

Guia Definitivo para Yii 2.0

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

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

Português brasileiro translation provided by:

Alcir Monteiro,
Alan Michel Willms Quinot,
Davidson Alencar,
Gustavo G. Andrade,
Jan Silva,
Lucas Barros,
Raphael de Almeida,
Sidney da Silva Lins,
Wanderson Bragança,
Anthony Tesche

This tutorial is released under the [Terms of Yii Documentation](#).

Copyright 2014 Yii Software LLC. All Rights Reserved.

Sumário

1	Introdução	1
1.1	O que é o Yii	1
1.2	Atualizando a partir da Versão 1.1	3
2	Primeiros Passos	15
2.1	O que você precisa saber	15
2.2	Instalando o Yii	16
2.3	Executando Aplicações	23
2.4	Dizendo “Olá!”	27
2.5	Trabalhando com Formulários	30
2.6	Trabalhando com Bancos de Dados	37
2.7	Gerando Código com Gii	43
2.8	Seguindo em Frente	50
3	Estrutura da Aplicação	53
3.1	Visão Geral	53
3.2	Scripts de Entrada	54
3.3	Aplicações	56
3.4	Componentes de Aplicação	68
3.5	Controllers (Controladores)	71
3.6	Models (Modelos)	80
3.7	Visões (Views)	91
3.8	Módulos	107
3.9	Filtros	112
3.10	Widgets	120
3.11	Assets	124
3.12	Extensões	142
4	Tratando Requisições	153
4.1	Visão Geral	153
4.2	Inicialização (Bootstrapping)	154
4.3	Roteamento e Criação de URL	155
4.4	Requisições	170

4.5	Sessões e Cookies	175
4.6	Tratamento de Erros	182
4.7	Log	186
5	Conceitos Chave	195
5.1	Componentes	195
5.2	Propriedades	197
5.3	Eventos	199
5.4	Behaviors (Comportamentos)	205
5.5	Configurações	212
5.6	Aliases (Apelidos)	217
5.7	Autoloading de Classes	220
5.8	Service Locator	222
5.9	Container de Injeção de Dependência	224
6	Trabalhando com Banco de Dados	233
6.1	Query Builder (Construtor de Consulta)	235
6.2	Active Record	247
6.3	Migrações de Dados (Migrations)	276
7	Coletando Dados de Usuários	287
8	Exibindo Dados	295
8.1	Paginação	297
8.2	Ordenação	298
8.3	Data Providers (Provedores de Dados)	300
8.4	Temas	310
9	Segurança	313
9.1	Autenticação	315
9.2	Autorização	319
10	Cache	339
10.1	Cache	339
10.2	Cache de Dados	339
10.3	Cache de Fragmentos	348
10.4	Cache de Página	351
10.5	Cache HTTP	352
11	Web Services RESTful	357
11.1	Introdução	357
11.2	Recursos	361
11.3	Controllers (Controladores)	366
11.4	Roteamento	369
11.5	Formatando Respostas	371

11.6 Autenticação	374
11.7 Limitador de Acesso	377
11.8 Versionamento	378
11.9 Tratamento de Erros	381
12 Ferramentas de Desenvolvimento	385
13 Testes	387
13.1 Testes	387
14 Tópicos Especiais	395
14.1 Validadores Nativos	398
14.2 Yii e Docker	410
14.3 Ambiente de Hospedagem Compartilhada	416
14.4 Trabalhando com Códigos de Terceiros	420
15 Widgets	425
16 Helpers - Funções Auxiliares	427
16.1 Helpers	427
16.2 URL Helper	431

Capítulo 1

Introdução

1.1 O que é o Yii

Yii é um framework PHP de alta performance baseado em componentes para desenvolvimento rápido de aplicações web modernas. O nome Yii (pronunciado *ii*) significa “simples e evolutivo” em chinês. Ele também pode ser considerado um acrônimo de **Yes It Is** (*Sim, ele é!*)

1.1.1 Yii é melhor para que tipo de aplicações?

Yii é um framework de programação web genérico, o que significa que ele pode ser usado para o desenvolvimento de todo tipo de aplicações web usando PHP. Por causa de sua arquitetura baseada em componentes e suporte sofisticado a caching, ele é especialmente adequado para o desenvolvimento de aplicações de larga escala como portais, fóruns, sistemas de gerenciamento de conteúdo (CMS), projetos de e-commerce, Web services RESTful e assim por diante.

