{
    // ajoute les champs en relation avec les attributs susceptibles d'être
    cherchés
```

```
    return array_merge(parent::attributes(), ['author.name']);
}

public function rules()
{
    return [
        [['id'], 'integer'],
        [['title', 'creation_date', 'author.name'], 'safe'],
    ];
}
```

Dans `search()`, il vous suffit ensuite d'ajouter une autre condition de filtrage avec :

```
$query->andFilterWhere(['LIKE', 'author.name',
$this->getAttribute('author.name')]);
```

> **Info :** dans ce qui précède, nous utilisons la même chaîne de caractères pour le nom de la relation et pour l'alias de table ; cependant, lorsque votre nom de relation et votre alias diffèrent, vous devez faire attention aux endroits où vous utilisez l'alias et à ceux où vous utilisez le nom de la relation. Une règle simple pour cela est d'utiliser l'alias partout où cela sert à construire le requête de base de données et le nom de la relation dans toutes les autres définitions telles que `attributes()` et `rules()` etc.
>
> Par exemple, si vous utilisez l'alias `au` pour la table auteur en relation, l'instruction *joinWith* ressemble à ceci :
>
> ```
> $query->joinWith(['author au']);
> ```
>
> Il est également possible d'appeler simplement `$query->joinWith(['author']);` lorsque l'alias est défini dans la définition de la relation.
>
> L'alias doit être utilisé dans la condition de filtrage mais le nom d'attribut reste le même :
>
> ```
> $query->andFilterWhere(['LIKE', 'au.name',
> $this->getAttribute('author.name')]);
> ```
>
> La même chose est vraie pour la définition du tri :
>
> ```
> $dataProvider->sort->attributes['author.name'] = [
>     'asc' => ['au.name' => SORT_ASC],
>     'desc' => ['au.name' => SORT_DESC],
> ];
> ```
>
> Également, lorsque vous spécifiez la propriété `defaultOrder` (ordre de tri par défaut) pour le tri, vous avez besoin d'utiliser le nom de la relation au lieu de l'alias :
>
> ```
> $dataProvider->sort->defaultOrder = ['author.name' => SORT_ASC];
> ```
>
> **Info :** pour plus d'informations sur `joinWith` et sur les requêtes effectuées en arrière-plan, consultez la documentation sur l'enregistrement actif à la section Jointure avec des relations.

Utilisation de vues SQL pour le filtrage, le tri et l'affichage des données Il existe une autre approche qui peut être plus rapide et plus utile – les vues SQL. Par exemple, si vous avez besoin d'afficher la vue en grille avec des utilisateurs et leur profil, vous pouvez le faire de cette manière :

```
CREATE OR REPLACE VIEW vw_user_info AS
    SELECT user.*, user_profile.lastname, user_profile.firstname
    FROM user, user_profile
    WHERE user.id = user_profile.user_id
```

Ensuite vous devez créer l'enregistrement actif qui représente cette vue :

```
namespace app\models\views\grid;

use yii\db\ActiveRecord;

class UserView extends ActiveRecord
{

    /**
     * {@inheritdoc}
     */
    public static function tableName()
    {
        return 'vw_user_info';
    }

    public static function primaryKey()
    {
        return ['id'];
    }

    /**
     * {@inheritdoc}
     */
    public function rules()
    {
        return [
            // définissez vos règle ici
        ];
    }

    /**
     * {@inheritdoc}
     */
    public function attributeLabels()
    {
        return [
            // définissez vos étiquettes d'attribut ici
        ];
    }


}
```

Après cela, vous pouvez utiliser l'enregistrement actif *UserView* dans vos modèle de recherche, sans spécification additionnelle d'attribut de tri et de filtrage. Tous les attributs fonctionneront directement. Notez que cette approche a ses avantages et ses inconvénients :

— vous n'avez pas besoin de spécifier des conditions de tri et de filtrage. Tout fonctionne d'emblée ;
— cela peut être beaucoup plus rapide à cause de la taille des données et du nombre de requêtes SQL effectuées (pour chacune des relations vous n'avez pas besoin de requête supplémentaire) ;
— comme cela n'est qu'une simple mise en relation de l'interface utilisateur avec la vue SQL, il lui manque un peu de la logique qui apparaît dans vos entités, ainsi, si vous avez des méthodes comme `isActive`, `isDeleted` ou autres qui influencent l'interface utilisateur, vous devez les dupliquer dans cette classe également.

Plusieurs vues en grille par page

Vous pouvez utiliser plus d'une vue en grille sur une page unique mais quelques éléments de configuration additionnels sont nécessaires afin qu'elles n'entrent pas en interférence entre elles. Lorsque vous utilisez plusieurs instances de la vue en grille, vous devez configurer des noms de paramètre différents pour les liens de tri et de pagination générés de manière à ce que chacune des vues en grille possède ses propres liens de tri et de pagination. Vous faites cela en définissant les paramètres `sortParam` (tri) et `pageParam` (page) des instances `sort` et `pagination` du fournisseur de données.

Supposez que vous vouliez lister les modèles `Post` et `User` pour lesquels vous avez déjà préparé deux fournisseurs de données `$userProvider` et `$postProvider` :

```
use yii\grid\GridView;

$userProvider->pagination->pageParam = 'user-page';
$userProvider->sort->sortParam = 'user-sort';

$postProvider->pagination->pageParam = 'post-page';
$postProvider->sort->sortParam = 'post-sort';

echo '<h1>Users</h1>';
echo GridView::widget([
    'dataProvider' => $userProvider,
]);

echo '<h1>Posts</h1>';
echo GridView::widget([
    'dataProvider' => $postProvider,
]);
```

Utilisation de la vue en grille avec Pjax

Le composant graphique `Pjax` vous permet de mettre à jour une certaine section de votre page plutôt que d'avoir à recharger la page toute entière. Vous pouvez l'utiliser pour mettre uniquement à jour le contenu de la `GridView` (vue en grille) lors de l'utilisation de filtres.

```
use yii\widgets\Pjax;
use yii\grid\GridView;

Pjax::begin([
    // PJax options
]);
    Gridview::widget([
        // GridView options
    ]);
Pjax::end();
```

Pjax fonctionne également pour les liens à l'intérieur du composant graphique `Pjax` et pour les liens spécifiés par `Pjax::$linkSelector`. Mais cela peut être un problème pour les liens d'une `ActionColumn` (colonne d'action). Pour empêcher cela, ajoutez l'attribut HTML `data-pjax="0"` aux liens lorsque vous définissez la propriété `ActionColumn::$buttons`.

Vue en grille et vue en liste avec Pjax dans Gii Depuis la version 2.0.5, le générateur d'actions CRUD de Gii dispose d'une option appelée `$enablePjax` qui peut être utilisée, soit via l'interface web, soit en ligne de commande.

```
yii gii/crud --controllerClass="backend\\controllers\PostController" \
  --modelClass="common\\models\\Post" \
  --enablePjax=1
```

Qui génère un composant graphique `Pjax` enveloppant les composants graphiques `GridView` ou `ListView`.

8.5.4 Lectures complémentaires

— Rendering Data in Yii 2 with GridView and ListView [8] d'Arno Slatius.

8. https://www.sitepoint.com/rendering-data-in-yii-2-with-gridview-and-listview/

Error : not existing file : output-theming.md

Chapitre 9

Securité

9.1 Authentification

L'authentification est le processus qui consiste à vérifier l'identité d'un utilisateur. Elle utilise ordinairement un identifiant (p. ex. un nom d'utilisateur ou une adresse de courriels) et un jeton secret (p. ex. un mot de passe ou un jeton d'accès) pour juger si l'utilisateur est bien qui il prétend être. L'authentification est à la base de la fonctionnalité de connexion.

Yii fournit une base structurée d'authentification qui interconnecte des composants variés pour prendre en charge la connexion. Pour utiliser cette base structurée, vous devez essentiellement accomplir les tâches suivantes :

— Configurer le composant d'application `user` ;
— Créer une classe qui implémente l'interface `yii\web\IdentityInterface`.

9.1.1 Configuration de `yii\web\User`

Le composant d'application `user` gère l'état d'authentification de l'utilisateur. Il requiert que vous spécifiiez une `classe d'identité` contenant la logique réelle d'authentification. Dans la configuration suivante de l'application, la `classe d'identité` pour `user` est configurée sous le nom `app\models\User` dont la mise en œuvre est expliquée dans la sous-section suivante :

```
return [
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
        ],
    ],
];
```

9.1.2 Mise en œuvre de `yii\web\IdentityInterface`

La `classe d'identité` doit implémenter l'interface `yii\web\IdentityInterface` qui comprend les méthodes suivantes :

- `findIdentity()` : cette méthode recherche une instance de la classe d'identité à partir de l'identifiant utilisateur spécifié. Elle est utilisée lorsque vous devez conserver l'état de connexion via et durant la session.
- `findIdentityByAccessToken()` : cette méthode recherche une instance de la classe d'identité à partir du jeton d'accès spécifié. Elle est utilisée lorsque vous avez besoin d'authentifier un utilisateur par un jeton secret (p. ex. dans une application pleinement REST sans état).
- `getId()` : cette méthode retourne l'identifiant de l'utilisateur que cette instance de la classe d'identité représente.
- `getAuthKey()` : cette méthode retourne une clé utilisée pour vérifier la connexion basée sur les témoins de connexion (*cookies*). La clé est stockée dans le témoin de connexion `login` et est ensuite comparée avec la version côté serveur pour s'assurer que le témoin de connexion est valide.
- `validateAuthKey()` : cette méthode met en œuvre la logique de vérification de la clé de connexion basée sur les témoins de connexion.

Si une méthode particulière n'est pas nécessaire, vous devez l'implémenter avec un corps vide. Par exemple, si votre application est une application sans état et pleinement REST, vous devez seulement implémenter `findIdentityByAccessToken()` et `getId()` et laisser toutes les autres méthodes avec un corps vide.

Dans l'exemple qui suit, une `classe d'identité` est mise en œuvre en tant que classe Active Record associée à la table de base de données `user`.

```
<?php

use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function tableName()
    {
        return 'user';
    }

    /**
     * Trouve une identité à partir de l'identifiant donné.
     *
     * @param string|int $id l'identifiant à rechercher
     * @return IdentityInterface|null l'objet identité qui correspond à
l'identifiant donné
     */
    public static function findIdentity($id)
    {
        return static::findOne($id);
    }

    /**
```

```
     * Trouve une identité à partir du jeton donné
     *
     * @param string $token le jeton à rechercher
     * @return IdentityInterface|null l'objet identité qui correspond au
jeton donné
     */
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }

    /**
     * @return int|string l'identifiant de l'utilisateur courant
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @return string la clé d'authentification de l'utilisateur courant
     */
    public function getAuthKey()
    {
        return $this->auth_key;
    }

    /**
     * @param string $authKey
     * @return bool si la clé d'authentification est valide pour
l'utilisateur courant
     */
    public function validateAuthKey($authKey)
    {
        return $this->getAuthKey() === $authKey;
    }
}
```

Comme nous l'avons expliqué précédemment, vous devez seulement implémenter `getAuthKey()` et `validateAuthKey()` si votre application utilise la fonctionnalité de connexion basée sur les témoins de connexion. Dans ce cas, vous devez utiliser le code suivant pour générer une clé d'authentification pour chacun des utilisateurs et la stocker dans la table `user` :

```
class User extends ActiveRecord implements IdentityInterface
{
    ......

    public function beforeSave($insert)
    {
        if (parent::beforeSave($insert)) {
            if ($this->isNewRecord) {
```

```
                $this->auth_key =
                \Yii::$app->security->generateRandomString();
            }
            return true;
        }
        return false;
    }
}
```

> **Note :** ne confondez pas la classe d'identité `User` avec la classe `yii\web\User`. La première est la classe mettant en œuvre la logique d'authentification. Elle est souvent mise en œuvre sous forme de classe Active Record associée à un moyen de stockage persistant pour conserver les éléments d'authentification de l'utilisateur. La deuxième est une classe de composant d'application qui gère l'état d'authentification de l'utilisateur.

9.1.3 Utilisation de `yii\web\User`

Vous utilisez `yii\web\User` essentiellement en terme de composant d'application `user`.

Vous pouvez détecter l'identité de l'utilisateur courant en utilisant l'expression `Yii::$app->user->identity`. Elle retourne une instance de la `classe d'identité` représentant l'utilisateur connecté actuellement ou `null` si l'utilisateur courant n'est pas authentifié (soit un simple visiteur). Le code suivant montre comment retrouver les autres informations relatives à l'authentification à partir de `yii\web\User` :

```
// l'identité de l'utilisateur courant. Null si l'utilisateur n'est pas
authentifié.
$identity = Yii::$app->user->identity;

// l'identifiant de l'utilisateur courant. Null si l'utilisateur n'est pas
authentifié.
$id = Yii::$app->user->id;

// si l'utilisateur courant est un visiteur (non authentifié).
$isGuest = Yii::$app->user->isGuest;
```

Pour connecter un utilisateur, vous devez utiliser le code suivant :

```
// trouve une identité d'utilisateur à partir du nom d'utilisateur spécifié
// notez que vous pouvez vouloir vérifier le mot de passe si besoin.
$identity = User::findOne(['username' => $username]);

// connecte l'utilisateur
Yii::$app->user->login($identity);
```

La méthode `yii\web\User::login()` assigne l'identité de l'utilisateur courant à `yii\web\User`. Si la session est `activée`, elle conserve l'identité de

façon à ce que l'état d'authentification de l'utilisateur soit maintenu durant la session tout entière. Si la connexion basée sur les témoins de connexion (*cookies*) est `activée`, elle sauvegarde également l'identité dans un témoin de connexion de façon à ce que l'état d'authentification de l'utilisateur puisse être récupéré du témoin de connexion durant toute la période de validité du témoin de connexion.

Pour activer la connexion basée sur les témoins de connexion, vous devez configurer `yii\web\User::$enableAutoLogin` à `true` (vrai) dans la configuration de l'application. Vous devez également fournir une durée de vie lorsque vous appelez la méthode `yii\web\User::login()`.

Pour déconnecter un utilisateur, appelez simplement

```
Yii::$app->user->logout();
```

Notez que déconnecter un utilisateur n'a de sens que si la session est activée. La méthode nettoie l'état d'authentification de l'utilisateur à la fois de la mémoire et de la session. Et, par défaut, elle détruit aussi *toutes* les données de session. Si vous voulez conserver les données de session, vous devez appeler `Yii::$app->user->logout(false)`, à la place.

9.1.4 Événement d'authentification

La classe `yii\web\User` lève quelques événements durant le processus de connexion et celui de déconnexion.

— `EVENT_BEFORE_LOGIN` : levé au début de `yii\web\User::login()`. Si le gestionnaire d'événement définit la propriété `isValid` de l'objet événement à `false` (faux), le processus de connexion avorte.
— `EVENT_AFTER_LOGIN` : levé après une connexion réussie.
— `EVENT_BEFORE_LOGOUT` : levé au début de `yii\web\User::logout()`. Si le gestionnaire d'événement définit la propriété `isValid` à `false` (faux) le processus de déconnexion avorte.
— `EVENT_AFTER_LOGOUT` : levé après une déconnexion réussie.

Vous pouvez répondre à ces événements pour mettre en œuvre des fonctionnalités telles que l'audit de connexion, les statistiques d'utilisateurs en ligne. Par exemple, dans le gestionnaire pour l'événement `EVENT_AFTER_LOGIN`, vous pouvez enregistrer le temps de connexion et l'adresse IP dans la tale `user`.

9.2 Autorisation

L'autorisation est le processus qui vérifie si un utilisateur dispose des permissions suffisantes pour faire quelque chose. Yii fournit deux méthodes d'autorisation : le filtre de contrôle d'accès (ACF — Access Control Filter) et le contrôle d'accès basé sur les rôles (RBAC — Role-Based Access Control).

9.2.1 Filtre de contrôle d'accès

Le filtre de contrôle d'accès (ACF) est une simple méthode d'autorisation mise en œuvre sous le nom `yii\filters\AccessControl` qui trouve son meilleur domaine d'application dans les applications qui n'ont besoin que d'un contrôle d'accès simplifié. Comme son nom l'indique, le filtre de contrôle d'accès est un filtre d'action qui peut être utilisé dans un contrôleur ou dans un module. Quand un utilisateur requiert l'exécution d'une action, le filtre de contrôle d'accès vérifie une liste de `règles d'accès` pour déterminer si l'utilisateur est autorisé à accéder à l'action requise.

Le code ci-dessous montre comment utiliser le filtre de contrôle d'accès dans le contrôleur `site` :

```
use yii\web\Controller;
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::class,
                'only' => ['login', 'logout', 'signup'],
                'rules' => [
                    [
                        'allow' => true,
                        'actions' => ['login', 'signup'],
                        'roles' => ['?'],
                    ],
                    [
                        'allow' => true,
                        'actions' => ['logout'],
                        'roles' => ['@'],
                    ],
                ],
            ],
        ];
    }
    // ...
}
```

Dans le code précédent, le filtre de contrôle d'accès est attaché au contrôleur `site` en tant que comportement (*behavior*). C'est la manière typique d'utiliser un filtre d'action. L'option `only` spécifie que le filtre de contrôle d'accès doit seulement être appliqué aux actions `login`, `logout` et `signup`. Toutes les autres actions dans le contrôleur `site`ne sont pas sujettes au contrôle d'accès. L'option `rules` liste les `règles d'accès`, qui se lisent comme suit :

— Autorise tous les visiteurs (non encore authentifiés) à accéder aux actions `login` et `signup`. l'option `roles` contient un point d'interrogation

`?` qui est un signe particulier représentant les « visiteurs non authentifiés ».
— Autorise les utilisateurs authentifiés à accéder à l'action `logout`. L'arobase `@` est un autre signe particulier représentant les « utilisateurs authentifiés ».

Le filtre de contrôle d'accès effectue les vérifications d'autorisation en examinant les règles d'accès une par une en commençant par le haut, jusqu'à ce qu'il trouve une règle qui correspond au contexte d'exécution courant. La valeur `allow` de la règle correspondante est utilisée ensuite pour juger si l'utilisateur est autorisé ou pas. Si aucune des règles ne correspond, cela signifie que l'utilisateur n'est PAS autorisé, et le filtre de contrôle d'accès arrête la suite de l'exécution de l'action.

Quand le filtre de contrôle d'accès détermine qu'un utilisateur n'est pas autorisé à accéder à l'action courante, par défaut, il prend les mesures suivantes :

— Si l'utilisateur est un simple visiteur, il appelle `yii\web\User::loginRequired()` pour rediriger le navigateur de l'utilisateur sur la page de connexion.
— Si l'utilisateur est déjà authentifié, il lève une exception `yii\web\ForbiddenHttpException`.

Vous pouvez personnaliser ce comportement en configurant la propriété `yii\filters\AccessControl::$denyCallback` comme indiqué ci-après :

```
[
    'class' => AccessControl::class,
    ...
    'denyCallback' => function ($rule, $action) {
        throw new \Exception('You are not allowed to access this page');
    }
]
```

Les `règles d'accès` acceptent beaucoup d'options. Ci-dessous, nous présentons un résumé des options acceptées. Vous pouvez aussi étendre la classe `yii\filters\AccessRule` pour créer vos propres classe de règles d'accès.

— `allow` : spécifie s'il s'agit d'une règle "allow" (autorise) ou "deny" (refuse).
— `actions` : spécifie à quelles actions cette règle correspond. Ce doit être un tableau d'identifiants d'action. La comparaison est sensible à la casse. Si cette option est vide ou non définie, cela signifie que la règle s'applique à toutes les actions.
— `controllers` : spécifie à quels contrôleurs cette règle correspond. Ce doit être un tableau d'identifiants de contrôleurs. Si cette option est vide ou non définie, la règle s'applique à tous les contrôleurs.
— `roles` : spécifie à quels rôles utilisateur cette règle correspond. Deux rôles spéciaux sont reconnus, et ils sont vérifiés via `yii\web\User::$isGuest` :
  — `?` : correspond à un visiteur non authentifié.
  — `@` : correspond à un visiteur authentifié.

L'utilisation d'autres noms de rôle déclenche l'appel de `yii\web\User::can()`, qui requiert l'activation du contrôle d'accès basé sur les rôles qui sera décrit dans la prochaine sous-section. Si cette option est vide ou non définie, cela signifie que la règle s'applique à tous les rôles.

— `ips` : spécifie à quelles `adresses IP de client` cette règle correspond. Une adresse IP peut contenir le caractère générique `*` à la fin pour indiquer que la règle correspond à des adresses IP ayant le même préfixe. Par exemple, '192.168.*' correspond à toutes les adresse IP dans le segment '192.168.'. Si cette option est vide ou non définie, cela signifie que la règle s'applique à toutes les adresses IP.

— `verbs` : spécifie à quelles méthodes de requête (p. ex. `GET`, `POST`) cette règle correspond. La comparaison est insensible à la casse.

— `matchCallback` : spécifie une fonction de rappel PHP qui peut être appelée pour déterminer si cette règle s'applique.

— `denyCallback` : spécifie une fonction de rappel PHP qui peut être appelée lorsqu'une règle refuse l'accès.

Ci-dessous nous présentons un exemple qui montre comment utiliser l'option `matchCallback`, qui vous permet d'écrire une logique d'accès arbitraire :

```php
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::class,
                'only' => ['special-callback'],
                'rules' => [
                    [
                        'actions' => ['special-callback'],
                        'allow' => true,
                        'matchCallback' => function ($rule, $action) {
                            return date('d-m') === '31-10';
                        }
                    ],
                ],
            ],
        ];
    }