1.1.2 Como o Yii se Compara a Outros Frameworks?

Se já estiver familiarizado com um outro framework, você pode gostar de saber como o Yii se compara:

- Como a maioria dos frameworks PHP, o Yii implementa o padrão de arquitetura MVC (Modelo-Visão-Controlador) e promove a organização do código baseada nesse padrão.
- Yii tem a filosofia de que o código deveria ser escrito de uma maneira simples, porém elegante. O Yii nunca vai tentar exagerar no projeto só para seguir estritamente algum padrão de projeto.
- Yii é um framework completo fornecendo muitas funcionalidades comprovadas e prontas para o uso, tais como: construtores de consultas (query builders) e ActiveRecord tanto para bancos de dados relacionais

quanto para NoSQL; suporte ao desenvolvimento de APIs RESTful; suporte a caching de múltiplas camadas; e mais.

- Yii é extremamente extensível. Você pode personalizá-lo ou substituir quase todas as partes do código central (core). Você também pode tirar vantagem de sua sólida arquitetura de extensões para utilizar ou desenvolver extensões que podem ser redistribuídas.
- Alta performance é sempre um objetivo principal do Yii.

Yii não é um show de um homem só, ele é apoiado por uma forte equipe de desenvolvedores do código central (core)¹ bem como por uma ampla comunidade de profissionais constantemente contribuindo com o desenvolvimento do Yii. A equipe de desenvolvedores do Yii acompanha de perto às últimas tendências do desenvolvimento Web e as melhores práticas e funcionalidades encontradas em outros frameworks e projetos. As mais relevantes e melhores práticas e características encontradas em outros lugares são incorporadas regularmente no core do framework e expostas via interfaces simples e elegantes.

1.1.3 Versões do Yii

Atualmente, o Yii tem duas versões principais disponíveis: a 1.1 e a 2.0. A Versão 1.1 é a antiga geração e agora está em modo de manutenção. A versão 2.0 é uma reescrita completa do Yii, adotando as tecnologias e protocolos mais recentes, incluindo Composer, PSR, namespaces, traits, e assim por diante. A versão 2.0 representa a geração atual do framework e receberá os nossos esforços principais de desenvolvimento nos próximos anos. Este guia trata principalmente da versão 2.0.

1.1.4 Requisitos e Pré-requisitos

Yii 2.0 requer PHP 7.3.0 ou superior. Você pode encontrar requisitos mais detalhados para recursos específicos executando o verificador de requisitos (requirement checker) incluído em cada lançamento do Yii.

Utilizar o Yii requer conhecimentos básicos sobre programação orientada a objetos (OOP), uma vez que o Yii é um framework puramente OOP. O Yii 2.0 também utiliza as funcionalidades mais recentes do PHP, tais como namespaces² e traits³. Compreender esses conceitos lhe ajudará a entender mais facilmente o Yii 2.0.

¹<https://www.yiiframework.com/team>

²https://www.php.net/manual/pt_BR/language.namespaces.php

³https://www.php.net/manual/pt_BR/language.oop5.traits.php

1.2 Atualizando a partir da Versão 1.1

Existem muitas diferenças entre as versões 1.1 e 2.0 do Yii, uma vez que o framework foi completamente reescrito na 2.0. Por causa disso, atualizar a partir da versão 1.1 não é tão trivial quanto atualizar de versões menores. Neste guia você encontrará as principais diferenças entre as duas versões.

Se você nunca usou o Yii 1.1 antes, você pode pular com segurança esta seção e ir diretamente para “[Instalando o Yii](#)”.

Por favor, note que o Yii 2.0 introduz outras novas funcionalidades além das que são abordadas neste resumo. Recomenda-se fortemente que você leia o guia definitivo por completo para aprender todas elas. É possível que algumas funcionalidades que antes você tinha de desenvolver por conta própria agora façam parte do código principal.

1.2.1 Instalação

O Yii 2.0 utiliza plenamente o Composer⁴, o gerenciador de pacotes PHP. Tanto a instalação do núcleo do framework quanto das extensões são feitas através do Composer. Por favor, consulte a seção [Instalando o Yii](#) para aprender como instalar o Yii 2.0. Se você quer criar novas extensões ou tornar compatíveis as suas extensões existentes do 1.1 com o 2.0, por favor consulte a seção [Criando Extensões](#) do guia.

1.2.2 Requisitos do PHP

O Yii 2.0 requer o PHP 5.4 ou superior, que é uma versão de grande melhoria sobre a versão 5.2, que era exigida pelo Yii 1.1. Como resultado, existem muitas diferenças na linguagem às quais você deve dar a devida atenção. Segue abaixo um resumo das principais mudanças do PHP:

- Namespaces⁵.
- Funções anônimas⁶.
- A sintaxe curta de arrays [...elementos...] é utilizada ao invés de `array(...elementos...)`.
- Tags curtas de `echo <?=` são usadas nos arquivos de view. É seguro utilizá-las a partir do PHP 5.4.
- Classes e interfaces da SPL⁷.
- Late Static Bindings⁸.
- Date e Time⁹.

⁴<https://getcomposer.org/>

⁵https://www.php.net/manual/pt_BR/language.namespaces.php

⁶https://www.php.net/manual/pt_BR/functions.anonymous.php

⁷https://www.php.net/manual/pt_BR/book.spl.php

⁸https://www.php.net/manual/pt_BR/language.oop5.late-static-bindings.php

⁹https://www.php.net/manual/pt_BR/book.datetime.php

- Traits¹⁰.
- intl¹¹. O Yii 2.0 utiliza a extensão intl do PHP para suportar as funcionalidades de internacionalização.

1.2.3 Namespace

A mudança mais óbvia no Yii 2.0 é o uso de namespaces. Praticamente todas as classes do *core* possuem namespace, por exemplo, `yii\web\Request`. O prefixo “C” não é mais utilizado nos nomes de classes. O esquema de nomenclatura agora segue a estrutura de diretórios. Por exemplo, `yii\web\Request` indica que o arquivo da classe correspondente é `web/Request.php` sob a pasta do Yii Framework.

(Você pode utilizar qualquer classe do *core* sem explicitamente incluir o arquivo dessa classe, graças ao carregador de classes do Yii).

1.2.4 Component e Object

O Yii 2.0 divide a classe `CComponent` do 1.1 em duas classes: `yii\base\BaseObject` e `yii\base\Component`. A classe `BaseObject` é uma classe base leve que permite a definição das [propriedades de objetos](#) via getters e setters. A classe `Component` estende de `BaseObject` e suporta [eventos](#) e [comportamentos](#) (behaviors).

Se a sua classe não precisa de eventos nem de comportamentos, você deveria considerar utilizar `BaseObject` como classe base. Esse geralmente é o caso de classes que representam estruturas básicas de dados.

1.2.5 Configuração de Objetos

A classe `BaseObject` introduz uma maneira uniforme de configurar objetos. Qualquer classe descendente de `BaseObject` deveria declarar seu construtor (se necessário) da seguinte maneira, para que ela seja configurada adequadamente:

```
class MinhaClasse extends \yii\base\BaseObject
{
    public function __construct($param1, $param2, $config = [])
    {
        // ... inicialização antes da configuração ser aplicada

        parent::__construct($config);
    }

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

¹⁰https://www.php.net/manual/pt_BR/language.oop5.traits.php

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