    // Fonction de rappel appelée ! Cette page ne peut être accédée que
    chaque 31 octobre
    public function actionSpecialCallback()
    {
        return $this->render('happy-halloween');
    }
}
```

9.2.2 Contrôle d'accès basé sur les rôles

Le contrôle d'accès basé sur les rôles (Role-Based Access Control – RBAC) fournit un contrôle d'accès centralisé simple mais puissant. Reportez-vous à Wikipedia [1] pour des détails comparatifs entre le contrôle d'accès basé sur les rôles et d'autres schéma de contrôle d'accès plus traditionnels.

Yii met en œuvre un contrôle d'accès basé sur les rôles général hiérarchisé, qui suit le modèle NIST RBAC [2]. Il fournit la fonctionnalité de contrôle d'accès basé sur les rôles via le composant d'application`yii\RBAC\ManagerInterface`.

L'utilisation du contrôle d'accès basé sur les rôles implique deux partie de travail. La première partie est de construire les données d'autorisation du contrôle d'accès basé sur les rôles, et la seconde partie est d'utiliser les données d'autorisation pour effectuer les vérifications d'autorisation d'accès là où elles sont nécessaires.

Pour faciliter la description qui suit, nous allons d'abord introduire quelques concepts sur le contrôle d'accès basé sur les rôles.

Concepts de base

Un rôle représente une collection de *permissions* (p. ex. créer des articles, mettre des articles à jour). Un rôle peut être assigné à un ou plusieurs utilisateurs. Pour vérifier qu'un utilisateur dispose d'une permission spécifiée, nous pouvons vérifier si un rôle contenant cette permission a été assigné à l'utilisateur.

Associée à chacun des rôles, il peut y avoir une *règle*. Une règle représente un morceau de code à exécuter lors de l'accès pour vérifier si le rôle correspondant, ou la permission correspondante, s'applique à l'utilisateur courant. Par exemple, la permission « mettre un article à jour » peut disposer d'une règle qui vérifie si l'utilisateur courant est celui qui a créé l'article. Durant la vérification de l'accès, si l'utilisateur n'est PAS le créateur de l'article, il est considéré comme ne disposant pas la permission « mettre un article à jour ».

À la fois les rôles et les permissions peuvent être organisés en une hiérarchie. En particulier, un rôle peut être constitué d'autres rôles ou permissions ; Yii met en œuvre une hiérarchie *d'ordre partiel* qui inclut la hiérarchie plus spécifique dite *en arbre*. Tandis qu'un rôle peut contenir une permission, l'inverse n'est pas vrai.

Configuration du contrôle d'accès basé sur les rôles

Avant que nous ne nous lancions dans la définition des données d'autorisation et effectuions la vérification d'autorisation d'accès, nous devons confi-

1. https://fr.wikipedia.org/wiki/Contr%C3%B4le_d%27acc%C3%A8s_%C3%A0_base_de_r%C3%B4les
2. https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1992/10/13/role-based-access-controls/documents/ferraiolo-kuhn-92.pdf

gurer le composant d'application `gestionnaire d'autorisations (`. Yii fournit deux types de gestionnaires d'autorisations : `yii\rbac\PhpManager` et `yii\rbac\DbManager`. Le premier utilise un script PHP pour stocker les données d'autorisation, tandis que le second stocke les données d'autorisation dans une base de données. Vous pouvez envisager d'utiliser le premier si votre application n'a pas besoin d'une gestion des rôles et des permissions très dynamique.

Utilisation de Le code qui suit montre comment configurer la propriété `authManager` dans la configuration de l'application en utilisant la classe `yii \rbac\PhpManager` :

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
        // ...
    ],
];
```

Le gestionnaire `authManager` peut désormais être obtenu via `\Yii::$app->authManager`.

Par défaut, `yii\rbac\PhpManager` stocke les données du contrôle d'accès basé sur les rôles dans des fichiers du dossier `@app/rbac`. Assurez-vous que le dossier et tous les fichiers qui sont dedans sont accessibles en écriture par le processus du serveur Web si la hiérarchie des permissions a besoin d'être changée en ligne.

Utilisation de Le code qui suit monte comment configurer la propriété `authManager` dans la configuration de l'application en utilisant la classe `yii \rbac\DbManager` :

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\DbManager',
        ],
        // ...
    ],
];
```

> **Note :** si vous utilisez le modèle d'application yii2-basic-app, il y a un fichier de configuration `config/console.php` où la propriété `authManager` doit être également déclarée en plus de `config/web.php`.
>
> Dans le cas du modèle yii2-advanced-app, la propriété `authManager` doit être déclarée seulement une fois dans `common/config/main.php`.

`DbManager` utilise quatre tables de base de données pour stocker ses données :

— `itemTable` : la table pour stocker les items d'autorisation. Valeur par défaut « auth_item ».
— `itemChildTable` : la table pour stocker la hiérarchie des items d'autorisation. Valeur par défaut « auth_item_child ».
— `assignmentTable` : la table pour stocker les assignations d'items d'autorisation. Valeur par défaut « auth_assignment ».
— `ruleTable` : la table pour stocker les règles. Valeur par défaut « auth_rule ».

Avant de continuer vous devez créer ces tables dans la base de données. Pour le faire , vous pouvez utiliser la migration stockée dans `@yii/rbac/migrations` :

```
yii migrate --migrationPath=@yii/rbac/migrations
```

Le gestionnaire d'autorisations `authManager` peut désormais être obtenu par `\Yii::$app->authManager`.

Construction des données d'autorisation

Construire les donnés d'autorisation consiste à effectuer les tâches suivantes :

— définir les rôles et les permissions ;
— établir les relations entre les rôles et les permissions ;
— définir les règles ;
— associer les règles avec les rôles et les permissions ;
— assigner des rôles aux utilisateurs.

Selon les exigences de flexibilité des autorisations, les tâches énumérées ci-dessus peuvent être accomplies de différentes manières :

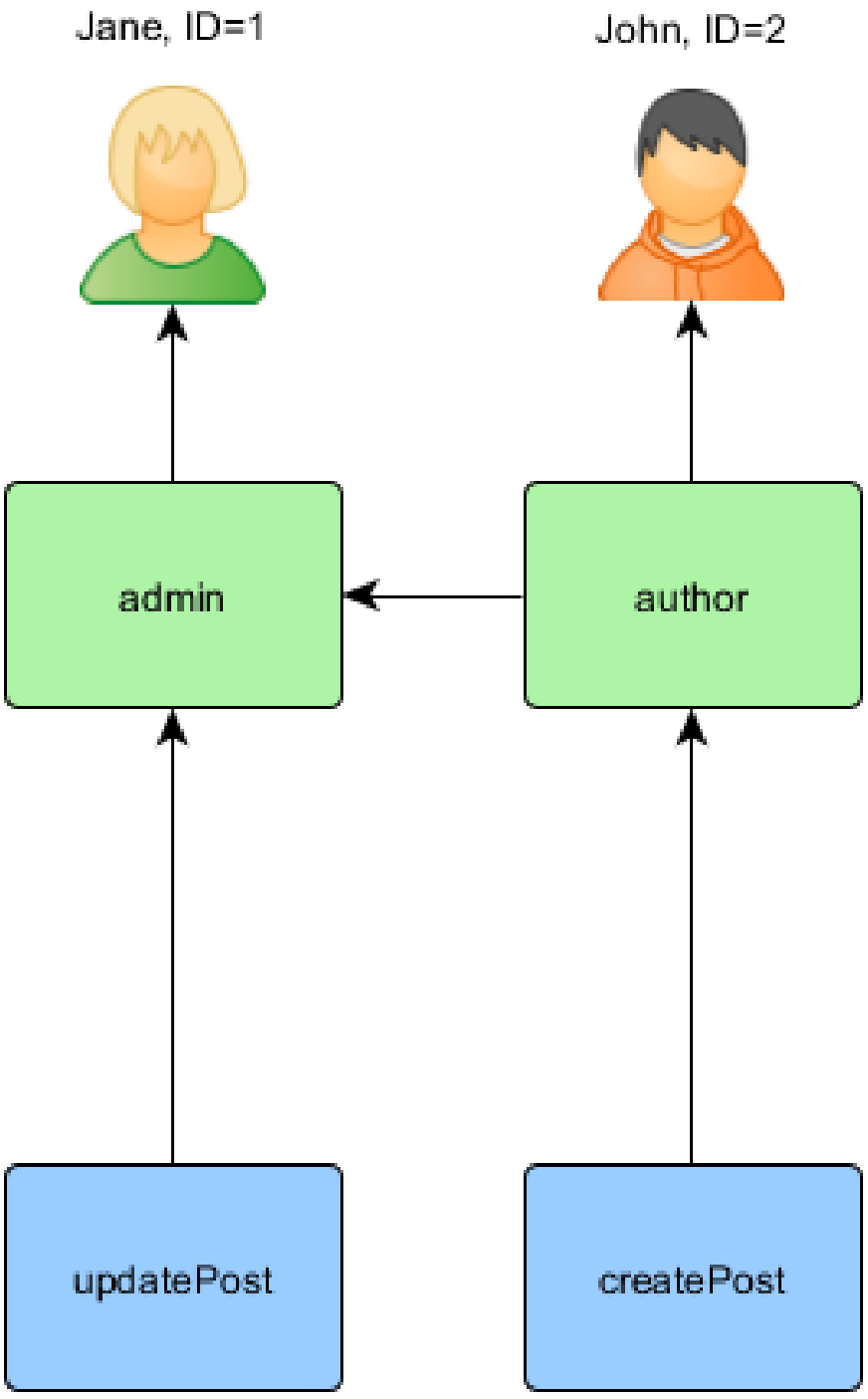
Si la hiérarchie de vos permissions ne change pas du tout et que vous avez un nombre fixé d'utilisateurs, vous pouvez créer une commande de console qui initialise les données d'autorisation une fois via l'API que fournit `authManager` :

```php
<?php
namespace app\commands;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
{
    public function actionInit()
    {
        $auth = Yii::$app->authManager;

        // ajoute une permission "createPost"
        $createPost = $auth->createPermission('createPost');
        $createPost->description = 'Créer un article';
        $auth->add($createPost);

        // ajoute une permission "updatePost"
```

```
        $updatePost = $auth->createPermission('updatePost');
        $updatePost->description = 'Mettre à jour un article';
        $auth->add($updatePost);

        // ajoute un rôle  "author" et donne à ce rôle la permission
        "createPost"
        $author = $auth->createRole('author');
        $auth->add($author);
        $auth->addChild($author, $createPost);

        // ajoute un rôle "admin" role et donne à ce rôle la permission
        "updatePost"
        // aussi bien que les permissions du rôle "author"
        $admin = $auth->createRole('admin');
        $auth->add($admin);
        $auth->addChild($admin, $updatePost);
        $auth->addChild($admin, $author);

        // Assigne des rôles aux utilisateurs. 1 et 2 sont des identifiants
        retournés par IdentityInterface::getId()
        // ordinairement mis en œuvre dans votre modèle  User .
        $auth->assign($author, 2);
        $auth->assign($admin, 1);
    }
}
```

Note : si vous utilisez le modèle avancé, vous devez mettre votre `RbacController` dans le dossier `console/controllers` et changer l'espace de noms en `console\controllers`.

Après avoir exécuté la commande `yii rbac/init` vous vous retrouverez avec la hiérarchie suivante :



Le rôle *Author* peut créer des articles, le rôle *admin* peut mettre les articles à jour et faire tout ce que le rôle *author* peut faire.

Si votre application autorise l'enregistrement des utilisateurs, vous devez assigner des rôles à ces nouveaux utilisateurs une fois. Par exemple, afin que tous les utilisateurs enregistrés deviennent des auteurs (rôle *author*) dans votre modèle de projet avancé, vous devez modifier la méthode `frontend\models\SignupForm::signup()` comme indiqué ci-dessous :

```
public function signup()
{
    if ($this->validate()) {
        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
        $user->setPassword($this->password);
        $user->generateAuthKey();
        $user->save(false);

        // Ces trois lignes ont été ajoutées
        $auth = Yii::$app->authManager;
        $authorRole = $auth->getRole('author');
        $auth->assign($authorRole, $user->getId());

        return $user;
    }

    return null;
}
```

Pour les applications qui requièrent un contrôle d'accès complexe avec des autorisations mises à jour dynamiquement, des interfaces utilisateur spéciales (c.-à-d. un panneau d'administration) doivent être développées en utilisant l'API offerte par `authManager`.

Utilisation des règles

Comme mentionné plus haut, les règles ajoutent des contraintes supplémentaires aux rôles et aux permissions. Une règle est une classe qui étend la classe `yii\rbac\Rule`. Elle doit implémenter la méthode `execute()`. Dans la hiérarchie, que nous avons créée précédemment le rôle *author* ne peut pas modifier ses propres articles. Essayons de régler ce problème. Tout d'abord, nous devons vérifier que l'utilisateur courant est l'auteur de l'article :

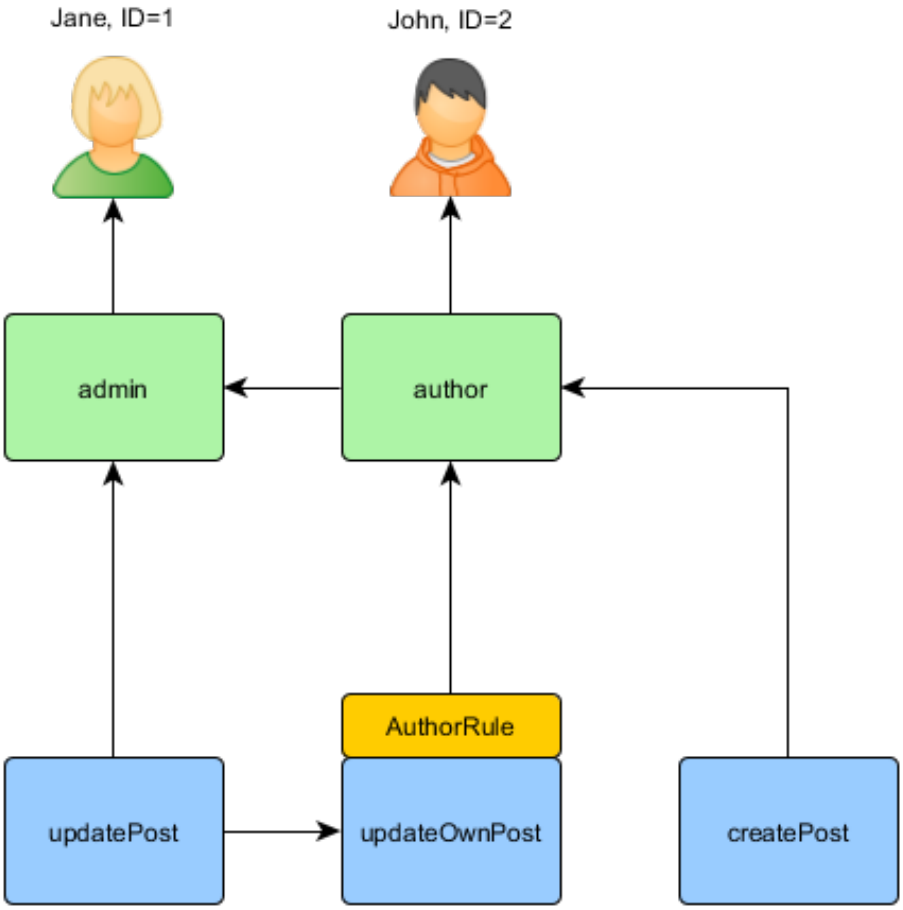
```
namespace app\rbac;

use yii\rbac\Rule;

/**
 * Vérifie si l'identifiant de l'auteur correspond à celui passé en
paramètre
 */
class AuthorRule extends Rule
```

```
{
    public $name = 'isAuthor';

    /**
     * @param string|int $user l'identifiant de l'utilisateur.
     * @param Item $item le rôle ou la permission avec laquelle cette règle
est associée
     * @param array $params les paramètres passés à
ManagerInterface::checkAccess().
     * @return bool une valeur indiquant si la règles autorise le rôle ou la
permission qui lui est associé.
     */
    public function execute($user, $item, $params)
    {
        return isset($params['post']) ? $params['post']->createdBy == $user
        : false;
    }
}
```

La règles ci-dessus vérifie si l'article `post` a été créé par l'utilisateur `$user`. Nous allons créer une permission spéciale `updateOwnPost` dans la commande que nous avons utilisée précédemment :

```
$auth = Yii::$app->authManager;

// ajoute la règle
$rule = new \app\rbac\AuthorRule;
$auth->add($rule);

// ajoute la permission "updateOwnPost" et associe lui la règle
$updateOwnPost = $auth->createPermission('updateOwnPost');
$updateOwnPost->description = 'Mettre à jour un des ses propres articles';
$updateOwnPost->ruleName = $rule->name;
$auth->add($updateOwnPost);

// "updateOwnPost" sera utilisé depuis  "updatePost"
$auth->addChild($updateOwnPost, $updatePost);

// autorise les utilisateurs ayant le rôle  "author" à mettre à jour leurs
propres articles.
$auth->addChild($author, $updateOwnPost);
```

Nous nous retrouvons avec la hiérarchie suivante :

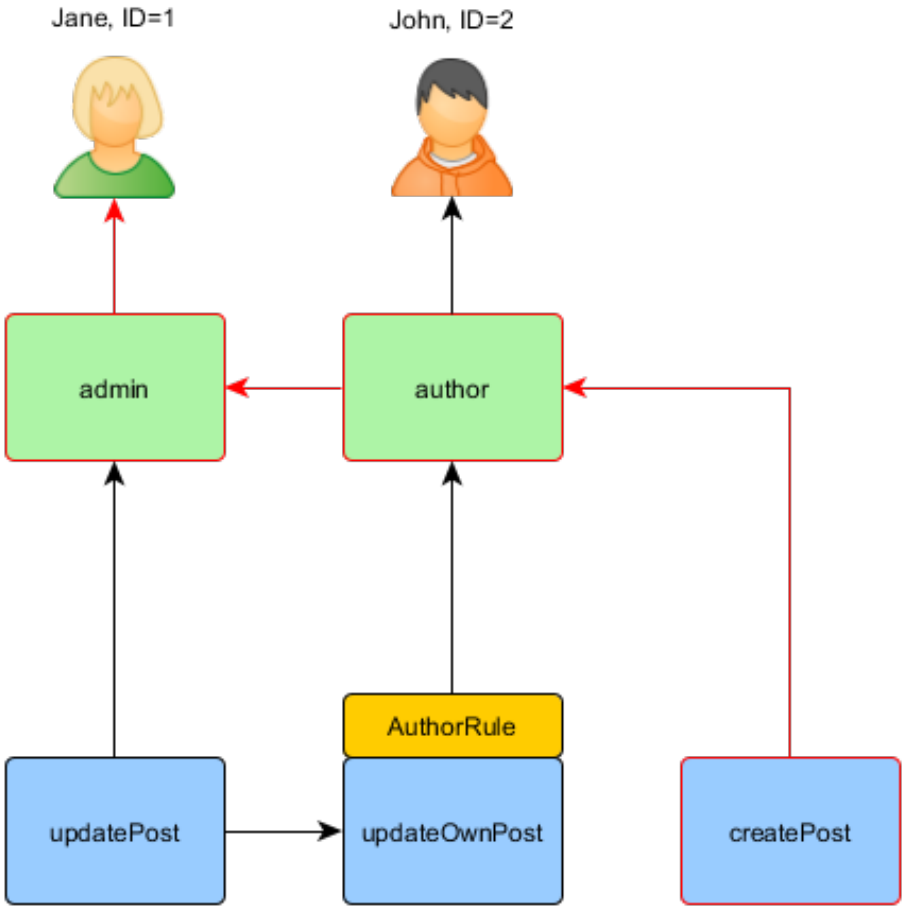


Vérification de l'autorisation d'accès

Avec les données d'autorisation préparées, la vérification de l'autorisation d'accès est aussi simple que d'appeler la méthode `yii\rbac\ManagerInterface::checkAccess()`. Étant donné que la plupart des vérification d'autorisation d'accès concernent l'utilisateur courant, pour commodité, Yii procure une méthode raccourcie `yii\web\User::can()`, qui peut être utilisée comme suit :