```
        // ... inicialização depois da configuração ser aplicada
    }
}
```

No código acima, o último parâmetro do construtor deve receber um array de configuração que contém pares de nome-valor para a inicialização das propriedades no final do construtor. Você pode sobrescrever o método `init()` para fazer o trabalho de inicialização que deve ser feito após a configuração ter sido aplicada.

Seguindo esta convenção, você poderá criar e configurar novos objetos usando um array de configuração:

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

Mais detalhes sobre configurações podem ser encontrados na seção de [Configurações](#).

1.2.6 Eventos

No Yii 1 os eventos eram criados definindo-se um método `on-alguma-coisa` (por exemplo, `onBeforeSave`). No Yii 2 você pode usar qualquer nome de evento. Você dispara um evento chamando o método `trigger()`:

```
$evento = new \yii\base\Event;
$componente->trigger($nomeDoEvento, $evento);
```

Para anexar um ouvinte (handler) a um evento, use o método `on()`:

```
$componente->on($nomeDoEvento, $handler);
// Para desanexar o handler, utilize:
// $componente->off($nomeDoEvento, $handler);
```

Há muitas melhorias nas funcionalidades de evento. Para mais detalhes, por favor, consulte a seção [Eventos](#).

1.2.7 Path Aliases

O Yii 2.0 expande o uso de *path aliases* (apelidos de caminhos) tanto para caminhos de arquivos e diretórios como para URLs. Agora ele requer que um nome de alias comece com o caractere `@` para diferenciar entre aliases e caminhos e URLs normais de arquivos e diretórios. Por exemplo, o alias `@yii` se refere ao diretório de instalação do Yii. Os path aliases são suportados na maior parte do código do core do Yii. Por exemplo, o método `yii\caching`

`\FileCache::$cachePath` pode receber tanto um path alias quanto um caminho de diretório normal.

Um path alias também está intimamente relacionado a um namespace de classe. É recomendado que um path alias seja definido para cada namespace raiz, desta forma permitindo que você use o auto-carregamento de classes do Yii sem qualquer configuração adicional. Por exemplo, como `@yii` se refere ao diretório de instalação do Yii, uma classe como `yii\web\Request` pode ser carregada automaticamente. Se você utilizar uma biblioteca de terceiros, tal como o Zend Framework, você pode definir um path alias `@zend` que se refere ao diretório de instalação desse framework. Uma vez que você tenha feito isso, o Yii também poderá carregar automaticamente qualquer classe nessa biblioteca do Zend Framework.

Você pode encontrar mais informações sobre *path aliases* na seção [Aliases](#).

1.2.8 Views (Visões)

A mudança mais significativa das views no Yii 2 é que a variável especial `$this` em uma view não se refere mais ao controller ou widget atual. Ao invés disso, `$this` agora se refere a um objeto ***view**, um novo conceito introduzido no 2.0. O objeto *view** é do tipo `yii\web\View` e representa a parte da visão do padrão MVC. Se você quiser acessar o controller ou o widget em uma visão, você pode utilizar `$this->context`.

Para renderizar uma view parcial (partial view) dentro de outra view, você usa `$this->render()`, e não `$this->renderPartial()`. Agora a chamada de `render` também precisa ser explicitamente impressa com *echo*, uma vez que o método `render()` retorna o resultado da renderização ao invés de exibi-lo diretamente. Por exemplo:

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

Além de utilizar o PHP como linguagem de template principal, o Yii 2.0 também é equipado com suporte oficial a duas populares engines de template: Smarty e Twig. A engine de template do Prado não é mais suportada. Para utilizar essas engines de template, você precisa configurar o componente de aplicação `view` definindo a propriedade `View::$renderers`. Por favor consulte a seção [Template Engines](#) para mais detalhes.

1.2.9 Models (Modelos)

O Yii 2.0 usa o `yii\base\Model` como base, semelhante à `CModel` no 1.1. A classe `CFormModel` foi removida inteiramente. Ao invés dela, no Yii 2 você deve estender a classe `yii\base\Model` para criar uma classe de model de formulário.

O Yii 2.0 introduz um novo método chamado `scenarios()` para declarar os cenários suportados, para indicar sob qual cenário um atributo precisa ser validado ou pode ser considerado safe (seguro) ou não, etc. Por exemplo:

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

No código acima, dois cenários são declarados: `backend` e `frontend`. Para o cenário `backend`, os atributos `email` e `cargo` são seguros (`safe`) e podem ser atribuídos em massa. Para o cenário `frontend`, `email` pode ser atribuído em massa enquanto `cargo` não. Tanto `email` quanto `role` devem ser validados utilizando-se *rules* (regras).

O método `rules()` ainda é usado para declarar regras de validação. Perceba que devido à introdução do método `scenarios()`, não existe mais o validador `unsafe` (inseguro).

Na maioria dos casos, você não precisa sobrescrever `scenarios()` se o método `rules()` especifica completamente os cenários que existirão, e se não houver necessidade para declarar atributos `unsafe`.

Para aprender mais sobre `models`, por favor consulte a seção `Models` (Modelos).

1.2.10 Controllers (Controladores)

O Yii 2.0 utiliza a `yii\web\Controller` como classe base dos controllers de maneira semelhante à `CWebController` no Yii 1.1. A `yii\base\Action` é a classe base para classes de actions (ações).

O impacto mais óbvio destas mudanças em seu código é que uma action de um controller deve sempre retornar o conteúdo que você quer renderizar ao invés de dar *echo* nele:

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

Por favor, consulte a seção `Controllers` (Controladores) para mais detalhes.

1.2.11 Widgets

O Yii 2.0 usa `yii\base\Widget` como a classe base dos widgets, de maneira semelhante à `CWidget` no Yii 1.1.

Para obter um melhor suporte ao framework nas IDEs, o Yii 2.0 introduz uma nova sintaxe para utilização de widgets. Os métodos estáticos `begin()`, `end()` e `widget()` foram introduzidos, para serem utilizados do seguinte modo:

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

// Note que você tem que dar um "echo" no resultado para exibi-lo
echo Menu::widget(['items' => $items]);

// Passando um array para inicializar as propriedades do objeto
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge']],
]);
... campos do formulário aqui ...
ActiveForm::end();
```

Por favor, consulte a seção [Widgets](#) para mais detalhes.

1.2.12 Temas

Os temas funcionam de maneira completamente diferente no 2.0. Agora eles se baseiam em um mecanismo de mapeamento de caminhos que mapeia um caminho de arquivo de view fonte a um caminho de arquivo de view com o tema. Por exemplo, se o mapa de caminho de um tema é `['/web/views' => '/web/themes/basic']`, então a versão com tema deste arquivo de view `/web/views/site/index.php` será `/web/themes/basic/site/index.php`. Por esse motivo, os temas agora podem ser aplicados a qualquer arquivo de view, até mesmo uma view renderizada fora do contexto de um controller ou widget.

Além disso, não há mais um componente `CThemeManager`. Em vez disso, `theme` é uma propriedade configurável do componente `view` da aplicação.

Por favor, consulte a seção [Temas](#) para mais detalhes.

1.2.13 Aplicações de Console

As aplicações de console agora são organizadas como controllers assim como as aplicações web. Os controllers de console devem estender de `yii\console\Controller`, de maneira semelhante à `CConsoleCommand` no 1.1.

Para rodar um comando do console, use `yii <rota>`, onde `<rota>` representa a rota de um controller (por exemplo, `sitemap/index`). Argumentos anônimos adicionais são passados como parâmetros à action correspondente no controller, enquanto argumentos com nome são “convertidos” de acordo com as declarações em `yii\console\Controller::options()`.

O Yii 2.0 suporta a geração automática de informação de ajuda do comando a partir de blocos de comentários.

Por favor consulte a seção [Comandos de Console](#) para mais detalhes.

1.2.14 I18N

O Yii 2.0 remove os formatadores de data e número embutidos em favor do módulo intl do PECL do PHP¹².

A tradução de mensagens agora é realizada pelo componente `i18n` da aplicação. Este componente gerencia um conjunto de fontes de mensagens, o que permite a você usar diferentes fontes de mensagens baseadas em categorias de mensagens.

Por favor consulte a seção [Internacionalização](#) para mais detalhes.