```
if (\Yii::$app->user->can('createPost')) {
    // create post
}
```

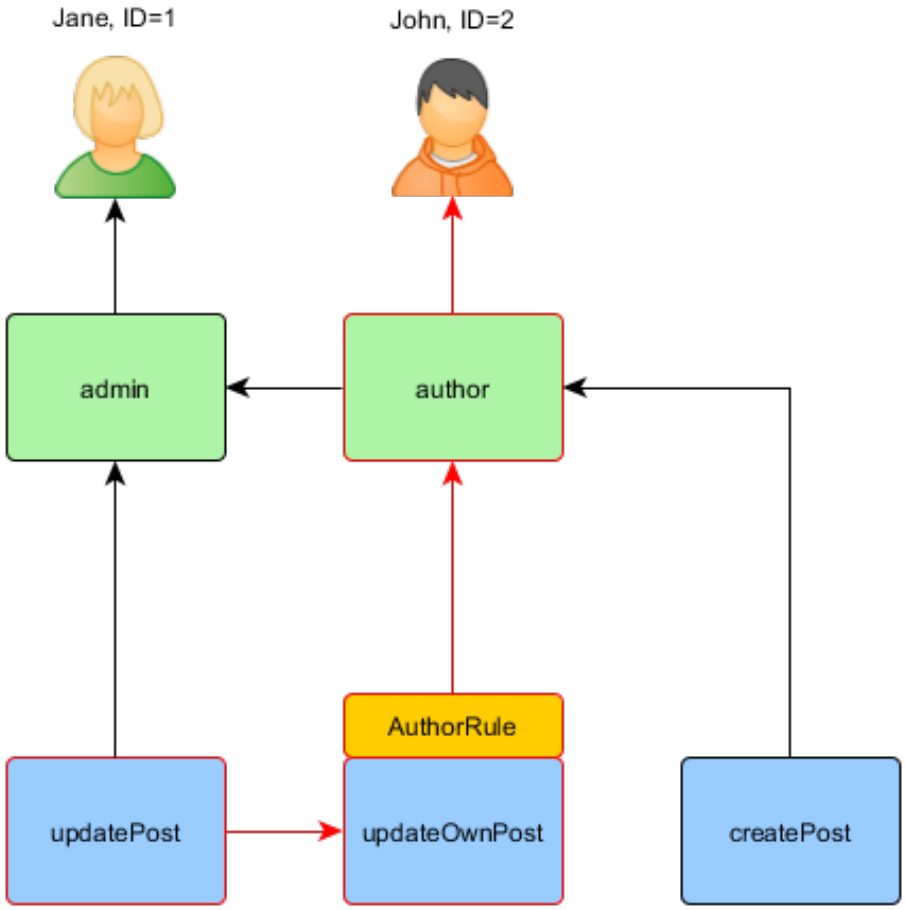
Si l'utilisateur courant est Jane avec l'identifiant `ID=1`, nous commençons à `createPost` et essayons d'arriver à `Jane` :



Afin de vérifier sur un utilisateur peut mettre un article à jour, nous devons passer un paramètre supplémentaire qui est requis par la règle `AuthorRule` décrite précédemment :

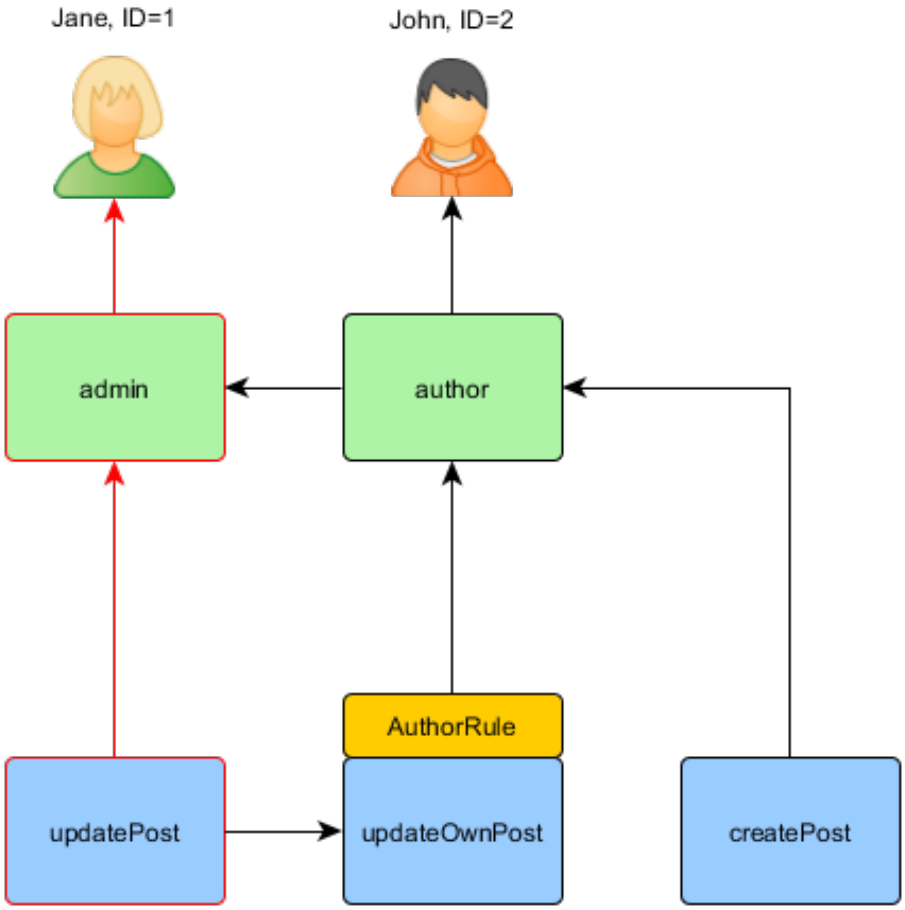
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {
    // met à jour l'article
}
```

Ici que se passe-t-il si l'utilisateur courant est John :



Nous commençons à `updatePost` et passons par `updateOwnPost`. Afin d'obtenir l'autorisation, la méthode `execute()` de `AuthorRule` doit retourner `true` (vrai). La méthode reçoit ses paramètres `$params` de l'appel à la méthode `can()` et sa valeur est ainsi `['post' => $post]`. Si tout est bon, nous arrivons à `author` auquel John est assigné.

Dans le cas de Jane, c'est un peu plus simple puisqu'elle a le rôle admin :



Dans votre contrôleur, il y a quelques façons de mettre en œuvre les autorisations. Si vous voulez des permission granulaires qui séparent l'accès entre ajouter et supprimer, alors vous devez vérifier l'accès pour chacune des actions. Vous pouvez soit utiliser la condition ci-dessus dans chacune des méthodes d'action, ou utiliser `yii\filters\AccessControl` :

```
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::class,
            'rules' => [
                [
                    'allow' => true,
                    'actions' => ['index'],
                    'roles' => ['managePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['view'],
```

```
                    'roles' => ['viewPost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['create'],
                    'roles' => ['createPost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['update'],
                    'roles' => ['updatePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['delete'],
                    'roles' => ['deletePost'],
                ],
            ],
        ],
    ];
}
```

Si toutes les opérations CRUD sont gérées ensemble, alors c'est une bonne idée que d'utiliser une permission unique comme `managePost` (gérer article), et de la vérifier dans `yii\web\Controller::beforeAction()`.

Utilisation des rôles par défaut

Un rôle par défaut est un rôle qui est assigné *implicitement* à tous les *utilisateurs*. L'appel de la méthode `yii\rbac\ManagerInterface::assign()` n'est pas nécessaire, et les données d'autorisations ne contiennent pas ses informations d'assignation.

Un rôle par défaut est ordinairement associé à une règle qui détermine si le rôle s'applique à l'utilisateur en cours de vérification.

Les rôles par défaut sont souvent utilisés dans des applications qui ont déjà une sorte d'assignation de rôles. Par exemple, un application peut avoir une colonne « group » dans sa table des utilisateurs pour représenter à quel groupe de privilèges chacun des utilisateurs appartient. Si chaque groupe de privilèges peut être mis en correspondance avec un rôle du contrôle d'accès basé sur les rôles, vous pouvez utiliser la fonctionnalité de rôle par défaut pour assigner automatiquement un rôle du contrôle d'accès basé sur les rôles à chacun des utilisateurs. Prenons un exemple pour montrer comment cela se fait.

Supposons que dans la table des utilisateurs, il existe en colonne `group` qui utilise la valeur 1 pour représenter le groupe des administrateurs et la valeur 2 pour représenter le groupe des auteurs. Vous envisagez d'avoir deux rôles dans le contrôle d'accès basé sur les rôles `admin` et`author` pour représenter les

permissions de ces deux groupes respectivement. Vous pouvez configurer le contrôle d'accès basé sur les rôles comme suit :

```
namespace app\rbac;

use Yii;
use yii\rbac\Rule;

/**
 * Vérifie si le groupe utilisateurs correspond
 */
class UserGroupRule extends Rule
{
    public $name = 'userGroup';

    public function execute($user, $item, $params)
    {
        if (!Yii::$app->user->isGuest) {
            $group = Yii::$app->user->identity->group;
            if ($item->name === 'admin') {
                return $group == 1;
            } elseif ($item->name === 'author') {
                return $group == 1 || $group == 2;
            }
        }
        return false;
    }
}

$auth = Yii::$app->authManager;

$rule = new \app\rbac\UserGroupRule;
$auth->add($rule);

$author = $auth->createRole('author');
$author->ruleName = $rule->name;
$auth->add($author);
// ... ajoute les  permissions en tant qu'enfant de  $author ...

$admin = $auth->createRole('admin');
$admin->ruleName = $rule->name;
$auth->add($admin);
$auth->addChild($admin, $author);
// ... ajoute les  permissions en tant qu'enfant de  $admin ...
```

Notez que dans ce qui est présenté ci-dessus, comme « author » est ajouté en tant qu'enfant de « admin », lorsque vous implémentez la méthode `execute()` de la classe de règle, vous devez respecter cette hiérarchie elle aussi. C'est pourquoi, lorsque le nom de rôle est « author », la méthode `execute()` retourne `true` (vrai) si le groupe de l'utilisateur est soit 1, soit 2 (ce qui signifie que l'utilisateur est soit dans le groupe « admin », soit dans le groupe « author »).

Ensuite, configurez `authManager` en listant les deux rôles dans `yii\rbac`

`\BaseManager::$defaultRoles` :

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
            'defaultRoles' => ['admin', 'author'],
        ],
        // ...
    ],
];
```

Désormais, si vous effectuez une vérification d'autorisation d'accès, les deux rôles `admin` et `author` seront vérifiés en évaluant les règles qui leur sont associées. Si les règles retournent `true` (vrai), cela signifie que le rôle s'applique à l'utilisateur courant. En se basant sur la mise en œuvre des règles ci-dessus, cela signifie que si la valeur du `group` d'un utilisateur est 1, le rôle `admin` s'applique à l'utilisateur, si la valeur du `group` est 2, le rôle `author` s'applique.

9.3 Utilisation de mots de passe

La plupart des développeurs savent que les mots de passe ne peuvent pas être stockés « en clair », mais beaucoup d'entre-eux croient qu'il est toujours sûr des les hacher avec `md5` ou `sha1`. Il fut un temps où utiliser ces algorithmes de hachage était suffisant, mais les matériels modernes font qu'il est désormais possible de casser de tels hachages – même les plus robustes – très rapidement en utilisant des attaques en force brute.

Pour apporter une sécurité améliorée pour les mots de passe des utilisateurs, même dans le pire des scénario (une brèche est ouverte dans votre application), vous devez utiliser des algorithmes de hachage qui résistent aux attaques en force brute. Le choix le meilleur couramment utilisé est `bcrypt`.

En PHP, vous pouvez créer une valeur de hachage `bcrypt` à l'aide de la fonction crypt [3]. Yii fournit deux fonctions d'aide qui facilitent l'utilisation de `crypt` pour générer et vérifier des valeurs de hachage de manière sure.

Quand un utilisateur fournit un mot de passe pour la première fois (p. ex. à l'enregistrement), le mot de passe doit être haché :

```
$hash = Yii::$app->getSecurity()->generatePasswordHash($password);
```

La valeur de hachage peut ensuite être associée à l'attribut du modèle correspondant afin de pouvoir être stockée dans la base de données pour utilisation ultérieure.

Lorsqu'un utilisateur essaye ensuite de se connecter, le mot de passe soumis est comparé au mot de passe précédemment haché et stocké :

3. https://www.php.net/manual/fr/function.crypt.php

```php
if (Yii::$app->getSecurity()->validatePassword($password, $hash)) {
    // tout va bien, nous connectons l'utilisateur
} else {
    // wrong password
}
```

Error : not existing file : security-auth-clients.md

9.4 Meilleures pratiques de sécurité

Ci-dessous, nous passons en revue les principes de sécurité courants et décrivons comment éviter les menaces lorsque vous développez une application Yii.

9.4.1 Principes de base

Il y a deux principes essentiels quand on en vient à traiter de la sécurité des applications quelles qu'elles soient :

1. Filtrer les entrées.
2. Échapper les sorties.

Filtrer les entrées

Filtrer les entrées signifie que les entrées ne doivent jamais être considérées comme sures et que vous devriez toujours vérifier qu'une valeur que vous avez obtenue fait réellement partie de celles qui sont autorisées. Par exemple, si nous savons qu'un tri doit être fait sur la base de trois champs `title`, `created_at` et `status`, et que ces champs sont fournis sous forme d'entrées de l'utilisateur, il vaut mieux vérifier les valeurs exactement là où nous les recevons. En terme de PHP de base, ça devrait ressembler à ceci :

```
$sortBy = $_GET['sort'];
if (!in_array($sortBy, ['title', 'created_at', 'status'])) {
        throw new Exception('Invalid sort value.');
}
```

Dans Yii, le plus probablement, vous utilisez la validation de formulaire pour faire de telles vérifications.

Échapper les sorties

Échapper les sorties signifie que, selon le contexte dans lequel vous utilisez les données, elles doivent être échappées c.-à-d. dans le contexte de HTML vous devez échapper les caractères <, > et autres caractères similaires. Dans le contexte de JavaScript ou de SQL, il s'agira d'un jeu différent de caractères. Comme échapper tout à la main serait propice aux erreurs, Yii fournit des outils variés pour effectuer l'échappement dans différents contextes.

9.4.2 Éviter les injections SQL

Les injections SQL se produisent lorsque le texte d'une requête est formé en concaténant des chaînes non échappées comme la suivante :

```
$username = $_GET['username'];
$sql = "SELECT * FROM user WHERE username = '$username'";
```

Au lieu de fournir un nom d'utilisateur réel, l'attaquant pourrait donner à votre application quelque chose comme `'; DROP TABLE user; --`. Ce qui aboutirait à la requête SQL suivante :

```
SELECT * FROM user WHERE username = ''; DROP TABLE user; --'
```

Cela est une requête tout à fait valide qui recherche les utilisateurs avec un nom vide et détruit probablement la table `user`, ce qui conduit à un site Web cassé et à une perte de données (vous faites des sauvegardes régulières, pas vrai ?).

Dans Yii la plupart des requêtes de base de données se produisent via la classe Active Record qui utilise correctement des instructions PDO préparées en interne. En cas d'instructions préparées, il n'est pas possible de manipuler la requête comme nous le montrons ci-dessus.

Cependant, parfois, vous avez encore besoin de requêtes brutes ou du constructeur de requêtes. Dans ce cas, vous devriez passer les données de manière sure. Si les données sont utilisées pour des valeurs de colonnes, il est préférable d'utiliser des instructions préparées :

```
// query builder
$userIDs = (new Query())
    ->select('id')
    ->from('user')
    ->where('status=:status', [':status' => $status])
    ->all();

// DAO
$userIDs = $connection
    ->createCommand('SELECT id FROM user where status=:status')
    ->bindValues([':status' => $status])
    ->queryColumn();
```

Si les données sont utilisées pour spécifier des noms de colonne ou des noms de table, la meilleure chose à faire est d'autoriser uniquement des jeux prédéfinis de valeurs :

```
function actionList($orderBy = null)
{
    if (!in_array($orderBy, ['name', 'status'])) {
        throw new BadRequestHttpException('Only name and status are allowed
        to order by.')
    }

    // ...
}
```

Dans le cas où cela n'est pas possible, les noms de colonne et de table doivent être échappés. Yii a recours à une syntaxe spéciale pour un tel échappement qui permet de le faire d'une manière identique pour toutes les bases de données prises en charge :

```
$sql = "SELECT COUNT([[$column]]) FROM {{table}}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

Vous pouvez obtenir tous les détails sur cette syntaxe dans la section Échappement des noms de colonne et de table.

9.4.3 Éviter le XSS

Le XSS ou scriptage inter site se produit lorsque la sortie n'est pas échappée correctement lors de l'envoi de code HTML au navigateur. Par exemple, si l'utilisateur peut entrer son nom, et qu'au lieu de saisir `Alexander` il saisit `<script>alert('Hello!');</script>`, chaque page qui émet sont nom sans échappement exécute le code JavaScript `alert('Hello!');` ce qui se traduit par une boîte d'alerte jaillissant dans le navigateur. Selon le site web, au lieu de quelque chose d'aussi innocent, le script pourrait envoyer des messages en votre nom ou même effectuer des transactions bancaires. L'évitement de XSS est tout à fait facile. Il y a en général deux cas :

1. Vous voulez que vos données soient transmises sous forme de texte simple.
2. Vous voulez que vos données soient transmises sous forme de code HTML.

Si vous désirez seulement du texte simple, l'échappement est aussi simple à réaliser que ce qui suit :

```
<?= \yii\helpers\Html::encode($username) ?>
```

Si ce doit être du code HTML vous pouvez obtenir de l'aide de HtmlPurifier :

```
<?= \yii\helpers\HtmlPurifier::process($description) ?>
```

Notez que le processus de HtmlPurifier est très lourd, c'est pourquoi vous devez envisager la mise en cache.

9.4.4 Éviter le CSRF

La CSRF est une abréviation de cross-site request forgery (falsification de requête inter sites). L'idée est que beaucoup d'applications partent du principe que les requêtes provenant d'un navigateur sont fabriquées par l'utilisateur lui-même. Cela peut être faux.

Par exemple, un site web `an.example.com` a une URL `/logout`, qui, lorsqu'elle est accédée en utilisant une simple requête GET, déconnecte l'utilisateur. Tant qu'il s'agit d'une requête de l'utilisateur lui-même, tout va bien. Mais, un jour, des gens mal intentionnés, postent `<img src="https://an.example.com/logout">` sur un forum que l'utilisateur visite fréquemment. Le navigateur ne fait pas de différence entre la requête d'une image et celle d'une page. C'est pourquoi,

lorsque l'utilisateur ouvre une page avec une telle balise `img`, le navigateur envoie la requête GET vers cette URL, et l'utilisateur est déconnecté du site `an.example.com`.

C'est l'idée de base. D'aucuns diront que déconnecter un utilisateur n'a rien de très sérieux, mais les gens mal intentionnés peuvent faire bien plus, à partir de cette idée. Imaginez qu'un site web possède une URL `https://an.example.com/purse/transfer?to=anotherUser&amount=2000`. Accéder à cette URL en utilisant une requête GET, provoque le transfert de 2000 EUR d'un compte autorisé à l'utilisateur vers un autre compte `anotherUser`. Nous savons que le navigateur envoie toujours une requête GET pour charger une image. Nous pouvons donc modifier le code pour que seules des requêtes POST soient acceptées sur cette URL. Malheureusement, cela ne nous est pas d'un grand secours parce qu'un attaquant peut placer un peu le JavaScript à la place de la balise `<img>`, ce qui permet d'envoyer des requêtes POST sur cette URL :

Afin d'éviter la falsification des requêtes inter-sites vous devez toujours :

1. Suivre la spécification HTTP c.-à-d. GET ne doit pas changer l'état de l'application.
2. Tenir la protection Yii CSRF activée.

Parfois vous avez besoin de désactiver la validation CSRF pour un contrôleur ou une action. Cela peut être fait en définissant sa propriété :

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $enableCsrfValidation = false;

    public function actionIndex()
    {
        // la validation CSRF ne sera pas appliquée à cette action ainsi
        qu'aux autres.
    }

}
```

Pour désactiver la validation CSRF pour des actions personnalisées vous pouvez faire :

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
```

```
    public function beforeAction($action)
    {
        // ...définit `$this->enableCsrfValidation` ici en se basant sur
        quelques conditions...
        // appelle la méthode  du parent qui vérifie CSRF si une telle
        propriété est vraie
        return parent::beforeAction($action);
    }
}
```

9.4.5 Éviter l'exposition de fichiers

Par défaut, le racine du serveur web est censé pointer sur le dossier `web`, là où se trouve le fichier `index.php`. Dans le cas d'un hébergement partagé, il peut être impossible de réaliser cela et vous pouvez vous retrouver avec tout le code, configurations et journaux sous la racine du serveur web.

Si c'est le cas, n'oubliez-pas de refuser l'accès à tout sauf au dossier `web`. Si cela n'est pas possible, envisagez d'héberger votre application ailleurs.

9.4.6 Éviter les informations et des outils de débogage en mode production

En mode débogage, Yii présente les erreurs de façon très verbeuse, ce qui s'avère très utile en développement. Le problème est que des erreurs aussi verbeuses sont pleines de renseignement pour l'attaquant lui aussi et peuvent révéler la structure de la base de données, les valeurs de configuration et des parties de votre code. Ne faites jamais tourner vos applications avec `YII_DEBUG` définit à `true` dans votre fichier `index.php`.

Vous ne devriez jamais activer Gii en production. Il pourrait être utilisé pour obtenir des informations sur la structure de la base de données, sur le code et tout simplement réécrire du code avec celui généré par Gii.

La barre de débogage devrait être neutralisée en production sauf si vous en avez réellement besoin. Elle expose toute l'application et les détails de configuration. Si vous avez absolument besoin de cette barre, vérifier que cet accès est correctement réservé à votre adresse IP seulement.

9.4.7 Utilisation de connexions sécurisées via TLS

Yii fournit des fonctionnalités qui comptent sur les témoins de connexion et/ou sur les sessions PHP. Cela peut créer des vulnérabilités dans le cas où votre connexion est compromise. Le risque est réduit si l'application utilise une connexion sécurisée via TLS. Reportez-vous à la documentation de votre serveur web pour des instructions sur la manière de la configurer. Vous pouvez aussi jeter un coup d'œil aux exemples de configuration du projet H5BP :

— Nginx [4]
— Apache [5].
— IIS [6].
— Lighttpd [7].

4. https://github.com/h5bp/server-configs-nginx
5. https://github.com/h5bp/server-configs-apache
6. https://github.com/h5bp/server-configs-iis
7. https://github.com/h5bp/server-configs-lighttpd

Chapitre 10

Cache

10.1 Mise en cache

La mise en cache est un moyen peu coûteux et efficace d'améliorer la performance d'une application Web. En stockant des données relativement statiques en cache et en les servant à partir de ce cache lorsqu'elles sont demandées, l'application économise le temps qu'il aurait fallu pour générer ces données à partir de rien à chaque demande.