1.2.15 Action Filters (Filtros de Ação)

Agora os filtros de ação (action filters) são implementados via comportamentos (behaviors). Para definir um novo filtro personalizado, estenda de `yii\base\ActionFilter`. Para usar um filtro, anexe a classe do filtro ao controller como um behavior. Por exemplo, para usar o filtro `yii\filters\AccessControl`, você teria o seguinte código em um controller:

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

Por favor, consulte a seção [Filtragem](#) para mais detalhes.

1.2.16 Assets

O Yii 2.0 introduz um novo conceito chamado de *asset bundle* (pacote de recursos estáticos) que substitui o conceito de script packages (pacotes de script) encontrado no Yii 1.1.

Um *asset bundle* é uma coleção de arquivos de assets (por exemplo, arquivos JavaScript, arquivos CSS, arquivos de imagens, etc.) dentro de um diretório. Cada *asset bundle* é representado por uma classe que estende `yii`

¹²<https://pecl.php.net/package/intl>

`\web\AssetBundle`. Ao registrar um *asset bundle* via `yii\web\AssetBundle::register()`, você torna os assets deste pacote acessíveis via Web. Ao contrário do Yii 1, a página que registra o *bundle* automaticamente conterá as referências aos arquivos JavaScript e CSS especificados naquele *bundle*.

Por favor consulte a seção Gerenciando Assets para mais detalhes.

1.2.17 Helpers - Classes Auxiliares

O Yii 2.0 introduz muitas classes auxiliares (helpers) estáticas comumente usadas, incluindo:

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

Por favor, consulte a seção [Visão Geral](#) dos helpers para mais detalhes.

1.2.18 Forms

O Yii 2.0 introduz o conceito de campos (*fields*) para a construção de formulários usando `yii\widgets\ActiveForm`. Um *field* é um container consistindo de um *label*, um *input*, uma mensagem de erro, e/ou um texto de ajuda. Um *field* é representado como um objeto `ActiveField`. Usando *fields* você pode construir um formulário de maneira mais limpa do que antes:

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

Por favor, consulte a seção [Criando um Formulário](#) para mais detalhes.

1.2.19 Query Builder (Construtor de Consultas)

No 1.1, a construção de consultas estava espalhada por diversas classes, incluindo a `CDbCommand`, a `CDbCriteria` e a `CDbCommandBuilder`. O Yii 2.0 representa uma consulta do banco de dados em termos de um objeto `Query` que pode ser convertido em uma instrução SQL com a ajuda do `QueryBuilder` que está por trás das cortinas. Por exemplo:

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



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

E o melhor de tudo, estes métodos de construção de consultas também podem ser utilizados ao trabalhar com o [Active Record](#).

Por favor, consulte a seção [Query Builder](#) para mais detalhes.

1.2.20 Active Record

O Yii 2.0 introduz várias mudanças ao [Active Record](#). As duas mais óbvias envolvem a construção de consultas simples e o tratamento de consultas relacionais.

A classe `CDbCriteria` do 1.1 foi substituída pela `yii\db\ActiveQuery` do Yii 2. Essa classe estende de `yii\db\Query`, e assim herda todos os métodos de construção de consultas. Você chama `yii\db\ActiveRecord::find()` para começar a construir uma consulta:

```
// Para obter todos os clientes *ativos* e ordená-los pelo ID:  
$customers = Cliente::find()  
    ->where(['status' => $ativo])  
    ->orderBy('id')  
    ->all();
```

Para declarar um relacionamento, simplesmente defina um método getter que retorne um objeto `ActiveQuery`. O nome da propriedade definida pelo getter representa o nome do relacionamento. Por exemplo, o código a seguir declara um relacionamento `pedidos` (no 1.1, você teria que declarar as relações em um local central, `relations()`):

```
class Cliente extends \yii\db\ActiveRecord  
{  
    public function getPedidos()  
    {  
        return $this->hasMany('Pedido', ['id_cliente' => 'id']);  
    }  
}
```