La mise en cache se produit à différents endroits et à différents niveaux dans une application Web. Du côté du serveur, au niveau le plus bas, le cache peut être utilisé pour stocker des données de base, telles qu'une liste des informations sur des articles recherchée dans une base de données ; et à un niveau plus élevé, il peut être utilisé pour stocker des fragments ou l'intégralité de pages Web, telles que le rendu des articles les plus récents.

Du côté client, la mise en cache HTTP peut être utilisée pour conserver le contenu des pages visitées les plus récentes dans le cache du navigateur.

Yii prend en charge tous ces mécanismes de mise en cache :

— Mise en cache de données
— Mise en cache de fragments
— Mise en cache de pages
— Mise en cache HTTP

10.2 Mise en cache de données

La mise en cache de données consiste à stocker quelques variables PHP dans un cache et à les y retrouver plus tard. C'est également la base pour des fonctionnalités de mise en cache plus avancées, comme la mise en cache de requêtes et la mise en cache de pages.

Le code qui suit est un modèle d'utilisation typique de la mise en cache de données, dans lequel `cache` fait référence à un composant de mise en cache :

```php
// tente de retrouver la donnée $data dans le cache
$data = $cache->get($key);

if ($data === false) {
    // la donnée $data n'a pas été trouvée dans le cache, on la recalcule
    $data = $this->calculateSomething();

    // stocke la donnée $data dans le cache de façon à la retrouver la
    prochaine fois
    $cache->set($key, $data);
}

// la donnée $data est disponible ici
```

Depuis la version 2.0.11, le composant de mise en cache fournit la méthode `getOrSet()` qui simplifie le code pour l'obtention, le calcul et le stockage des données. Le code qui suit fait exactement la même chose que l'exemple précédent :

```php
$data = $cache->getOrSet($key, function () {
    return $this->calculateSomething();
});
```

Lorsque le cache possède une donnée associée à la clé `$key`, la valeur en cache est retournée. Autrement, la fonction anonyme passée est exécutée pour calculer cette valeur qui est mise en cache et retournée.

Si la fonction anonyme a besoin de quelques données en dehors de la portée courante, vous pouvez les passer en utilisant l'instruction `use`. Par exemple :

```php
$user_id = 42;
$data = $cache->getOrSet($key, function () use ($user_id) {
    return $this->calculateSomething($user_id);
});
```

> Note : la méthode `getOrSet()` prend également en charge la durée et les dépendances. Reportez-vous à Expiration de la mise en cache et à Dépendances de mise en cache pour en savoir plus.

10.2.1 Composants de mise en cache

La mise en cache s'appuie sur ce qu'on appelle les *composants de mise en cache* qui représentent des supports de mise en cache tels que les mémoires, les fichiers et les bases de données.

Les composants de mise en cache sont généralement enregistrés en tant que composants d'application de façon à ce qu'ils puissent être configurables et accessibles globalement. Le code qui suit montre comment configurer le composant d'application `cache` pour qu'il utilise memcached [1] avec deux serveurs de cache :

1. https://memcached.org/

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
        'servers' => [
            [
                'host' => 'server1',
                'port' => 11211,
                'weight' => 100,
            ],
            [
                'host' => 'server2',
                'port' => 11211,
                'weight' => 50,
            ],
        ],
    ],
],
```

Vous pouvez accéder au composant de mise en cache configuré ci-dessus en utilisant l'expression `Yii::$app->cache`.

Comme tous les composants de mise en cache prennent en charge le même jeux d'API, vous pouvez remplacer le composant de mise en cache sous-jacent par un autre en le reconfigurant dans la configuration de l'application, cela sans modifier le code qui utilise le cache. Par exemple, vous pouvez modifier le code ci-dessus pour utiliser `APC cache` :

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
],
```

> **Conseil :** vous pouvez enregistrer de multiples composants d'application de mise en cache. Le composant nommé `cache` est utilisé par défaut par de nombreuses classes dépendantes d'un cache (p. ex.`yii\web\UrlManager`).

Supports de stockage pour cache pris en charge

Yii prend en charge un large panel de supports de stockage pour cache. Ce qui suit est un résumé :

— `yii\caching\ApcCache` : utilise l'extension PHP APC[2]. Cette option peut être considérée comme la plus rapide lorsqu'on utilise un cache pour une grosse application centralisée (p. ex. un serveur, pas d'équilibrage de charge dédié, etc.).

— `yii\caching\DbCache` : utilise une table de base de données pour stocker les données en cache. Pour utiliser ce cache, vous devez créer une table comme spécifié dans `yii\caching\DbCache::$cacheTable`.

2. https://www.php.net/manual/fr/book.apcu.php

— `yii\caching\DummyCache` : tient lieu de cache à remplacer qui n'assure pas de mise en cache réelle. Le but de ce composant est de simplifier le code qui a besoin de vérifier la disponibilité du cache. Par exemple, lors du développement ou si le serveur ne dispose pas de la prise en charge d'un cache, vous pouvez configurer un composant de mise en cache pour qu'il utilise ce cache. Lorsque la prise en charge réelle de la mise en cache est activée, vous pouvez basculer sur le composant de mise en cache correspondant. Dans les deux cas, vous pouvez utiliser le même code `Yii::$app->cache->get($key)` pour essayer de retrouver les données du cache sans vous préoccuper du fait que `Yii::$app->cache` puisse être `null`.
— `yii\caching\FileCache` : utilise des fichiers standards pour stocker les données en cache. Cela est particulièrement adapté à la mise en cache de gros blocs de données, comme le contenu d'une page.
— `yii\caching\MemCache` : utilise le memcache [3] PHP et l'extension memcached [4]. Cette option peut être considérée comme la plus rapide lorsqu'on utilise un cache dans des applications distribuées (p. ex. avec plusieurs serveurs, l'équilibrage de charge, etc.).
— `yii\redis\Cache` : met en œuvre un composant de mise en cache basé sur un stockage clé-valeur Redis [5] (une version de redis égale ou supérieure à 2.6.12 est nécessaire).
— `yii\caching\WinCache` : utilise le WinCache [6] PHP (voir aussi l'extension [7]).
— `yii\caching\XCache` *(deprecated)* : utilise l'extension PHP XCache [8].
— `yii\caching\ZendDataCache` *(deprecated)* : utilise le cache de données Zend [9] en tant que médium de cache sous-jacent.

> **Conseil :** vous pouvez utiliser différents supports de stockage pour cache dans la même application. Une stratégie courante est d'utiliser un support de stockage pour cache basé sur la mémoire pour stocker des données de petite taille mais d'usage constant (p. ex. des données statistiques), et d'utiliser des supports de stockage pour cache basés sur des fichiers ou des bases de données pour stocker des données volumineuses et utilisées moins souvent (p. ex. des contenus de pages).

3. https://www.php.net/manual/fr/book.memcache.php
4. https://www.php.net/manual/fr/book.memcached.php
5. https://redis.io/
6. https://iis.net/downloads/microsoft/wincache-extension
7. https://www.php.net/manual/fr/book.wincache.php
8. https://en.wikipedia.org/wiki/List_of_PHP_accelerators#XCache
9. https://files.zend.com/help/Zend-Server-6/zend-server.htm#data_cache_component.htm

10.2.2 Les API Cache

Tous les composants de mise en cache dérivent de la même classe de base `yii\caching\Cache` et par conséquent prennent en charge les API suivantes :

- `get()` : retrouve une donnée dans le cache identifiée par une clé spécifiée. Une valeur `false` (faux) est retournée si la donnée n'est pas trouvée dans le cache ou si elle a expiré ou été invalidée.
- `set()` : stocke une donnée sous une clé dans le cache.
- `add()` : stocke une donnée identifiée par une clé dans le cache si la clé n'existe pas déjà dans le cache.
- `getOrSet()` : retrouve une donnée dans le cache identifiée par une clé spécifiée ou exécute la fonction de rappel passée, stocke la valeur retournée par cette fonction dans le cache sous cette clé et retourne la donnée.
- `multiGet()` : retrouve de multiples données dans le cache identifiées par les clés spécifiées.
- `multiSet()` : stocke de multiples données dans le cache. Chaque donnée est identifiée par une clé.
- `multiAdd()` : stocke de multiples données dans le cache. Chaque donnée est identifiée par une clé. Si la clé existe déjà dans le cache, la donnée est ignorée.
- `exists()` : retourne une valeur indiquant si la clé spécifiée existe dans le cache.
- `delete()` : retire du cache une donnée identifiée par une clé.
- `flush()` : retire toutes les données du cache.

> Note : ne mettez pas directement en cache une valeur booléenne `false` parce que la méthode `get()` utilise la valeur `false` pour indiquer que la donnée n'a pas été trouvée dans le cache. Au lieu de cela, vous pouvez placer cette donnée dans un tableau et mettre ce tableau en cache pour éviter le problème.

Quelques supports de cache, tels que MemCache, APC, prennent en charge la récupération de multiples valeurs en cache en mode « batch » (lot), ce qui réduit la surcharge occasionnée par la récupération des données en cache. Les API `multiGet()` et `multiAdd()` sont fournies pour exploiter cette fonctionnalité. Dans le cas où le support de cache sous-jacent ne prend pas en charge cette fonctionnalité, elle est simulée. Comme `yii\caching\Cache` implémente `ArrayAccess`, un composant de mise en cache peut être utilisé comme un tableau. En voici quelques exemples :

```
$cache['var1'] = $value1;  // équivalent à : $cache->set('var1', $value1);
$value2 = $cache['var2'];  // équivalent à : $value2 = $cache->get('var2');
```

Clés de cache

Chacune des données stockée dans le cache est identifiée de manière unique par une clé. Lorsque vous stockez une donnée dans le cache, vous devez spécifier une clé qui lui est attribuée. Plus tard, pour récupérer la donnée, vous devez fournir cette clé.

Vous pouvez utiliser une chaîne de caractères ou une valeur arbitraire en tant que clé de cache. Lorsqu'il ne s'agit pas d'une chaîne de caractères, elle est automatiquement sérialisée sous forme de chaîne de caractères.

Une stratégie courante pour définir une clé de cache consiste à inclure tous les facteurs déterminants sous forme de tableau. Par exemple,`yii\db\Schema` utilise la clé suivante par mettre en cache les informations de schéma d'une table de base de données :

```
[
    __CLASS__,              // schema class name
    $this->db->dsn,         // DB connection data source name
    $this->db->username,    // DB connection login user
    $name,                  // table name
];
```

Comme vous le constatez, la clé inclut toutes les informations nécessaires pour spécifier de manière unique une table de base de données.

> Note : les valeurs stockées dans le cache via `multiSet()` ou `multiAdd()` peuvent n'avoir que des clés sous forme de chaînes de caractères ou de nombres entiers. Si vous avez besoin de définir des clés plus complexes, stockez la valeur séparément via `set()` ou `add()`.

Lorsque le même support de cache est utilisé par différentes applications, vous devriez spécifier un préfixe de clé de cache pour chacune des applications afin d'éviter les conflits de clés de cache. Cela peut être fait en configurant la propriété `yii\caching\Cache::$keyPrefix`. Par exemple, dans la configuration de l'application vous pouvez entrer le code suivant :

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
        'keyPrefix' => 'myapp',       // a unique cache key prefix
    ],
],
```

Pour garantir l'interopérabilité, vous ne devez utiliser que des caractères alpha-numériques.

Expiration de la mise en cache

Une donnée stockée dans le cache y restera à jamais sauf si elle en est retirée par l'application d'une quelconque politique de mise en cache (p. ex.

l'espace de mise en cache est plein et les données les plus anciennes sont retirées). Pour modifier ce comportement, vous pouvez fournir un paramètre d'expiration lors de l'appel de la fonction `set()` pour stocker une donnée. Le paramètre indique pour combien de secondes la donnée restera valide dans le cache. Lorsque vous appelez la fonction `get()` pour récupérer une donnée, si cette dernière a expiré, la méthode retourne `false`, pour indiquer que la donnée n'a pas été trouvée dans le cache. Par exemple,

```
// conserve la donnée dans le cache pour un maximum de 45 secondes
$cache->set($key, $data, 45);

sleep(50);

$data = $cache->get($key);
if ($data === false) {
    // $data a expiré ou n'a pas été trouvée dans le cache
}
```

Depuis la version 2.0.11, vous pouvez définir une valeur `defaultDuration` dans la configuration de votre composant de mise en cache si vous préférez utiliser une durée de mise en cache personnalisée au lieu de la durée illimitée par défaut. Cela vous évitera d'avoir à passer la durée personnalisée à la fonction `set()` à chaque fois.

Dépendances de mise en cache

En plus de la définition du temps d'expiration, les données mises en cache peuvent également être invalidées par modification de ce qu'on appelle les *dépendances de mise en cache*. Par exemple, `yii\caching\FileDependency` représente la dépendance à la date de modification d'un fichier. Lorsque cette dépendance est modifiée, cela signifie que le fichier correspondant est modifié. En conséquence, tout contenu de fichier périmé trouvé dans le cache devrait être invalidé et l'appel de la fonction `get()` devrait retourner `false`.

Les dépendances de mise en cache sont représentées sous forme d'objets dérivés de `yii\caching\Dependency`. Lorsque vous appelez la fonction `set()` pour stocker une donnée dans le cache, vous pouvez lui passer un objet de dépendance (« Dependency ») associé. Par exemple,

```
// Crée une dépendance à la date de modification du fichier example.txt
$dependency = new \yii\caching\FileDependency(['fileName' =>
'example.txt']);

// La donnée expirera dans 30 secondes.
// Elle sera également invalidée plus tôt si le fichier example.txt est
modifié.
$cache->set($key, $data, 30, $dependency);

// Le cache vérifiera si la donnée a expiré.
```

```
// Il vérifiera également si la dépendance associée a été modifiée.
// Il retournera `false` si l'une de ces conditions est vérifiée.
$data = $cache->get($key);
```

Ci-dessous nous présentons un résumé des dépendances de mise en cache disponibles :

- `yii\caching\ChainedDependency` : la dépendance est modifiée si l'une des dépendances de la chaîne est modifiée.
- `yii\caching\DbDependency` : la dépendance est modifiée si le résultat de le requête de l'instruction SQL spécifiée est modifié.
- `yii\caching\ExpressionDependency` : la dépendance est modifiée si le résultat de l'expression PHP spécifiée est modifié.
- `yii\caching\CallbackDependency` : la dépendance est modifiée si le résultat du rappel PHP spécifié est modifié.
- `yii\caching\FileDependency` : la dépendance est modifiée si la date de dernière modification du fichier est modifiée.
- `yii\caching\TagDependency` : associe une donnée mise en cache à une ou plusieurs balises. Vous pouvez invalider la donnée mise en cache associée à la balise spécifiée en appelant `yii\caching\TagDependency::invalidate()`.

  Note : évitez d'utiliser la méthode `exists()` avec des dépendances. Cela ne vérifie pas si la dépendance associée à la donnée mise en cache, s'il en existe une, a changé. Ainsi, un appel de la fonction `get()` peut retourner `false` alors que l'appel de la fonction `exists()` retourne `true`.

10.2.3 Mise en cache de requêtes

La mise en cache de requêtes est une fonctionnalité spéciale de la mise en cache construite sur la base de la mise en cache de données. Elle est fournie pour permettre la mise en cache du résultat de requêtes de base de données.

La mise en cache de requêtes nécessite une `connexion à une base de données` et un composant d'application`cache` valide. L'utilisation de base de la mise en cache de requêtes est la suivante, en supposant que `$db` est une instance de `yii\db\Connection` :

```
$result = $db->cache(function ($db) {

    // le résultat d'une requête SQL sera servi à partir du cache
    // si la mise en cache de requêtes est activée et si le résultat de la
    requête est trouvé dans le cache
    return $db->createCommand('SELECT * FROM customer WHERE
    id=1')->queryOne();

});
```

La mise en cache de requêtes peut être utilisée pour des DAO ainsi que pour des enregistrements actifs :

```
$result = Customer::getDb()->cache(function ($db) {
    return Customer::find()->where(['id' => 1])->one();
});
```

> Info : quelques systèmes de gestion de bases de données (DBMS) (p. ex. MySQL [10]) prennent également en charge la mise en cache de requêtes du côté serveur de base de données. Vous pouvez choisir d'utiliser l'un ou l'autre des ces mécanismes de mise en cache de requêtes. Les mises en cache de requêtes décrites ci-dessus offrent l'avantage de pouvoir spécifier des dépendances de mise en cache flexibles et sont potentiellement plus efficaces.

Vidage du cache

Lorsque vous avez besoin d'invalider toutes les données stockées dans le cache, vous pouvez appeler `yii\caching\Cache::flush()`.

Vous pouvez aussi vider le cache depuis la console en appelant `yii cache/flush`.

— `yii cache` : liste les caches disponibles dans une application
— `yii cache/flush cache1 cache2` : vide les composants de mise en cache `cache1`, `cache2` (vous pouvez passer de multiples composants en les séparant par une virgule)
— `yii cache/flush-all` : vide tous les composants de mise en cache de l'application

> Info : les applications en console utilisent un fichier de configuration séparé par défaut. Assurez-vous que vous utilisez le même composant de mise en cache dans les configurations de vos application web et console pour obtenir l'effet correct.

Configurations

La mise en cache de requêtes dispose de trois options globales configurables via `yii\db\Connection` :

— `enableQueryCache` : pour activer ou désactiver la mise en cache de requêtes. Valeur par défaut : `true`. Notez que pour activer effectivement la mise en cache de requêtes, vous devez également disposer d'un cache valide, tel que spécifié par `queryCache`.
— `queryCacheDuration` : ceci représente le nombre de secondes durant lesquelles le résultat d'une requête reste valide dans le cache. Vous pouvez utiliser 0 pour indiquer que le résultat de la requête doit rester valide indéfiniment dans le cache. Cette propriété est la valeur par défaut

10. https://dev.mysql.com/doc/refman/5.6/en/query-cache.html

utilisée lors de l'appel `yii\db\Connection::cache()` sans spécifier de durée.
— `queryCache` : ceci représente l'identifiant du composant d'application de mise en cache. Sa valeur par défaut est : `'cache'`. La mise en cache de requêtes est activée seulement s'il existe un composant d'application de mise en cache valide.

Utilisations

Vous pouvez utiliser `yii\db\Connection::cache()` si vous avez de multiples requêtes SQL qui doivent bénéficier de la mise en cache de requêtes. On l'utilise comme suit :

```
$duration = 60;     // mettre le résultat de la requête en cache durant 60
secondes.
$dependency = ...;  // dépendance optionnelle

$result = $db->cache(function ($db) {

    // ... effectuer les requêtes SQL ici ...

    return $result;

}, $duration, $dependency);
```

Toutes les requêtes SQL dans la fonction anonyme sont mises en cache pour la durée spécifiée avec la dépendance spécifiée. Si le résultat d'une requête est trouvé valide dans le cache, la requête est ignorée et, à la place, le résultat est servi à partir du cache. Si vous ne spécifiez pas le paramètre `$duration`, la valeur de `queryCacheDuration` est utilisée en remplacement.

Parfois, dans `cache()`, il se peut que vous vouliez désactiver la mise en cache de requêtes pour des requêtes particulières. Dans un tel cas, vous pouvez utiliser `yii\db\Connection::noCache()`.

```
$result = $db->cache(function ($db) {

    // requêtes SQL qui utilisent la mise en cache de requêtes

    $db->noCache(function ($db) {

        // requêtes qui n'utilisent pas la mise en cache de requêtes

    });

    // ...

    return $result;
});
```

Si vous voulez seulement utiliser la mise en cache de requêtes pour une requête unique, vous pouvez appeler la fonction `yii\db\Command::cache()` lors de la construction de la commande. Par exemple :

```
// utilise la mise en cache de requêtes et définit la durée de mise en cache
de la requête à 60 secondes
$customer = $db->createCommand('SELECT * FROM customer WHERE
id=1')->cache(60)->queryOne();
```

Vous pouvez aussi utiliser la fonction `yii\db\Command::noCache()` pour désactiver la mise en cache de requêtes pour une commande unique. Par exemple :

```
$result = $db->cache(function ($db) {

    // requêtes SQL qui utilisent la mise en cache de requêtes

    // ne pas utiliser la mise en cache de requêtes pour cette commande
    $customer = $db->createCommand('SELECT * FROM customer WHERE
    id=1')->noCache()->queryOne();

    // ...

    return $result;
});
```

Limitations

La mise en cache de requêtes ne fonctionne pas avec des résultats de requêtes qui contiennent des gestionnaires de ressources. Par exemple, lorsque vous utilisez de type de colonne `BLOB` dans certains systèmes de gestion de bases de données (DBMS), la requête retourne un gestionnaire de ressources pour la donnée de la colonne.

Quelques supports de stockage pour cache sont limités en taille. Par exemple, avec memcache, chaque entrée est limitée en taille à 1 MO. En conséquence, si le résultat d'une requête dépasse cette taille, la mise en cache échoue.

10.3 Mise en cache de fragments

La mise en cache de fragments fait référence à la mise en cache de fragments de pages Web. Par exemple, si une page affiche un résumé des ventes annuelles dans un tableau, vous pouvez stocker ce tableau en cache pour éliminer le temps nécessaire à sa génération à chacune des requêtes. La mise en cache de fragments est construite au-dessus de la mise en cache de données.

Pour utiliser la mise en cache de fragments, utilisez la construction qui suit dans une vue :

```
if ($this->beginCache($id)) {

    // ... générez le contenu ici ...

    $this->endCache();
}
```

C'est à dire, insérez la logique de génération du contenu entre les appels `beginCache()` et `endCache()`. Si le contenu est trouvé dans le cache, `beginCache()` rendra le contenu en cache et retournera `false` (faux), ignorant la logique de génération de contenu. Autrement, votre logique de génération de contenu sera appelée, et quand `endCache()` sera appelée, le contenu généré sera capturé et stocké dans le cache.

Comme pour la mise en cache de données, un `$id` (identifiant) unique est nécessaire pour identifier un cache de contenu.

10.3.1 Options de mise en cache

Vous pouvez spécifier des options additionnelles sur la mise en cache de fragments en passant le tableau d'options comme second paramètre à la méthode `beginCache()`. En arrière plan, ce tableau d'options est utilisé pour configurer un composant graphique `yii\widgets\FragmentCache` qui met en œuvre la fonctionnalité réelle de mise en cache de fragments.

Durée

L'option `duration` (durée) est peut-être l'option de la mise en cache de fragments la plus couramment utilisée. Elle spécifie pour combien de secondes le contenu peut demeurer valide dans le cache. Le code qui suit met le fragment de contenu en cache pour au maximum une heure :

```
if ($this->beginCache($id, ['duration' => 3600])) {

    // ... générez le contenu ici...

    $this->endCache();
}
```

Si cette option n'est pas définie, la valeur utilisée par défaut est 60, ce qui veut dire que le contenu mise en cache expirera au bout de 60 secondes.

Dépendances

Comme pour la mise en cache de données, le fragment de contenu mis en cache peut aussi avoir des dépendances. Par exemple, le contenu d'un article affiché dépend du fait que l'article a été modifié ou pas.

Pour spécifier une dépendance, définissez l'option `dependency`, soit sous forme d'objet `yii\caching\Dependency`, soit sous forme d'un tableau de

configuration pour créer un objet `yii\caching\Dependency`. Le code qui suit spécifie que le fragment de contenu dépend du changement de la valeur de la colonne `updated_at` (mis à jour le) :

```
$dependency = [
    'class' => 'yii\caching\DbDependency',
    'sql' => 'SELECT MAX(updated_at) FROM post',
];

if ($this->beginCache($id, ['dependency' => $dependency])) {

    // ... générez le contenu ici ...

    $this->endCache();
}
```

Variations

Le contenu mise en cache peut connaître quelques variations selon certains paramètres. Par exemple, pour une application Web prenant en charge plusieurs langues, le même morceau de code d'une vue peut générer le contenu dans différentes langues. Par conséquent, vous pouvez souhaitez que le contenu mis en cache varie selon la langue courante de l'application.

Pour spécifier des variations de mise en cache, définissez l'option `variations`, qui doit être un tableau de valeurs scalaires, représentant chacune un facteur de variation particulier. Par exemple, pour que le contenu mis en cache varie selon la langue, vous pouvez utiliser le code suivant :

```
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {

    // ... générez le contenu ici ...

    $this->endCache();
}
```

Activation désactivation de la mise en cache

Parfois, vous désirez activer la mise en cache de fragments seulement lorsque certaines conditions sont rencontrées. Par exemple, pour une page qui affiche un formulaire, vous désirez seulement mettre le formulaire en cache lorsqu'il est initialement demandé (via une requête GET). Tout affichage subséquent du formulaire (via des requêtes POST) ne devrait pas être mise en cache car il contient des données entrées par l'utilisateur. Pour mettre en œuvre ce mécanisme, vous pouvez définir l'option `enabled`, comme suit :

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {

    // ... générez le contenu ici ...
```

```
    $this->endCache();
}
```

10.3.2 Mises en cache imbriquées

La mise en cache de fragments peut être imbriquée. C'est à dire qu'un fragment mis en cache peut être contenu dans un autre fragment lui aussi mis en cache. Par exemple, les commentaires sont mis en cache dans un cache de fragment interne, et sont mis en cache en même temps et avec le contenu de l'article dans un cache de fragment externe. Le code qui suit montre comment deux caches de fragment peuvent être imbriqués :

```
if ($this->beginCache($id1)) {

    // ...logique de génération du contenu ...

    if ($this->beginCache($id2, $options2)) {

        // ...logique de génération du contenu...

        $this->endCache();
    }

    // ... logique de génération de contenu ...

    $this->endCache();
}
```

Différentes options de mise en cache peuvent être définies pour les caches imbriqués. Par exemple, les caches internes et les caches externes peuvent utiliser des valeurs de durée différentes. Même lorsque les données mises en cache dans le cache externe sont invalidées, le cache interne peut continuer à fournir un fragment interne valide. Néanmoins, le réciproque n'est pas vraie ; si le cache externe est évalué comme valide, il continue à fournir la même copie mise en cache après que le contenu du cache interne a été invalidé. Par conséquent, vous devez être prudent lors de la définition des durées ou des dépendances des caches imbriqués, autrement des fragments internes périmés peuvent subsister dans le fragment externe.

10.3.3 Contenu dynamique

Lors de l'utilisation de la mise en cache de fragments, vous pouvez rencontrer une situation dans laquelle un gros fragment de contenu est relativement statique en dehors de quelques endroits particuliers. Par exemple, l'entête d'une page peut afficher le menu principal avec le nom de l'utilisateur courant. Un autre problème se rencontre lorsque le contenu mis en cache, contient du code PHP qui doit être exécuté à chacune des requêtes (p. ex. le

code pour enregistrer un paquet de ressources). Ces deux problèmes peuvent être résolus par une fonctionnalité qu'on appelle *contenu dynamique*.

Un contenu dynamique signifie un fragment de sortie qui ne doit jamais être mis en cache même s'il est contenu dans un fragment mis en cache. Pour faire en sorte que le contenu soit dynamique en permanence, il doit être généré en exécutant un code PHP à chaque requête, même si le contenu l'englobant est servi à partir du cache.

Vous pouvez appeler la fonction `yii\base\View::renderDynamic()` dans un fragment mis en cache pour y insérer un contenu dynamique à l'endroit désiré, comme ceci :

```
if ($this->beginCache($id1)) {

    // ... logique de génération de contenu ...

    echo $this->renderDynamic('return Yii::$app->user->identity->name;');

    // ... logique de génération de contenu ...

    $this->endCache();
}
```

La méthode `renderDynamic()` accepte un morceau de code PHP en paramètre. La valeur retournée est traitée comme un contenu dynamique. Le même code PHP est exécuté à chacune des requêtes, peu importe que le fragment englobant soit servi à partir du cache ou pas.

10.4 Mise en cache de pages

La mise en cache de pages fait référence à la mise en cache du contenu d'une page entière du côté serveur. Plus tard, lorsque la même page est demandée à nouveau, son contenu est servi à partir du cache plutôt que d'être régénéré entièrement.

La mise en cache de pages est prise en charge par `yii\filters\PageCache`, un filtre d'action. On peut l'utiliser de la manière suivante dans une classe contrôleur :

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\PageCache',
            'only' => ['index'],
            'duration' => 60,
            'variations' => [
                \Yii::$app->language,
            ],
            'dependency' => [
```

```
                'class' => 'yii\caching\DbDependency',
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
        ],
    ];
}
```

Le code ci-dessus établit que la mise en cache de pages doit être utilisée uniquement pour l'action `index`. Le contenu de la page doit être mis en cache pour au plus 60 secondes et doit varier selon la langue courante de l'application. De plus, le contenu de la page mis en cache doit être invalidé si le nombre total d'articles (post) change.

Comme vous pouvez le constater, la mise en cache de pages est très similaire à la mise en cache de fragments. Les deux prennent en charge les options telles que `duration`, `dependencies`, `variations` et `enabled`. La différence principale est que la mise en cache de pages est mis en œuvre comme un filtre d'action alors que la mise en cache de framgents l'est comme un composant graphique.

Vous pouvez utiliser la mise en cache de fragments ainsi que le contenu dynamique en simultanéité avec la mise en cache de pages.

10.5 Mise en cache HTTP

En plus de la mise en cache côté serveur que nous avons décrite dans les sections précédentes, les applications Web peuvent aussi exploiter la mise en cache côté client pour économiser le temps de génération et de transfert d'un contenu de page inchangé.

Pour utiliser la mise en cache côté client, vous pouvez configurer `yii\filters\HttpCache` comme un filtre pour des actions de contrôleur dont le résultat rendu peut être mis en cache du côté du client. `HttpCache` ne fonctionne que pour les requêtes `GET` et `HEAD`. Il peut gérer trois sortes d'entêtes HTTP relatifs à la mise en cache pour ces requêtes :

- `Last-Modified`
- `Etag`
- `Cache-Control`

10.5.1 Entête

L'entête `Last-Modified` (dernière modification) utilise un horodatage pour indiquer si la page a été modifiée depuis sa mise en cache par le client.

Vous pouvez configurer la propriété `yii\filters\HttpCache::$lastModified` pour activer l'envoi de l'entête `Last-modified`. La propriété doit être une fonction de rappel PHP qui retourne un horodatage UNIX concernant la modification de la page. La signature de la fonction de rappel PHP doit être comme suit :

```
/**
 * @param Action $action l'objet action qui est actuellement géré
 * @param array $params la valeur de la propriété "params"
 * @return int un horodatage UNIX représentant l'instant de modification de
la page
 */
function ($action, $params)
```

Ce qui suit est un exemple d'utilisation de l'entête `Last-Modified` :

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('post')->max('updated_at');
            },
        ],
    ];
}
```

Le code précédent établit que la mise en cache HTTP doit être activée pour l'action `index` seulement. Il doit générer un entête HTTP `Last-Modified` basé sur l'instant de la dernière mise à jour d'articles (posts). Lorsque le navigateur visite la page `index` pour la première fois, la page est générée par le serveur et envoyée au navigateur. Si le navigateur visite à nouveau la même page, et qu'aucun article n'a été modifié, le serveur ne régénère par la page, et le navigateur utilise la version mise en cache du côté du client. En conséquence, le rendu côté serveur et la transmission de la page sont tous deux évités.

10.5.2 Entête

L'entête "Entity Tag" (or `ETag` en raccourci) utilise une valeur de hachage pour représenter le contenu d'une page. Si la page est modifiée, la valeur de hachage change également. En comparant la valeur de hachage conservée sur le client avec la valeur de hachage générée côté serveur, le cache peut déterminer si la page a été modifiée et doit être retransmise.

Vous pouvez configurer la propriété `yii\filters\HttpCache::$etagSeed` pour activer l'envoi de l'entête `ETag`. La propriété doit être une fonction de rappel PHP qui retourne un nonce (sel) pour la génération de la valeur de hachage Etag. La signature de la fonction de rappel PHP doit être comme suit :

```
/**
 * @param Action $action l'objet action qui est actuellement géré
```

```
     * @param array $params la valeur de la propriété "params"
     * @return string une chaîne de caractères à utiliser comme nonce (sel) pour
    la génération d'une valeur de hachage ETag
     */
    function ($action, $params)
```

Ce qui suit est un exemple d'utilisation de l'entête `ETag` :

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['view'],
            'etagSeed' => function ($action, $params) {
                $post = $this->findModel(\Yii::$app->request->get('id'));
                return serialize([$post->title, $post->content]);
            },
        ],
    ];
}
```

Le code ci-dessus établit que la mise en cache HTTP doit être activée pour l'action `view` seulement. Il doit générer un entête HTTP `ETag` basé sur le titre et le contenu de l'article demandé. Lorsque le navigateur visite la page pour la première fois, la page est générée par le serveur et envoyée au navigateur. Si le navigateur visite à nouveau la même page et que ni le titre, ni le contenu de l'article n'ont changé, le serveur ne régénère pas la page et le navigateur utilise la version mise en cache côté client. En conséquence, le rendu par le serveur et la transmission de la page sont tous deux évités.

ETags vous autorise des stratégies de mises en cache plus complexes et/ou plus précises que l'entête `Last-Modified`. Par exemple, un ETag peut être invalidé si on a commuté le site sur un nouveau thème.

Des génération coûteuses d'ETag peuvent contrecarrer l'objectif poursuivi en utilisant `HttpCache` et introduire une surcharge inutile, car il faut les réévaluer à chacune des requêtes. Essayez de trouver une expression simple qui invalide le cache si le contenu de la page a été modifié.

> Note : en conformité avec la norme RFC 7232[11], `HttpCache` envoie les entêtes `ETag` et `Last-Modified` à la fois si ils sont tous deux configurés. Et si le client envoie les entêtes `If-None-Match` et `If-Modified-Since` à la fois, seul le premier est respecté.

10.5.3 Entête

L'entête `Cache-Control` spécifie la politique de mise en cache générale pour les pages. Vous pouvez l'envoyer en configurant la propriété `yii\filters`

11. https://datatracker.ietf.org/doc/html/rfc7232#section-2.4

`\HttpCache::$cacheControlHeader` avec la valeur de l'entête. Par défaut, l'entête suivant est envoyé :

```
Cache-Control: public, max-age=3600
```

10.5.4 Propriété "Session Cache Limiter"

Lorsqu'une page utilise une session, PHP envoie automatiquement quelques entêtes HTTP relatifs à la mise en cache comme spécifié dans la propriété `session.cache_limiter` de PHP INI. Ces entêtes peuvent interférer ou désactiver la mise en cache que vous voulez obtenir de `HttpCache`. Pour éviter ce problème, par défaut, `HttpCache` désactive l'envoi de ces entêtes automatiquement. Si vous désirez modifier ce comportement, vous devez configurer la propriété `yii\filters\HttpCache::$sessionCacheLimiter`. Cette propriété accepte une chaîne de caractères parmi `public`, `private`, `private_no_expire` et `nocache`. Reportez-vous au manuel de PHP à propos de session_cache_limiter() [12] pour des explications sur ces valeurs.

10.5.5 Implications SEO

Les robots moteurs de recherche ont tendance à respecter les entêtes de mise en cache. Comme certains moteurs d'indexation du Web sont limités quant aux nombre de pages par domaine qu'ils sont à même de traiter dans un certain laps de temps, l'introduction d'entêtes de mise en cache peut aider à l'indexation de votre site car ils limitent le nombre de pages qui ont besoin d'être traitées.

12. https://www.php.net/manual/fr/function.session-cache-limiter.php

Chapitre 11

Services Web RESTful

Error : not existing file : rest-quick-start.md

Error : not existing file : rest-resources.md

Error : not existing file : rest-controllers.md

Error : not existing file : rest-routing.md

Error : not existing file : rest-response-formatting.md

Error : not existing file : rest-authentication.md

Error : not existing file : rest-rate-limiting.md

Error : not existing file : rest-versioning.md

Error : not existing file : rest-error-handling.md

Chapitre 12

Outils de développement

Error : not existing file : tool-debugger.md

Error : not existing file : tool-gii.md

Error : not existing file : tool-api-doc.md

Chapitre 13

Tests

Error : not existing file : test-overview.md

Error : not existing file : test-unit.md

Error : not existing file : test-functional.md

Error : not existing file : test-acceptance.md

Error : not existing file : test-fixtures.md

Chapitre 14

Etendre Yii

Error : not existing file : extend-creating-extensions.md

Error : not existing file : extend-customizing-core.md

Error : not existing file : extend-using-libs.md

Error : not existing file : extend-embedding-in-others.md

Error : not existing file : extend-using-v1-v2.md

Error : not existing file : extend-using-composer.md

Chapitre 15

Sujets avancés

Error : not existing file : tutorial-advanced-app.md

Error : not existing file : tutorial-start-from-scratch.md

Error : not existing file : tutorial-console.md

Error : not existing file : tutorial-core-validators.md

15.1 Internationalisation

Le terme *Internationalisation* (I18N) fait référence au processus de conception d'une application logicielle qui permet son adaptation à diverses langues et régions sans intervenir dans le code. Pour des applications Web, la chose est particulièrement importante puisque celle-ci peut concerner des utilisateurs potentiels répartis sur toute la surface de la terre. Yii met à votre disposition tout un arsenal de fonctionnalités qui prennent en charge la traduction des messages et des vues, ainsi que le formatage des nombres et des dates.

15.1.1 Locale et Langue

Une *locale* est un jeu de paramètres qui définissent la langue de l'utilisateur, son pays et des préférences spéciales que celui-ci désire voir dans l'interface utilisateur.

Elle est généralement identifiée par un identifiant (ID), lui-même constitué par un identifiant de langue et un identifiant de région. Par exemple, l'identifiant `en-US` représente la locale *anglais* pour la langue et *États-Unis* pour la région.

Pour assurer la cohérence, tous les identifiants utilisés par les application Yii doivent être présentés sous leur forme canonique `ll-CC`, où `ll` est un code à 2 ou 3 lettres pour la langue conforme à la norme ISO-639 [1] et `CC` est un code à deux lettres pour le pays conforme à la norme ISO-3166 [2]. Pour plus de détails sur les locales, reportez-vous à la documentation du projet ICU [3].

Dans Yii, nous utilisons souvent le mot « langue » pour faire référence à la locale.

Une application Yii utilise deux langues : la `langue source` et la `langue cible`. La première fait référence à la langue dans laquelle les messages sont rédigés dans le code source, tandis que la deuxième est celle qui est utilisée pour présenter les textes à l'utilisateur final. Pour l'essentiel, le service appelé *message translation service*(service de traduction des messages) assure la traduction d'un message textuel de la langue source vers la langue cible.

Vous pouvez configurer les langues de l'application dans la configuration de la manière suivante :

```
return [
    // définit la langue cible comme étant le français-France
    'language' => 'fr-FR',

    // définit la langue source comme étant l'anglais États-Unis
    'sourceLanguage' => 'en-US',
```

1. https://www.loc.gov/standards/iso639-2/
2. https://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html
3. https://unicode-org.github.io/icu/userguide/locale/#the-locale-concept

```
    ......
];
```

La valeur par défaut pour la `langue source` est `en-US`, qui signifie « anglais États-Unis ». Il est recommandé de conserver cette valeur sans la changer car il est généralement plus facile de trouver des gens capables de traduire de l'anglais vers d'autres langues que d'une langue non anglaise vers une autre langue.

Il est souvent nécessaire de définir la `langue cible` de façon dynamique en se basant sur différents facteurs tels que, par exemple, les préférences linguistiques de l'utilisateur final. Au lieu de la définir dans la configuration de l'application vous pouvez utiliser l'instruction suivante pour changer la langue cible :

```
// modifier la langue cible pour qu'elle soit français-FRANCE
\Yii::$app->language = 'fr-FR';
```

> **Conseil :** si votre langue source change selon les différentes parties de votre code, vous pouvez modifier la valeur de la langue source localement comme c'est expliqué dans la section suivante.

15.1.2 Traduction des messages

Le service de traduction des messages traduit un message textuel d'une langue (généralement la `langue source`) vers une autre langue (généralement la `langue cible`). Il effectue la traduction en recherchant le message à traduire dans une source de messages qui stocke les messages originaux et les messages traduits. Si le message est trouvé, le message traduit correspondant est renvoyé ; dans le cas contraire, le message original est renvoyé sans traduction.

Pour utiliser le service de traduction des messages, vous devez principalement effectuer les opérations suivantes :

— Envelopper le message textuel à traduire dans un appel à la méthode `Yii::t()` ;
— Configurer une ou plusieurs sources de messages dans lesquelles le service de traduction des messages peut rechercher des traductions ;
— Confier aux traducteurs le soin de traduire les messages et de les stocker dans les sources de messages.

La méthode `Yii::t()` peut être utilisée comme le montre l'exemple suivant :

```
echo \Yii::t('app', 'This is a string to translate!');
```

où le deuxième paramètre fait référence au message textuel à traduire, tandis que le premier paramètre fait référence au nom de la catégorie à laquelle le message appartient.

La méthode `Yii::t()` appelle la méthode `translate` du composant d'application `i18n` pour assurer le travail réel de traduction. Le composant peut être configuré dans la configuration de l'application de la manière suivante :

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                //'basePath' => '@app/messages',
                //'sourceLanguage' => 'en-US',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
            ],
        ],
    ],
],
```

Dans le code qui précède, une source de messages prise en charge par `yii\i18n\PhpMessageSource` est configurée. Le motif `app*` indique que toutes les catégories de messages dont les noms commencent par `app` doivent être traduites en utilisant cette source de messages. La classe `yii\i18n\PhpMessageSource` utilise des fichiers PHP pour stocker les traductions de messages. Chacun des fichiers PHP correspond aux messages d'une même catégorie. Par défaut, le nom du fichier doit être celui de la catégorie. Néanmoins, vous pouvez configurer `fileMap (table de mise en correspondance de fichiers)` pour faire correspondre une catégorie à un fichier PHP dont le nom obéit à une autre approche de nommage. Dans l'exemple qui précède, la catégorie `app/error` correspond au fichier PHP `@app/messages/fr-FR/error.php` (en supposant que `fr-FR` est la langue cible). Sans cette configuration, la catégorie correspondrait à `@app/messages/fr-FR/app/error.php`.

En plus de la possibilité de stocker les messages dans des fichiers PHP, vous pouvez aussi utiliser les sources de messages suivantes pour stocker vos traductions sous une autre forme :

— `yii\i18n\GettextMessageSource` utilise des fichiers GNU Gettext, MO ou PO pour maintenir les messages traduits.
— `yii\i18n\DbMessageSource` utilise une base de donnée pour stocker les messages traduits.

15.1.3 Format des messages

Lorsque vous traduisez un message, vous pouvez inclure dans le messages des « valeurs à remplacer » qui seront remplacées dynamiquement en fonction de la valeur d'un paramètre. Vous pouvez même utiliser une syntaxe spéciale des « valeurs à remplacer » pour que les valeurs de remplacement soient

formatées en fonction de la langue cible. Dans cette sous-section, nous allons décrire différentes manières de formater un message.

Valeurs à remplacer des message

Dans un message à traduire, vous pouvez inclure une ou plusieurs « valeurs à remplacer » pour qu'elles puissent être remplacées par les valeurs données. En spécifiant différents jeux de valeurs, vous pouvez faire varier le message dynamiquement. Dans l'exemple qui suit, la valeur à remplacer `{username}` du message `'Hello, {username}!'` sera remplacée par `'Alexander'` et `'Qiang'`, respectivement.

```
$username = 'Alexander';
// affiche un message traduit en remplaçant {username} par "Alexander"
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);

$username = 'Qiang';
// affiche un message traduit en remplaçant {username} par "Qiang"
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);
```

Lorsque le traducteur traduit un message contenant une valeur à remplacer, il doit laisser la valeur à remplacer telle quelle. Cela tient au fait que les valeurs à remplacer seront remplacées par les valeurs réelles au moment de l'appel de `Yii::t()` pour traduire le message.

Dans un même message, vous pouvez utiliser, soit des « valeurs à remplacer nommées », soit des « valeurs à remplacer positionnelles », mais pas les deux types.

L'exemple précédent montre comment utiliser des valeurs à remplacer nommées, c'est à dire, des valeurs à remplacer écrites sous la forme `{nom}`, et pour lesquelles vous fournissez un tableau associatif dont les clés sont les noms des valeurs à remplacer (sans les accolades) et les valeurs, les valeurs de remplacement.

Les valeurs à remplacer positionnelles utilisent une suite d'entiers démarrant de zéro en tant que noms de valeurs à remplacer qui seront remplacées par les valeurs de remplacement, fournies sous forme d'un tableau, en fonction de leur position dans le tableau lors de l'appel de la méthode `Yii::t()`. Dans l'exemple suivant, les valeurs à remplacer positionnelles `{0}`, `{1}` et `{2}` seront remplacées respectivement par les valeurs de `$price`, `$count` et `$subtotal`.

```
$price = 100;
$count = 2;
$subtotal = 200;
echo \Yii::t('app', 'Price: {0}, Count: {1}, Subtotal: {2}', [$price,
$count, $subtotal]);
```

Dans le cas d'une seule valeur à remplacer, la valeur de remplacement peut être donnée sans la placer dans un tableau :

```
echo \Yii::t('app', 'Price: {0}', $price);
```

> **Conseil :** dans la plupart des cas, vous devriez utiliser des valeurs à remplacer nommées, parce que les noms permettent aux traducteurs de mieux comprendre le sens des messages qu'ils doivent traduire.

Formatage des valeurs de remplacement

Vous pouvez spécifier des règles de formatage additionnelles dans les valeurs à remplacer qui seront appliquées aux valeurs de remplacement. Dans l'exemple suivant, la valeur de remplacement *price* est traitée comme un nombre et formatée comme une valeur monétaire :

```
$price = 100;
echo \Yii::t('app', 'Price: {0,number,currency}', $price);
```

> **Note :** le formatage des valeurs de remplacement nécessite l'installation de extension intl de PHP [4].

Vous pouvez utiliser, soit la forme raccourcie, soit la forme complète pour spécifier une valeur à remplacer avec un format : `forme raccourcie : {name,type}` `forme complète : {name,type,style}`

> **Note :** si vous avez besoin des caractères spéciaux tels que `{}`, `\mintinline{text}{}`, `'`, `#`, entourez-les de `'` :
>
> ```
> echo Yii::t('app', "Example of string with ''-escaped characters'':
> '{' '}' '{test}' {count,plural,other{''count'' value is # '#{}'}}",
> ['count' => 3]);
> ```

Le format complet est décrit dans la documentation ICU [5].

Dans ce qui suit, nous allons présenter quelques usages courants.

Nombres La valeur de remplacement est traitée comme un nombre. Par exemple,

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0,number}', $sum);
```

Vous pouvez spécifier un style facultatif pour la valeur de remplacement `integer` (entier), `currency` (valeur monétaire), ou `percent` (pourcentage) :

4. https://www.php.net/manual/fr/intro.intl.php
5. https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classMessageFormat.html

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0,number,currency}', $sum);
```

Vous pouvez aussi spécifier un motif personnalisé pour formater le nombre. Par exemple,

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0,number,,000,000000}', $sum);
```

Les caractères à utiliser dans les formats personnalisés sont présentés dans le document ICU API reference [6] à la section "Special Pattern Characters" (Caractères pour motifs spéciaux).

La valeur de remplacement est toujours formatée en fonction de la locale cible c'est à dire que vous ne pouvez pas modifier les séparateurs de milliers et de décimales, les symboles monétaires, etc. sans modifier la locale de traduction. Si vous devez personnaliser ces éléments vous pouvez utiliser `yii\i18n\Formatter::asDecimal()` et `yii\i18n\Formatter::asCurrency()`.

Date La valeur de remplacement doit être formatée comme une date. Par exemple,

```
echo \Yii::t('app', 'Today is {0,date}', time());
```

Vous pouvez spécifier des styles facultatifs pour la valeur de remplacement comme `short` (court), `medium` (moyen), `long` (long) ou `full` (complet) :

```
echo \Yii::t('app', 'Today is {0,date,short}', time());
```

Vous pouvez aussi spécifier un motif personnalisé pour formater la date :

```
echo \Yii::t('app', 'Today is {0,date,yyyy-MM-dd}', time());
```

Voir Formatting reference [7].

Heure La valeur de remplacement doit être formatée comme une heure (au sens large heure minute seconde). Par exemple,

```
echo \Yii::t('app', 'It is {0,time}', time());
```

Vous pouvez spécifier des styles facultatifs pour la valeur de remplacement comme `short` (court), `medium` (moyen), `long` (long) ou `full` (complet) :

```
echo \Yii::t('app', 'It is {0,time,short}', time());
```

6. https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classDecimalFormat.html
7. https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classicu_1_1SimpleDateFormat.html#details

Vous pouvez aussi spécifier un motif personnalisé pour formater l'heure :

```
echo \Yii::t('app', 'It is {0,date,HH:mm}', time());
```

Voir Formatting reference [8].

Prononciation La valeur de remplacement doit être traitée comme un nombre et formatée comme une prononciation. Par exemple,

```
//  produit "42 is spelled as forty-two"
echo \Yii::t('app', '{n,number} is spelled as {n,spellout}', ['n' => 42]);
```

Par défaut le nombre est épelé en tant que cardinal. Cela peut être modifié :

```
// produit  "I am forty-seventh agent"
echo \Yii::t('app', 'I am {n,spellout,%spellout-ordinal} agent', ['n' =>
47]);
```

Notez qu'il ne doit pas y avoir d'espace après `spellout,` et avant `%`.

Pour trouver une liste des options disponibles pour votre locale, reportez-vous à "Numbering schemas, Spellout" à https ://intl.rmcreative.ru/ [9].

Nombre ordinal La valeur de remplacement doit être traitée comme un nombre et formatée comme un nombre ordinal. Par exemple,

```
// produit "You are the 42nd visitor here!" (vous êtes le 42e visiteur ici
!)
echo \Yii::t('app', 'You are the {n,ordinal} visitor here!', ['n' => 42]);
```

Les nombres ordinaux acceptent plus de formats pour des langues telles que l'espagnol :

```
// produit 471ª
echo \Yii::t('app', '{n,ordinal,%digits-ordinal-feminine}', ['n' => 471]);
```

Notez qu'il ne doit pas y avoir d'espace après `ordinal,` et avant `%`.

Pour trouver une liste des options disponibles pour votre locale, reportez-vous à "Numbering schemas, Ordinal" à https ://intl.rmcreative.ru/ [10].

8. https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classicu_1_1SimpleDateFormat.html#details
9. https://intl.rmcreative.ru/
10. https://intl.rmcreative.ru/

Durée La valeur de remplacement doit être traitée comme un nombre de secondes et formatée comme une durée. Par exemple,

```
// produit "You are here for 47 sec. already!" (Vous êtes ici depuis 47sec.
déjà !)
echo \Yii::t('app', 'You are here for {n,duration} already!', ['n' => 47]);
```

La durée accepte d'autres formats :

```
// produit  130:53:47
echo \Yii::t('app', '{n,duration,%in-numerals}', ['n' => 471227]);
```

Notez qu'il ne doit pas y avoir d'espace après `duration,` et avant `%`.

Pour trouver une liste des options disponibles pour votre locale, reportez-vous à "Numbering schemas, Duration" à https ://intl.rmcreative.ru/ [11].

Pluriel Les langues diffèrent dans leur manière de marquer le pluriel. Yii fournit un moyen pratique pour traduire les messages dans différentes formes de pluriel qui fonctionne même pour des règles très complexes. Au lieu de s'occuper des règles d'inflexion directement, il est suffisant de fournir la traductions des mots infléchis dans certaines situations seulement. Par exemple,

```
// Lorsque  $n = 0, produit "There are no cats!"
// Losque $n = 1, produit "There is one cat!"
// Lorsque $n = 42, produit  "There are 42 cats!"
echo \Yii::t('app', 'There {n,plural,=0{are no cats} =1{is one cat}
other{are # cats}}!', ['n' => $n]);
```

Dans les arguments des règles de pluralisation ci-dessus, `=` signifie valeur exacte. Ainsi `=0` signifie exactement zéro, `=1` signifie exactement un. `other` signifie n'importe quelle autre valeur. `#` est remplacé par la valeur de `n` formatée selon la langue cible.

Les formes du pluriel peuvent être très compliquées dans certaines langues. Dans l'exemple ci-après en russe, `=1` correspond exactement à `n = 1` tandis que `one` correspond à `21` ou `101` :

```
Здесь {n,plural,=0{котов нет} =1{есть один кот} one{# кот} few{# кота}
many{# котов} other{# кота}}!
```

Ces noms d'arguments spéciaux tels que `other`, `few`, `many` et autres varient en fonction de la langue. Pour savoir lesquels utiliser pour une locale particulière, reportez-vous aux "Plural Rules, Cardinal" à https ://intl.rmcreative.ru/ [12]. En alternative, vous pouvez vous reporter aux rules reference at unicode.org [13].

11. https://intl.rmcreative.ru/
12. https://intl.rmcreative.ru/
13. https://cldr.unicode.org/index/cldr-spec/plural-rules

> **Note :** le message en russe ci-dessus est principalement utilisé comme message traduit, pas comme message source, sauf si vous définissez la `langue source` de votre application comme étant `ru-RU` et traduisez à partir du russe.
>
> Lorsqu'une traduction n'est pas trouvée pour un message source spécifié dans un appel de `Yii::t()`, les règles du pluriel pour la `langue source` seront appliquées au message source. Il existe un paramètre `offset` dans le cas où la chaîne est de la forme suivante :

```
$likeCount = 2;
echo Yii::t('app', 'You {likeCount,plural,
    offset: 1
    =0{did not like this}
    =1{liked this}
    one{and one other person liked this}
    other{and # others liked this}
}', [
    'likeCount' => $likeCount
]);

// You and one other person liked this
```

Sélection ordinale L'argument `selectordinal` pour une valeur à remplacer numérique a pour but de choisir une chaîne de caractères basée sur les règles linguistiques de la locale cible pour les ordinaux. Ainsi,

```
$n = 3;
echo Yii::t('app', 'You are the {n,selectordinal,one{#st} two{#nd} few{#rd}
other{#th}} visitor', ['n' => $n]);
```

//Produit pour l'anglais : //You are the 3rd visitor

//Traduction en russe, 'You are the {n,selectordinal,one{#st} two{#nd} few{#rd} other{#th}} visitor' => 'Вы {n,selectordinal,other{#-й}} посетитель',

//Traduit en russe produit : //Вы 3-й посетитель

//Traduction en français 'You are the {n,selectordinal,one{#st} two{#nd} few{#rd} other{#th}} visitor' => 'Vous êtes le {n,selectordinal,one{#er} other{#e}} visiteur'

//Traduit en français produit : //Vous êtes le 3e visiteur `

Le format est assez proche de celui utilisé pour le pluriel. Pour connaître quels arguments utiliser pour une locale particulière, reportez-vous aux "Plural Rules, Ordinal" à https ://intl.rmcreative.ru/ [14]. En alternative, vous pouvez vous reporter aux rules reference at unicode.org [15].

14. https://intl.rmcreative.ru/
15. https://unicode-org.github.io/cldr-staging/charts/37/supplemental/language_plural_rules.html

Sélection Vous pouvez utiliser l'argument `select` dans une valeur à remplacer pour choisir une phrase en fonction de la valeur de remplacement. Par exemple,

```
// Peut produire "Snoopy is a dog and it loves Yii!"
echo \Yii::t('app', '{name} is a {gender} and {gender,select,female{she}
male{he} other{it}} loves Yii!', [
    'name' => 'Snoopy',
    'gender' => 'dog',
]);
```

Dans l'expression qui précède, `female` et `male` sont des valeurs possibles de l'argument, tandis que `other` prend en compte les valeurs qui ne sont ni l'une ni l'autre des ces valeurs. Derrière chacune des valeurs possibles de l'argument, vous devez spécifier un segment de phrase en l'entourant d'accolades.

Spécification des sources de messages par défaut

Vous pouvez spécifier les sources de messages par défaut qui seront utilisées comme solution de repli pour les catégories qui ne correspondent à aucune des catégories configurées. Cette source de message doit être marquée par un caractère générique `*`. Pour cela ajoutez les lignes suivantes dans la configuration de l'application :

```
//configure i18n component

'i18n' => [
    'translations' => [
        '*' => [
            'class' => 'yii\i18n\PhpMessageSource'
        ],
    ],
],
```

Désormais, vous pouvez utiliser une catégorie sans la configurer, ce qui est un comportement identique à celui de Yii 1.1. Les messages pour cette catégorie proviendront d'une source de messages par défaut située dans le dossier `basePath` c.-à-d. `@app/messages` :

```
echo Yii::t('not_specified_category', 'message from unspecified category');
```

Le message sera chargé depuis `@app/messages/<LanguageCode>/not_specified_category.php`.

Traduction des messages d'un module

Si vous voulez traduire les messages d'un module et éviter d'avoir un unique fichier de traduction pour tous les messages, vous pouvez procéder comme suit :

```php
<?php

namespace app\modules\users;

use Yii;

class Module extends \yii\base\Module
{
    public $controllerNamespace = 'app\modules\users\controllers';

    public function init()
    {
        parent::init();
        $this->registerTranslations();
    }

    public function registerTranslations()
    {
        Yii::$app->i18n->translations['modules/users/*'] = [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/modules/users/messages',
            'fileMap' => [
                'modules/users/validation' => 'validation.php',
                'modules/users/form' => 'form.php',
                ...
            ],
        ];
    }

    public static function t($category, $message, $params = [], $language =
    null)
    {
        return Yii::t('modules/users/' . $category, $message, $params,
        $language);
    }

}
```

Dans l'exemple précédent, nous utilisons le caractère générique pour la correspondance puis nous filtrons chacune des catégories par fichier requis. Au lieu d'utiliser `fileMap`, vous pouvez utiliser la convention de mise en correspondance du fichier de même nom.

Désormais, vous pouvez utiliser `Module::t('validation', 'your custom validation message')` ou `Module::t('form', 'some form label')` directement.

Traduction des messages d'objets graphiques

La règle applicable aux modules présentée ci-dessus s'applique également aux objets graphiques, par exemple :

```php
<?php

namespace app\widgets\menu;

use yii\base\Widget;
use Yii;

class Menu extends Widget
{

    public function init()
    {
        parent::init();
        $this->registerTranslations();
    }

    public function registerTranslations()
    {
        $i18n = Yii::$app->i18n;
        $i18n->translations['widgets/menu/*'] = [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/widgets/menu/messages',
            'fileMap' => [
                'widgets/menu/messages' => 'messages.php',
            ],
        ];
    }

    public function run()
    {
        echo $this->render('index');
    }

    public static function t($category, $message, $params = [], $language =
    null)
    {
        return Yii::t('widgets/menu/' . $category, $message, $params,
        $language);
    }

}
```

Au lieu d'utiliser `fileMap`, vous pouvez utiliser la convention de mise en correspondance du fichier de même nom. Désormais, vous pouvez utiliser `Menu::t('messages', 'new messages {messages}', ['{messages}' => 10])` directement.

> **Note :** pour les objets graphiques, vous pouvez aussi utiliser les vues i18n, en y appliquant les mêmes règles que celles applicables aux contrôleurs.

Traduction des messages du framework

Yii est fourni avec les traductions par défaut des messages d'erreurs de validation et quelques autres chaînes. Ces messages sont tous dans la catégorie `yii`. Parfois, vous souhaitez corriger la traduction par défaut des messages du framework pour votre application. Pour le faire, configurez le composant d'application `i18n` comme indiqué ci-après :

```
'i18n' => [
    'translations' => [
        'yii' => [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/messages'
        ],
    ],
],
```

Vous pouvez désormais placer vos traductions corrigées dans `@app/messages/<language>/yii.php`.

Gestion des traductions manquantes

Même si la traduction n'est pas trouvée dans la source de traductions, Yii affiche le contenu du message demandé. Un tel comportement est très pratique tant que le message est une phrase valide. Néanmoins, quelques fois, cela ne suffit pas. Vous pouvez désirer faire quelque traitement de la situation, au moment où le message apparaît manquant. Vous pouvez utiliser pour cela l'événement `missingTranslation (traduction manquante)` de `yii\i18n\MessageSource`.

Par exemple, vous désirez peut-être marquer toutes les traductions manquantes par quelque chose de voyant, de manière à les repérer facilement dans la page. Vous devez d'abord configurer un gestionnaire d'événement. Cela peut se faire dans la configuration de l'application :

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
                'on missingTranslation' =>
                ['app\components\TranslationEventHandler',
                'handleMissingTranslation']
            ],
        ],
    ],
],
```

Vous devez ensuite implémenter votre gestionnaire d'événement :

```
<?php

namespace app\components;

use yii\i18n\MissingTranslationEvent;

class TranslationEventHandler
{
    public static function handleMissingTranslation(MissingTranslationEvent
    $event)
    {
        $event->translatedMessage = "@MISSING:
        {$event->category}.{$event->message} FOR LANGUAGE {$event->language}
        @";
    }
}
```

Si `yii\i18n\MissingTranslationEvent::$translatedMessage` est défini par le gestionnaire d'événement, il sera affiché en tant que résultat de la traduction.

> **Note :** chacune des sources de messages gère ses traductions manquantes séparément. Si vous avez recours à plusieurs sources de messages et que vous voulez qu'elles gèrent les messages manquants de la même manière, vous devez assigner le gestionnaire d'événement correspondant à chacune d'entre-elles.

Utilisation de la commande

Les traductions peuvent être stockées dans des `fichiers php`, des `fichiers .po` ou dans une `bases de données`. Reportez-vous aux classes spécifiques pour connaître les options supplémentaires.

En premier lieu, vous devez créer un fichier de configuration. Décidez de son emplacement et exécutez la commande suivante :

```
./yii message/config-template path/to/config.php
```

Ouvrez le fichier ainsi créé et ajustez-en les paramètres pour qu'ils répondent à vos besoins. Portez-une attention particulière à :

- `languages` : un tableau des langues dans lesquelles votre application doit être traduite ;
- `messagePath` : le chemin du dossier où doivent être placés les fichiers de messages, qui doit correspondre le paramètre `basePath` de `i18n` dans la configuration de l'application.

Vous pouvez également utiliser la commande './yii message/config' pour générer dynamiquement le fichier de configuration avec les options spécifiées via le ligne de commande. Par exemple, vous pouvez définir `languages` et `messagePath` comme indiqué ci-dessous :

```
./yii message/config --languages=de,ja --messagePath=messages
path/to/config.php
```

Pour connaître toutes les options utilisables, exécutez la commande :

```
./yii help message/config
```

Une fois que vous en avez terminé avec la configuration, vous pouvez finalement extraire vos messages par la commande :

```
./yii message path/to/config.php
```

Vous pouvez aussi utiliser des options pour changer dynamiquement les paramètres d'extraction.

Vous trouverez alors vos fichiers de traduction (si vous avez choisi les traductions basées sur des fichiers) dans votre dossier `messagePath`.

15.1.4 Traduction des vues

Plutôt que de traduire des textes de messages individuels, vous pouvez parfois désirer traduire le script d'une vue tout entier. Pour cela, contentez-vous de traduire la vue et de la sauvegarder dans un sous-dossier dont le nom est le code de la langue cible. Par exemple, si vous avez traduit le script de la vue `views/site/index.php` et que la langue cible est `fr-FR`, vous devez sauvegarder la traduction dans `views/site/fr-FR/index.php`. Désormais, à chaque fois que vous appellerez `yii\base\View::renderFile()` ou toute méthode qui invoque cette méthode (p. ex. `yii\base\Controller::render()`) pour rendre la vue, `views/site/index.php`, ce sera la vue `views/site/fr-FR/index.php` qui sera rendue à sa place.

> **Note :** si la `langue cible` est identique à la `langue source` la vue originale sera rendue sans tenir compte de l'existence de la vue traduite.

15.1.5 Formatage des dates et des nombres

Reportez-vous à la section Formatage des données pour les détails.

15.1.6 Configuration de l'environnement PHP

Yii utilise l'extension intl de PHP [16] pour fournir la plupart de ses fonctionnalités d'internationalisation, telles que le formatage des dates et des nombres de la classe `yii\i18n\Formatter` et le formatage des messages de la classe `yii\i18n\MessageFormatter`. Les deux classes fournissent un mécanisme de remplacement lorsque l'extension `intl` n'est pas installée. Néanmoins,

16. https://www.php.net/manual/fr/book.intl.php

l'implémentation du mécanisme de remplacement ne fonctionne bien que quand la langue cible est l'anglais. C'est pourquoi, il est fortement recommandé d'installer `intl` quand c'est nécessaire. L'extension intl de PHP [17] est basée sur la bibliothèque ICU [18] qui fournit la base de connaissances et les règles de formatage pour les différentes locales. Des versions différentes d'ICU peuvent conduire à des formatages différents des dates et des nombres. Pour être sûr que votre site Web donne les même résultats dans tous les environnements, il est recommandé d'installer la même version de l'extension `intl` (et par conséquent la même version d'ICU) dans tous les environnements.

Pour savoir quelle version d'ICU est utilisée par PHP, vous pouvez exécuter le script suivant, qui vous restitue la version de PHP et d'ICU en cours d'utilisation.

```
<?php
echo "PHP: " . PHP_VERSION . "\n";
echo "ICU: " . INTL_ICU_VERSION . "\n";
echo "ICU Data: " . INTL_ICU_DATA_VERSION . "\n";
```

Il est également recommandé d'utiliser une version d'ICU supérieure ou égale à 48. Cela garantit que toutes les fonctionnalités décrites dans ce document sont utilisables. Par exemple, une version d'ICU inférieure à 49 ne prend pas en charge la valeur à remplacer `#` dans les règles de pluralisation. Reportez-vous à `https://icu.unicode.org/download` pour obtenir une liste complète des versions d'ICU disponibles. Notez que le numérotage des versions a changé après la version 4.8 (p. ex., ICU 4.8, ICU 49, ICU 50, etc.)

En outre, les informations dans la base de donnée des fuseaux horaires fournie par la bibliothèque ICU peuvent être surannées. Reportez-vous au manuel d'ICU [19] pour les détails sur la manière de mettre la base de données des fuseaux horaires à jour. Bien que la base de données des fuseaux horaires d'ICU soit utilisée pour le formatage, celle de PHP peut aussi être d'actualité. Vous pouvez la mettre à jour en installant la dernière version du paquet `timezonedb` de pecl [20].

17. https://www.php.net/manual/fr/book.intl.php
18. https://icu.unicode.org/
19. https://unicode-org.github.io/icu/userguide/datetime/timezone/#updating-the-time-zone-data
20. https://pecl.php.net/package/timezonedb

Error : not existing file : tutorial-mailing.md

Error : not existing file : tutorial-performance-tuning.md

Error : not existing file : tutorial-shared-hosting.md

Error : not existing file : tutorial-template-engines.md

Chapitre 16

Widgets

Error : not existing file : bootstrap-widgets.md

Error : not existing file : jui-widgets.md

Chapitre 17

Assistants

17.1 Classes assistantes

Note : cette section est en cours de développement.

Yii procure de nombreuses classes qui vous aident à simplifier le code de tâches courantes, telles que la manipulation de chaînes de caractères ou de tableaux, la génération de code HTML, et ainsi de suite. Ces classes assistantes sont organisées dans l'espace de noms `yii\helpers` et sont toutes des classes statiques (ce qui signifie qu'elles ne contiennent que des propriétés et des méthodes statiques et ne doivent jamais être instanciées).

Vous utilisez une classe assistante en appelant directement une de ses méthodes statiques, comme ceci :

```
use yii\helpers\Html;

echo Html::encode('Test > test');
```

> **Note :** pour prendre en charge la personnalisation des classes assistantes, Yii éclate chacune des classes assistantes du noyau en deux classes : une classe de base (p. ex. `BaseArrayHelper`) et une classe concrète (p. ex. `ArrayHelper`). Lorsque vous utilisez une classe assistante, vous devez utiliser la version concrète uniquement et ne jamais utiliser la classe de base.

17.1.1 Classes assistantes du noyau

Les versions de Yii fournissent les classes assistantes du noyau suivantes :

— ArrayHelper
— Console
— FileHelper
— FormatConverter
— Html

— HtmlPurifier
— Imagine (provided by yii2-imagine extension)
— Inflector
— Json
— Markdown
— StringHelper
— Url
— VarDumper

17.1.2 Personnalisation des classes assistantes

Pour personnaliser une classe assistante du noyau (p. ex. `yii\helpers\ArrayHelper`), vous devez créer une nouvelle classe qui étend la classe de base correspondant à la classe assistante (p. ex. `yii\helpers\ArrayHelper`), y compris son espace de noms. Cette classe sera ensuite configurée pour remplacer l'implémentation originale de Yii.

L'exemple qui suit montre comment personnaliser la méthode `merge()` de la classe `yii\helpers\ArrayHelper` :

```
<?php

namespace yii\helpers;

class ArrayHelper extends BaseArrayHelper
{
    public static function merge($a, $b)
    {
        // votre implémentation personnalisée
    }
}
```

Sauvegardez votre classe dans un fichier nommé `ArrayHelper.php`. Le fichier peut se trouver dans n'importe quel dossier, par exemple, `@app/components`.

Ensuite, dans le script d'entrée de votre application, ajoutez la ligne de code suivante, après l'inclusion du fichier `yii.php` pour dire à la classe autoloader de Yii de charger votre classe personnalisée au lieu de la classe assistance originale de Yii.

```
Yii::$classMap['yii\helpers\ArrayHelper'] =
'@app/components/ArrayHelper.php';
```

Notez que la personnalisation d'une classe assistante n'est utile que si vous désirez changer le comportement d'une fonction existante de la classe assistante. Si vous désirez ajouter une fonction additionnelle à utiliser dans votre application, le mieux est de créer une classe assistante séparée pour cela.

17.2 Classe assistante ArrayHelper

En plus du jeu riche de fonctions de tableaux [1] qu'offre PHP, la classe assistante traitant les tableaux dans Yii fournit des méthodes statiques supplémentaires qui vous permettent de traiter les tableaux avec plus d'efficacité.

17.2.1 Obtention de valeurs

Récupérer des valeurs d'un tableau ou d'un objet ou une structure complexe écrits tous deux en PHP standard est un processus assez répétitif. Vous devez d'abord vérifier que la clé existe avec `isset`, puis si c'est le cas, vous récupérez la valeur associée, sinon il vous faut fournir une valeur par défaut :

```
class User
{
    public $name = 'Alex';
}

$array = [
    'foo' => [
        'bar' => new User(),
    ]
];

$value = isset($array['foo']['bar']->name) ? $array['foo']['bar']->name :
null;
```

Yii fournit une méthode très pratique pour faire cela :

```
$value = ArrayHelper::getValue($array, 'foo.bar.name');
```

Le premier argument de la méthode indique de quelle source nous voulons récupérer une valeur. Le deuxième spécifie comment récupérer la donnée. Il peut s'agir d'un des éléments suivants :

— Nom d'une clé de tableau ou de la propriété d'un objet de laquelle récupérer une valeur.
— Un jeu de noms de clé de tableau ou de propriétés d'objet séparées par des points, comme dans l'exemple que nous venons de présenter ci-dessus.
— Une fonction de rappel qui retourne une valeur.

Le fonction de rappel doit être la suivante :

```
$fullName = ArrayHelper::getValue($user, function ($user, $defaultValue) {
    return $user->firstName . ' ' . $user->lastName;
});
```

Le troisième argument facultatif est la valeur par défaut qui est `null` si on ne la spécifie pas. Il peut être utilisé comme ceci :

1. https://www.php.net/manual/fr/book.array.php

```php
$username = ArrayHelper::getValue($comment, 'user.username', 'Unknown');
```

Dans le cas où vous voulez récupérer la valeur tout en la retirant immédiatement du tableau, vous pouvez utiliser la méthode `remove` :

```php
$array = ['type' => 'A', 'options' => [1, 2]];
$type = ArrayHelper::remove($array, 'type');
```

Après exécution du code, `$array` contiendra `['options' => [1, 2]]` et `$type` sera `A`. Notez que contrairement à la méthode `getValue`, `remove` accepte seulement les noms de clé.

17.2.2 Tester l'existence des clés

`ArrayHelper::keyExists` fonctionne comme array_key_exists [2] sauf qu'elle prend également en charge la comparaison de clés insensible à la casse. Par exemple,

```php
$data1 = [
    'userName' => 'Alex',
];

$data2 = [
    'username' => 'Carsten',
];

if (!ArrayHelper::keyExists('username', $data1, false) ||
!ArrayHelper::keyExists('username', $data2, false)) {
    echo "Veuillez fournir un nom d'utilisateur (username).";
}
```

17.2.3 Récupération de colonnes

Il arrive souvent que vous ayez à récupérer une colonne de valeurs d'un tableau de lignes de données ou d'objets. Un exemple courant est l'obtention d'une liste d'identifiants.

```php
$array = [
    ['id' => '123', 'data' => 'abc'],
    ['id' => '345', 'data' => 'def'],
];
$ids = ArrayHelper::getColumn($array, 'id');
```

Le résultat sera `['123', '345']`.

Si des transformations supplémentaires sont nécessaires ou si la manière de récupérer les valeurs est complexe, le second argument peut être formulé sous forme de fonction anonyme :

```php
$result = ArrayHelper::getColumn($array, function ($element) {
    return $element['id'];
});
```

2. https://www.php.net/manual/fr/function.array-key-exists.php

17.2.4 Réindexation de tableaux

La méthode `index` peut être utilisées pour indexer un tableau selon une clé spécifiée. L'entrée doit être soit un tableau multidimensionnel, soit un tableau d'objets. `$key` peut être un nom de clé du sous-tableau, un nom de propriété d'objet ou une fonction anonyme qui doit retourner la valeur à utiliser comme clé.

L'attribut `$groups` est un tableau de clés qui est utilisé pour regrouper le tableau d'entrée en un ou plusieurs sous-tableaux basés sur les clés spécifiées.

Si l'argument `$key` ou sa valeur pour l'élément particulier est `null` alors que `$groups` n'est pas défini, l'élément du tableau est écarté. Autrement, si `$groups` est spécifié, l'élément du tableau est ajouté au tableau résultant sans aucune clé.

Par exemple :

```
$array = [
    ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
    ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
];
$result = ArrayHelper::index($array, 'id');
```

Le résultat est un tableau associatif, dans lequel la clé est la valeur de l'attribut `id` :

```
[
    '123' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    '345' => ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone']
    // Le second élément du tableau d'origine est écrasé par le dernier
    élément parce que les identifiants sont identiques.
]
```

Une fonction anonyme passée en tant que `$key`, conduit au même résultat :

```
$result = ArrayHelper::index($array, function ($element) {
    return $element['id'];
});
```

Passer `id` comme troisième argument regroupe `$array` par `id` :

```
$result = ArrayHelper::index($array, null, 'id');
```

Le résultat est un tableau multidimensionnel regroupé par `id` au premier niveau et non indexé au deuxième niveau :

```
[
    '123' => [
        ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
    ],
```

```
    '345' => [ // all elements with this index are present in the result
    array
        ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
        ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
    ]
]
```

Une fonction anonyme peut également être utilisée dans le tableau de regroupement :

```
$result = ArrayHelper::index($array, 'data', [function ($element) {
    return $element['id'];
}, 'device']);
```

Le résultat est un tableau multidimensionnel regroupé par `id` au premier niveau, par `device` au deuxième niveau et par `data` au troisième niveau :

```
[
    '123' => [
        'laptop' => [
            'abc' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
        ]
    ],
    '345' => [
        'tablet' => [
            'def' => ['id' => '345', 'data' => 'def', 'device' => 'tablet']
        ],
        'smartphone' => [
            'hgi' => ['id' => '345', 'data' => 'hgi', 'device' =>
            'smartphone']
        ]
    ]
]
```

17.2.5 Construction de tableaux de mise en correspondance

Afin de construire un tableau de mise en correspondance (paires clé-valeur) sur la base d'un tableau multidimensionnel ou d'un tableau d'objets, vous pouvez utiliser la méthode `map`. Les paramètres `$from` et `$to` spécifient les noms de clé ou les noms des propriétés pour construire le tableau de mise en correspondance. Le paramètre facultatif `$group` est un nom de clé ou de propriété qui permet de regrouper les éléments du tableau au premier niveau. Par exemple :

```
$array = [
    ['id' => '123', 'name' => 'aaa', 'class' => 'x'],
    ['id' => '124', 'name' => 'bbb', 'class' => 'x'],
    ['id' => '345', 'name' => 'ccc', 'class' => 'y'],
];
```

```php
$result = ArrayHelper::map($array, 'id', 'name');
// le résultat est :
// [
//     '123' => 'aaa',
//     '124' => 'bbb',
//     '345' => 'ccc',
// ]

$result = ArrayHelper::map($array, 'id', 'name', 'class');
// le résultat est :
// [
//     'x' => [
//         '123' => 'aaa',
//         '124' => 'bbb',
//     ],
//     'y' => [
//         '345' => 'ccc',
//     ],
// ]
```

17.2.6 Tri multidimensionnel

La méthode `multisort` facilite le tri d'un tableau d'objets ou de tableaux imbriqués selon une ou plusieurs clés. Par exemple :

```php
$data = [
    ['age' => 30, 'name' => 'Alexander'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 19, 'name' => 'Barney'],
];
ArrayHelper::multisort($data, ['age', 'name'], [SORT_ASC, SORT_DESC]);
```

Après le tri, `data` contient ce qui suit :

```php
[
    ['age' => 19, 'name' => 'Barney'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 30, 'name' => 'Alexander'],
];
```

Le deuxième argument, qui spécifie les clés de tri peut être une chaîne de caractères si la clé est unique, un tableau dans le cas de clés multiples, ou une fonction anonyme telle que celle qui suit :

```php
ArrayHelper::multisort($data, function($item) {
    return isset($item['age']) ? ['age', 'name'] : 'name';
});
```

Le troisième argument précise la direction. Dans le cas d'un tri selon une clé unique, il s'agit soit de `SORT_ASC`, soit de `SORT_DESC`. Si le tri se fait selon des valeurs multiples, vous pouvez préciser des directions de tri différentes pour chacune des clés en présentant ces directions sous forme de tableau.

Le dernier argument est une option de tri de PHP qui peut prendre les mêmes valeurs que celles acceptées par la fonction sort() [3] de PHP.

17.2.7 Détection des types de tableau

Il est pratique de savoir si un tableau est indexé ou associatif. Voici un exemple :

```
// aucune clé spécifiée
$indexed = ['Qiang', 'Paul'];
echo ArrayHelper::isIndexed($indexed);

// toutes les clés sont des chaînes de caractères
$associative = ['framework' => 'Yii', 'version' => '2.0'];
echo ArrayHelper::isAssociative($associative);
```

17.2.8 Encodage et décodage de valeurs HTML

Afin d'encoder ou décoder des caractères spéciaux dans un tableau de chaînes de caractères en/depuis des entités HTML, vous pouvez utiliser les fonctions suivantes :

```
$encoded = ArrayHelper::htmlEncode($data);
$decoded = ArrayHelper::htmlDecode($data);
```

Seules les valeurs sont encodées par défaut. En passant un deuxième argument comme `false` vous pouvez également encoder les clés d'un tableau. L'encodage utilise le jeu de caractères de l'application et on peut le changer via un troisième argument.

17.2.9 Fusion de tableaux

La fonction `ArrayHelper::merge()` vous permet de fusionner deux, ou plus, tableaux en un seul de manière récursive. Si chacun des tableaux possède un élément avec la même chaîne clé valeur, le dernier écrase le premier (ce qui est un fonctionnement différent de array_merge_recursive() [4]). La fusion récursive est entreprise si les deux tableaux possèdent un élément de type tableau avec la même clé. Pour des éléments dont la clé est un entier, les éléments du deuxième tableau sont ajoutés aux éléments du premier tableau. Vous pouvez utiliser l'objet `yii\helpers\UnsetArrayValue` pour supprimer la valeur du premier tableau ou `yii\helpers\ReplaceArrayValue` pour forcer le remplacement de la première valeur au lieu de la fusion récursive.

Par exemple :

3. https://www.php.net/manual/fr/function.sort.php
4. https://www.php.net/manual/fr/function.array-merge-recursive.php

```php
$array1 = [
    'name' => 'Yii',
    'version' => '1.1',
    'ids' => [
        1,
    ],
    'validDomains' => [
        'example.com',
        'www.example.com',
    ],
    'emails' => [
        'admin' => 'admin@example.com',
        'dev' => 'dev@example.com',
    ],
];

$array2 = [
    'version' => '2.0',
    'ids' => [
        2,
    ],
    'validDomains' => new \yii\helpers\ReplaceArrayValue([
        'yiiframework.com',
        'www.yiiframework.com',
    ]),
    'emails' => [
        'dev' => new \yii\helpers\UnsetArrayValue(),
    ],
];

$result = ArrayHelper::merge($array1, $array2);
```

Le résultat est :

```php
[
    'name' => 'Yii',
    'version' => '2.0',
    'ids' => [
        1,
        2,
    ],
    'validDomains' => [
        'yiiframework.com',
        'www.yiiframework.com',
    ],
    'emails' => [
        'admin' => 'admin@example.com',
    ],
]
```

17.2.10 Conversion d'objets en tableaux

Il arrive souvent que vous ayez besoin de convertir un objet, ou un tableau d'objets, en tableau. Le cas le plus courant est la conversion de modèles

d'enregistrements actifs afin de servir des tableaux de données via une API REST ou pour un autre usage. Le code suivant peut alors être utilisé :

```
$posts = Post::find()->limit(10)->all();
$data = ArrayHelper::toArray($posts, [
    'app\models\Post' => [
        'id',
        'title',
        // the key name in array result => property name
        'createTime' => 'created_at',
        // the key name in array result => anonymous function
        'length' => function ($post) {
            return strlen($post->content);
        },
    ],
]);
```

Le premier argument contient les données à convertir. Dans notre cas, nous convertissons un modèle d'enregistrements actifs `Post`.

The second argument est un tableau de mise en correspondance de conversions par classe. Nous définissons une mise en correspondance pour le modèle `Post`. Chaque tableau de mise en correspondance contient un jeu de mise en correspondance. Chaque mise en correspondance peut être :

— Un nom de champ à inclure tel quel.
— Une paire clé-valeur dans laquelle la clé est donnée sous forme de chaîne de caractères et la valeur sous forme du nom de la colonne dont on doit prendre la valeur.
— Une paire clé-valeur dans laquelle la clé est donnée sous forme de chaîne de caractères et la valeur sous forme de fonction de rappel qui la retourne.

Le résultat de la conversion ci-dessus pour un modèle unique est :

```
[
    'id' => 123,
    'title' => 'test',
    'createTime' => '2013-01-01 12:00AM',
    'length' => 301,
]
```

Il est possible de fournir une manière par défaut de convertir un objet en tableau pour une classe spécifique en implémentant l'interface `Arrayable` dans cette classe.

17.2.11 Test de l'appartenance à un tableau

Souvent, vous devez savoir si un élément se trouve dans un tableau ou si un jeu d'éléments est un sous-ensemble d'un autre. Bien que PHP offre la fonction `in_array()`, cette dernière ne prend pas en charge les sous-ensembles ou les objets `\Traversable`.

Pour faciliter ce genre de tests, `yii\helpers\ArrayHelper` fournit les méthodes `isIn()` et `isSubset()` avec la même signature que in_array() [5].

```
// true
ArrayHelper::isIn('a', ['a']);
// true
ArrayHelper::isIn('a', new ArrayObject(['a']));

// true
ArrayHelper::isSubset(new ArrayObject(['a', 'c']), new ArrayObject(['a',
'b', 'c']));
```

17.3 Classe assistante Html

Toutes les applications Web génèrent un grand nombre de balises HTML. Si le code HTML est statique, il peut être créé efficacement sous forme de mélange de code PHP et de code HTML dans un seul fichier [6], mais lorsqu'il est généré dynamiquement, cela commence à être compliqué à gérer sans une aide supplémentaire. Yii fournit une telle aide sous la forme de la classe assistante Html, qui offre un jeu de méthodes statiques pour manipuler les balises Html les plus courantes, leurs options et leur contenu.

> **Note :** si votre code HTML est presque statique, il vaut mieux utiliser HTML directement. Il n'est pas nécessaire d'envelopper tout dans des appels aux méthodes de la classe assistante Html.

17.3.1 Les bases

Comme la construction de code HTML dynamique en concaténant des chaînes de caractère peut très vite tourner à la confusion, Yii fournit un jeu de méthodes pour manipuler les options de balises et construire des balises s'appuyant sur ces options.

Génération de balises

Le code pour générer une balise ressemble à ceci :

```
<?= Html::tag('p', Html::encode($user->name), ['class' => 'username']) ?>
```

Le premier argument est le nom de la balise. Le deuxième est le contenu qui apparaît entre l'ouverture de la balise et sa fermeture. Notez que nous utilisons `Html::encode` – c'est parce que le contenu n'est pas encodé automatiquement pour permetre l'utilisation de HTML quand c'est nécessaire. Le troisième est un tableau d'options HTML ou, en d'autres mots, les attributs

5. https://www.php.net/manual/fr/function.in-array.php
6. https://www.php.net/manual/fr/language.basic-syntax.phpmode.php

de la balise. Dans ce tableau, la clé est le nom de l'attribut (comme `class`, `href` ou `target`) et la valeur est sa valeur.

Le code ci-dessus génère le code HTML suivant :

```
<p class="username">samdark</p>
```

Dans le cas où vous avez simplement besoin d'ouvrir ou de fermer la balise, vous pouvez utiliser les méthodes `Html::beginTag()` et `Html::endTag()`.

Des options sont utilisées dans de nombreuses méthodes de la classe assistante Html et de nombreux composants graphiques (widgets). Dans tous ces cas, il y a quelques manipulations supplémentaires à connaître :

- Si une valeur est `null`, l'attribut correspondant n'est pas rendu.
- Les attributs du type booléen sont traités comme des attributs booléens [7].
- Les valeurs des attributs sont encodés HTML à l'aide de la méthode `Html::encode()`.
- Si la valeur d'un attribut est un tableau, il est géré comme suit :
  - Si l'attribut est un attribut de donnée tel que listé dans `yii\helpers\Html::$dataAttributes`, tel que `data` ou `ng`, une liste d'attributs est rendue, un pour chacun des élément dans le tableau de valeurs. Par exemple, `'data' => ['id' => 1, 'name' => 'yii']` génère `data-id="1" data-name="yii"` ; et `'data' => ['params' => ['id' => 1, 'name' => 'yii'], 'status' => 'ok']` génère `data-params='{"id":1,"name":"yii"}' data-status="ok"`. Notez que dans le dernier exemple le format JSON est utilisé pour rendre le sous-tableau.
  - Si l'attribut n'est PAS un attribut de donnée, la valeur est encodée JSON. Par exemple, `['params' => ['id' => 1, 'name' => 'yii']` génère `params='{"id":1,"name":"yii"}'`.

Formation des classes et des styles CSS

Lors de la construction des options pour des balises HTML, nous démarrons souvent avec des valeurs par défaut qu'il faut modifier. Afin d'ajouter ou de retirer une classe, vous pouvez utiliser ce qui suit :

```
$options = ['class' => 'btn btn-default'];

if ($type === 'success') {
    Html::removeCssClass($options, 'btn-default');
    Html::addCssClass($options, 'btn-success');
}

echo Html::tag('div', 'Pwede na', $options);

// si la valeur de $type est 'success' le rendu sera
// <div class="btn btn-success">Pwede na</div>
```

7. https://html.spec.whatwg.org/multipage/common-microsyntaxes.html#boolean-attributes

Vous pouvez spécifier de multiples classe CSS en utilisant le tableau de styles également :

```
$options = ['class' => ['btn', 'btn-default']];

echo Html::tag('div', 'Save', $options);
// rend '<div class="btn btn-default">Save</div>'
```

Vous pouvez aussi utiliser le tableau de styles pour ajouter ou retirer des classes :

```
$options = ['class' => 'btn'];

if ($type === 'success') {
    Html::addCssClass($options, ['btn-success', 'btn-lg']);
}

echo Html::tag('div', 'Save', $options);
// rend '<div class="btn btn-success btn-lg">Save</div>'
```

`Html::addCssClass()` empêche la duplication, vous n'avez donc pas à vous préoccuper de savoir si une classe apparaît deux fois :

```
$options = ['class' => 'btn btn-default'];

Html::addCssClass($options, 'btn-default'); // class 'btn-default' is
already present

echo Html::tag('div', 'Save', $options);
// rend '<div class="btn btn-default">Save</div>'
```

Si l'option classe CSS est spécifiée en utilisant le tableau de styles, vous pouvez utiliser une clé nommée pour indiquer le but logique de la classe. Dans ce cas, une classe utilisant la même clé dans le tableau de styles passé à `Html::addClass()` est ignorée :

```
$options = [
    'class' => [
        'btn',
        'theme' => 'btn-default',
    ]
];

Html::addCssClass($options, ['theme' => 'btn-success']); // la clé 'theme'
est déjà utilisée

echo Html::tag('div', 'Save', $options);
// rend '<div class="btn btn-default">Save</div>'
```

Les styles CSS peuvent être définis d'une façon similaire en utilisant l'attribut `style` :

```php
$options = ['style' => ['width' => '100px', 'height' => '100px']];

// donne style="width: 100px; height: 200px; position: absolute;"
Html::addCssStyle($options, 'height: 200px; position: absolute;');

// gives style="position: absolute;"
Html::removeCssStyle($options, ['width', 'height']);
```

Lors de l'utilisation de `addCssStyle()`, vous pouvez spécifier soit un tableau de paires clé-valeur qui correspond aux propriétés CSS noms et valeurs, soit une chaîne de caractères telle que `width: 100px; height: 200px;`. Ces formats peuvent être convertis de l'un en l'autre en utilisant les méthodes `cssStyleFromArray()` et `cssStyleToArray()`. La méthode `removeCssStyle()` accepte un tableau de propriétés à retirer. S'il s'agit d'une propriété unique, elle peut être spécifiée sous forme de chaîne de caractères.

Encodage et décodage du contenu

Pour que le contenu puisse être affiché en HTML de manière propre et en toute sécurité, les caractères spéciaux du contenu doivent être encodés. En PHP, cela s'obtient avec htmlspecialchars [8] et htmlspecialchars_decode [9]. Le problème rencontré en utilisant ces méthodes directement est que vous devez spécifier l'encodage et des options supplémentaires tout le temps. Comme ces options restent toujours les mêmes et que l'encodage doit correspondre à celui de l'application pour éviter les problèmes de sécurité, Yii fournit deux méthodes compactes et faciles à utiliser :

```php
$userName = Html::encode($user->name);
echo $userName;

$decodedUserName = Html::decode($userName);
```

17.3.2 Formulaires

Manipuler des formulaires dans le code HTML est tout à fait répétitif et sujet à erreurs. À cause de cela, il existe un groupe de méthodes pour aider à les manipuler.

> Note : envisagez d'utiliser `ActiveForm` dans le cas où vous avez affaire à des modèles et que ces derniers doivent être validés.

Création de formulaires

Les formulaires peut être ouverts avec la méthode `beginForm()` comme ceci :

8. https://www.php.net/manual/fr/function.htmlspecialchars.php
9. https://www.php.net/manual/fr/function.htmlspecialchars-decode.php

```
<?= Html::beginForm(['order/update', 'id' => $id], 'post', ['enctype' =>
'multipart/form-data']) ?>
```

Le premier argument est l'URL à laquelle le formulaire sera soumis. Il peut être spécifié sous la forme d'une route Yii et de paramètres acceptés par `Url::to()`. Le deuxième est la méthode à utiliser. `post` est la méthode par défaut. Le troisième est un tableau d'options pour la balise form. Dans ce cas, nous modifions l'encodage des données du formulaire dans la requête POST en `multipart/form-data`, ce qui est requis pour envoyer des fichiers.

La fermeture du formulaire se fait simplement par :

```
<?= Html::endForm() ?>
```

Boutons

Pour générer des boutons, vous pouvez utiliser le code suivant :

```
<?= Html::button('Pressez-mo!', ['class' => 'teaser']) ?>
<?= Html::submitButton('Envoyer', ['class' => 'submit']) ?>
<?= Html::resetButton('Ré-initialiser', ['class' => 'reset']) ?>
```

Le premier argument pour les trois méthodes est l'intitulé du bouton, le deuxième est un tableau d'options. L'intitulé n'est pas encodé, mais si vous affichez des données en provenance de l'utilisateur, encodez les avec `Html::encode()`.

Champs d'entrée

Il y a deux groupes de méthodes d'entrée de données. Celles qui commencent par `active`, est qui sont appelées entrées actives, et celles qui ne commencent pas par ce mot. Les entrées actives prennent leurs données dans le modèle à partir des attributs spécifiés, tandis que pour les entrées régulières, les données sont spécifiées directement.

Les méthodes les plus génériques sont :

```
type, nom de l'entrée, valeur de l'entrée, options
<?= Html::input('text', 'username', $user->name, ['class' => $username]) ?>

type, modèle, nom de l'attribut du modèle, options
<?= Html::activeInput('text', $user, 'name', ['class' => $username]) ?>
```

Si vous connaissez le type de l'entrée à l'avance, il est plus commode d'utiliser les méthodes raccourcis :

- `yii\helpers\Html::buttonInput()`
- `yii\helpers\Html::submitInput()`
- `yii\helpers\Html::resetInput()`
- `yii\helpers\Html::textInput()`, `yii\helpers\Html::activeTextInput()`

— `yii\helpers\Html::hiddenInput()`, `yii\helpers\Html::activeHiddenInput()`
— `yii\helpers\Html::passwordInput()` / `yii\helpers\Html::activePasswordInput()`
— `yii\helpers\Html::fileInput()`, `yii\helpers\Html::activeFileInput()`
— `yii\helpers\Html::textarea()`, `yii\helpers\Html::activeTextarea()`

Les listes radio et les boîtes à cocher sont un peu différentes en matière de signature de méthode :

```
<?= Html::radio('agree', true, ['label' => 'I agree']);
<?= Html::activeRadio($model, 'agree', ['class' => 'agreement'])

<?= Html::checkbox('agree', true, ['label' => 'I agree']);
<?= Html::activeCheckbox($model, 'agree', ['class' => 'agreement'])
```

Les listes déroulantes et les boîtes listes peuvent être rendues comme suit :

```
<?= Html::dropDownList('list', $currentUserId, ArrayHelper::map($userModels,
'id', 'name')) ?>
<?= Html::activeDropDownList($users, 'id', ArrayHelper::map($userModels,
'id', 'name')) ?>

<?= Html::listBox('list', $currentUserId, ArrayHelper::map($userModels,
'id', 'name')) ?>
<?= Html::activeListBox($users, 'id', ArrayHelper::map($userModels, 'id',
'name')) ?>
```

Le premier argument est le nom de l'entrée, le deuxième est la valeur sélectionnée actuelle et le troisième est un tableau de paires clé-valeur, dans lequel la clé est la valeur d'entrée dans la liste et la valeur est l'étiquette qui correspond à cette valeur dans la liste.

Si vous désirez que des choix multiples soient sélectionnables, vous pouvez utiliser la liste à sélection multiples (checkbox list) :

```
<?= Html::checkboxList('roles', [16, 42], ArrayHelper::map($roleModels,
'id', 'name')) ?>
<?= Html::activeCheckboxList($user, 'role', ArrayHelper::map($roleModels,
'id', 'name')) ?>
```

Sinon utilisez la liste radio :

```
<?= Html::radioList('roles', [16, 42], ArrayHelper::map($roleModels, 'id',
'name')) ?>
<?= Html::activeRadioList($user, 'role', ArrayHelper::map($roleModels,
'id', 'name')) ?>
```

Étiquettes et erreurs

Comme pour les entrées, il existe deux méthodes pour générer les étiquettes de formulaire. Celles pour les entrées « actives » qui prennent leurs étiquettes dans le modèle, et celles « non actives » qui sont étiquetées directement :

```
<?= Html::label('User name', 'username', ['class' => 'label username']) ?>
<?= Html::activeLabel($user, 'username', ['class' => 'label username']) ?>
```

Pour afficher les erreurs de formulaire à partir d'un modèle ou sous forme de résumé pour un modèle, vous pouvez utiliser :

```
<?= Html::errorSummary($posts, ['class' => 'errors']) ?>
```

Pour afficher une erreur individuellement :

```
<?= Html::error($post, 'title', ['class' => 'error']) ?>
```

Nom et valeur des entrées

Il existe deux méthodes pour obtenir des noms, des identifiants et des valeurs pour des champs d'entrée basés sur un modèle. Elles sont essentiellement utilisées en interne, mais peuvent être pratiques quelques fois :

```
// Post[title]
echo Html::getInputName($post, 'title');

// post-title
echo Html::getInputId($post, 'title');

// my first post
echo Html::getAttributeValue($post, 'title');

// $post->authors[0]
echo Html::getAttributeValue($post, '[0]authors[0]');
```

Dans ce qui précède, le premier argument est le modèle, tandis que le deuxième est l'expression d'attribut. Dans sa forme la plus simple, l'expression est juste un nom d'attribut, mais il peut aussi s'agir d'un nom d'attribut préfixé et-ou suffixé par des index de tableau, ce qui est essentiellement le cas pour des entrées tabulaires :

- `[0]content` est utilisé dans des entrées de données tabulaires pour représenter l'attribut `content` pour le premier modèle des entrées tabulaires ;
- `dates[0]` représente le premier élément du tableau de l'attribut `dates` ;
- `[0]dates[0]` représente le premier élément du tableau de l'attribut `dates` pour le premier modèle des entrées tabulaires.

Afin d'obtenir le nom de l'attribut sans suffixe ou préfixe, vous pouvez utiliser ce qui suit :

```
// dates
echo Html::getAttributeName('dates[0]');
```

17.3.3 Styles et scripts

Il existe deux méthodes pour générer les balises enveloppes des styles et des scripts :

```
<?= Html::style('.danger { color: #f00; }') ?>

Produit

<style>.danger { color: #f00; }</style>


<?= Html::script('alert("Hello!");', ['defer' => true]);

Produit

<script defer>alert("Hello!");</script>
```

Si vous désirez utiliser utiliser un style externe d'un fichier CSS :

```
<?= Html::cssFile('@web/css/ie5.css', ['condition' => 'IE 5']) ?>

génère

<!--[if IE 5]>
    <link href="https://example.com/css/ie5.css" />
<![endif]-->
```

Le premier argument est l'URL. Le deuxième est un tableau d'options. En plus des options normales, vous pouvez spécifier :

— `condition` pour envelopper `<link` dans des commentaires conditionnels avec la condition spécifiée. Nous espérons que vous n'aurez jamais besoin de commentaires conditionnels ;

— `noscript` peut être défini à `true` pour envelopper `<link` dans une balise `<noscript>` de façon à ce qu'elle soit incluse seulement si le navigateur ne prend pas en charge JavaScript ou si l'utilisateur l'a désactivé.

Pour lier un fichier JavaScript :

```
<?= Html::jsFile('@web/js/main.js') ?>
```

Se passe comme avec CSS, le premier argument spécifie l'URL du fichier à inclure. Les options sont passées via le deuxième argument. Dans les options vous pouvez spécifier `condition` de la même manière que dans les options pour un fichier CSS (méthode `cssFile`).

17.3.4 Hyperliens

Il y a une méthode commode pour générer les hyperliens :

```
<?= Html::a('Profile', ['user/view', 'id' => $id], ['class' =>
'profile-link']) ?>
```

Le premier argument est le titre. Il n'est pas encodé, mais si vous utilisez des données entrées par l'utilisateur, vous devez les encoder avec `Html::encode()`. Le deuxième argument est ce qui se retrouvera dans l'attribut `href` de la balise `<a>`.

Voir Url : :to() pour les détails sur les valeurs acceptées. Le troisième argument est un tableau pour les attributs de la balise.

Si vous devez générer des liens `mailto`, vous pouvez utiliser le code suivant :

```
<?= Html::mailto('Contact us', 'admin@example.com') ?>
```

17.3.5 Images

Pour générer une balise image, utilisez le code suivant :

```
<?= Html::img('@web/images/logo.png', ['alt' => 'My logo']) ?>
```

qui génère

```
<img src="https://example.com/images/logo.png" alt="My logo" />
```

En plus des alias, le premier argument accepte les routes, les paramètres et les URL, tout comme Url : :to().

17.3.6 Listes

Les listes non ordonnées peuvent être générées comme suit :

```
<?= Html::ul($posts, ['item' => function($item, $index) {
    return Html::tag(
        'li',
        $this->render('post', ['item' => $item]),
        ['class' => 'post']
    );
}]) ?>
```

Pour une liste ordonnée, utilisez plutôt `Html::ol()`.

17.4 Classe assistante Url

La classe assistante Url fournit un jeu de méthodes statiques pour gérer les URL.

17.4.1 Obtenir des URL communes

Vous pouvez utiliser deux méthodes pour obtenir des URL communes : l'URL de la page d'accueil et l'URL de base de la requête courante. Pour obtenir l'URL de la page d'accueil, utilisez ce qui suit :

```php
$relativeHomeUrl = Url::home();
$absoluteHomeUrl = Url::home(true);
$httpsAbsoluteHomeUrl = Url::home('https');
```

Si aucun paramètre n'est passé, l'URL générée est relative. Vous pouvez passer `true` pour obtenir une URL absolue pour le schéma courant ou spécifier un schéma explicitement (`https`, `http`).

Pour obtenir l'URL de base de la requête courante utilisez ceci :

```php
$relativeBaseUrl = Url::base();
$absoluteBaseUrl = Url::base(true);
$httpsAbsoluteBaseUrl = Url::base('https');
```

L'unique paramètre de la méthode fonctionne comme pour `Url::home()`.

17.4.2 Création d'URL

En vue de créer une URL pour une route donnée, utilisez la méthode `Url::toRoute()`. La méthode utilise `yii\web\UrlManager` pour créer une URL :

```php
$url = Url::toRoute(['product/view', 'id' => 42]);
```

Vous pouvez spécifier la route sous forme de chaîne de caractère, p. ex. `site/index`. Vous pouvez également utiliser un tableau si vous désirez spécifier des paramètres de requête supplémentaires pour l'URL créée. Le format du tableau doit être :

```php
// génère : /index.php?r=site%2Findex&param1=value1&param2=value2
['site/index', 'param1' => 'value1', 'param2' => 'value2']
```

Si vous voulez créer une URL avec une ancre, vous pouvez utiliser le format de tableau avec un paramètre `#`. Par exemple :

```php
// génère: /index.php?r=site%2Findex&param1=value1#name
['site/index', 'param1' => 'value1', '#' => 'name']
```

Une route peut être ,soit absolue, soit relative. Une route absolue commence par une barre oblique de division (p. ex. `/site/index`) tandis que route relative commence sans ce caractère (p. ex. `site/index` ou `index`). Une route relative peut être convertie en une route absolue en utilisant une des règles suivantes :

- Si la route est une chaîne de caractères vide, la `route` est utilisée ;
- Si la route ne contient aucune barre oblique de division (p. ex. `index`), elle est considérée être un identifiant d'action dans le contrôleur courant et sera préfixée par l'identifiant du contrôleur (`yii\web\Controller::$uniqueId`) ;
- Si la route ne commence pas par une barre oblique de division (p. ex. `site/index`), elle est considérée être une route relative au module courant et sera préfixée par l'identifiant du module (`uniqueId`).

Depuis la version 2.0.2, vous pouvez spécifier une route sous forme d'alias. Si c'est le cas, l'alias sera d'abord converti en la route réelle puis transformé en une route absolue en respectant les règles ci-dessus.

Voici quelques exemple d'utilisation de cette méthode :

```
// /index.php?r=site%2Findex
echo Url::toRoute('site/index');

// /index.php?r=site%2Findex&src=ref1#name
echo Url::toRoute(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post%2Fedit&id=100     assume the alias "@postEdit" is
defined as "post/edit"
echo Url::toRoute(['@postEdit', 'id' => 100]);

// https://www.example.com/index.php?r=site%2Findex
echo Url::toRoute('site/index', true);

// https://www.example.com/index.php?r=site%2Findex
echo Url::toRoute('site/index', 'https');
```

Il existe une autre méthode `Url::to()` très similaire à `toRoute()`. La seule différence est que cette méthode requiert la spécification d'une route sous forme de tableau seulement. Si une chaîne de caractères est données, elle est traitée comme une URL.

Le premier argument peut être :

— un tableau : `toRoute()` sera appelée pour générer l'URL. Par exemple : `['site/index']`, `['post/index', 'page' => 2]`. Reportez-vous à la méthode `toRoute()` pour plus de détails sur la manière de spécifier une route.

— une chaîne de caractères commençant par `@` : elle est traitée commme un alias, et la chaine aliasée correspondante est retournée ;

— une chaîne de caractères vide : l'URL couramment requise est retournée ;

— une chaîne de caractères normale : elle est retournée telle que.

Lorsque `$scheme` est spécifié (soit une chaîne de caractères, soit `true`), une URL absolue avec l'information hôte tirée de `yii\web\UrlManager::$hostInfo`) est retournée. Si`$url` est déjà une URL absolue, son schéma est remplacé par celui qui est spécifié.

Voici quelques exemples d'utilisation :

```
// /index.php?r=site%2Findex
echo Url::to(['site/index']);

// /index.php?r=site%2Findex&src=ref1#name
echo Url::to(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post%2Fedit&id=100     assume the alias "@postEdit" is
defined as "post/edit"
echo Url::to(['@postEdit', 'id' => 100]);
```

```php
// l'URL couramment requise
echo Url::to();

// /images/logo.gif
echo Url::to('@web/images/logo.gif');

// images/logo.gif
echo Url::to('images/logo.gif');

// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', true);

// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', 'https');
```

Depuis la version 2.0.3, vous pouvez utiliser `yii\helpers\Url::current()` pour créer une URL basée sur la route couramment requise et sur les paramètres de la méthode GET. Vous pouvez modifier ou retirer quelques uns des paramètres GET et en ajouter d'autres en passant le paramètre `$params` à la méthode. Par exemple :

```php
// suppose que $_GET = ['id' => 123, 'src' => 'google'],et que la route
courante est "post/view"

// /index.php?r=post%2Fview&id=123&src=google
echo Url::current();

// /index.php?r=post%2Fview&id=123
echo Url::current(['src' => null]);
// /index.php?r=post%2Fview&id=100&src=google
echo Url::current(['id' => 100]);
```

17.4.3 Se souvenir d'URL

Il y a des cas dans lesquels vous avez besoin de mémoriser une URL et ensuite de l'utiliser durant le traitement d'une des requêtes séquentielles. Cela peut être fait comme suit :

```php
// se souvenir de l'URL courante
Url::remember();

// Se souvenir de l'URL spécifiée. Voir Url::to() pour le format des
arguments.
Url::remember(['product/view', 'id' => 42]);

// Se souvenir de  l'URL spécifiée avec un nom
Url::remember(['product/view', 'id' => 42], 'product');
```

Dans la prochaine requête, vous pouvez récupérer l'URL mémorisée comme ceci :

```
$url = Url::previous();
$productUrl = Url::previous('product');
```

17.4.4 Vérification des URL relatives

Pour savoir si une URL est relative, c.-à-d. n'a pas de partie « hôte », vous pouvez utiliser le code suivant :

```
$isRelative = Url::isRelative('test/it');
```

Error : not existing file : helper-security.md