Agora você pode usar `$cliente->pedidos` para acessar os pedidos de um cliente a partir da tabela relacionada. Você também pode usar o código a seguir para realizar uma consulta relacional imediata (*on-the-fly*) com uma condição personalizada:

```
$pedidos = $cliente->getPedidos()->andWhere('status=1')->all();
```

Ao fazer o eager loading (carregamento antecipado) de um relacionamento, o Yii 2.0 faz isso de maneira diferente do 1.1. Em particular, no 1.1 uma

consulta JOIN seria criada para selecionar tanto o registro primário quanto os de relacionamentos. No Yii 2.0, duas instruções SQL são executadas sem usar JOIN: a primeira instrução retorna os registros primários e a segunda retorna os registros relacionados por filtrar pelas chaves primárias dos registros primários.

Em vez de retornar objetos `ActiveRecord`, você pode encadear o método `asArray()` ao construir uma consulta para retornar um grande número de registros. Isso fará com que o resultado da consulta retorne como arrays, o que pode reduzir significativamente o tempo de CPU e memória necessários para um grande número de registros. Por exemplo,

```
$clientes = Cliente::find()->asArray()->all();
```

Outra mudança é que você não pode mais definir valores padrão de atributos através de propriedades públicas. Se você precisar disso, você deve defini-las no método `init` na classe do seu registro.

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

Havia alguns problemas ao sobrescrever o construtor de uma classe `ActiveRecord` no 1.1. Estes não ocorrem mais na versão 2.0. Perceba que ao adicionar parâmetros ao construtor você pode ter que sobrescrever o método `yii\db\ActiveRecord::instantiate()`.

Existem muitas outras mudanças e melhorias no Active Record. Por favor, consulte a seção [Active Record](#) para mais detalhes.

1.2.21 Comportamentos (Behaviors) do Active Record

No Yii 2, removemos a classe base de behaviors `CActiveRecordBehavior`. Se você quer criar um Active Record Behavior, você terá que estender diretamente de `yii\base\Behavior`. Se a classe behavior class precisa responder a alguns eventos da classe que a possui, você deve sobrescrever o método `events()` conforme a seguir:

```
namespace app\components;

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

class MeuComportamento extends Behavior
{
    // ...

    public function events()
```

```

{
    return [
        ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
    ];
}

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

```

1.2.22 User e IdentityInterface

A classe `CWebUser` do 1.1 foi substituída pela `yii\web\User` e não há mais a classe `CUserIdentity`. Em vez disso, você deve implementar a interface `yii\web\IdentityInterface` que é muito mais simples de usar. O template avançado de projetos fornece um exemplo de como fazer isso.

Por favor, consulte as seções [Autenticação](#), [Autorização](#) e [Template Avançado de Projetos](#) para mais detalhes.

1.2.23 Gerenciamento de URLs

O gerenciamento de URLs no Yii 2 é semelhante ao do 1.1. Uma grande melhoria é que o gerenciamento de URLs agora suporta parâmetros opcionais. Por exemplo, se você tiver uma regra declarada como a seguir, ela vai corresponder tanto a `post/popular` quanto a `post/1/popular`. No 1.1, você teria que usar duas regras para fazer isso.

```

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

```

Por favor, consulte a seção [Roteamento e Criação de URL](#) para mais detalhes.

Uma importante mudança nas convenções de nomes para rotas é que actions e controllers com nomes em estilo camel case agora, quando referenciados em rotas, são convertidos para minúsculas separando cada palavra com um hífen. Por exemplo, o ID de controller `GestaoDeClientesController` deve ser referenciado em uma rota como `gestao-de-clientes`.

Veja as seções [IDs de Controllers](#) e [IDs de Actions](#) para mais detalhes.

1.2.24 Utilizando o Yii 1.1 e o 2.x juntos

Se você tem código legado do Yii 1.1 que você quer utilizar com o Yii 2.0, por favor, consulte a seção [Usando Yii 1.1 e 2.0 juntos](#).

Capítulo 2

Primeiros Passos

2.1 O que você precisa saber

A curva de aprendizado no Yii não é tão íngreme como em outros frameworks PHP mas, ainda assim, há algumas coisas que você devia aprender antes de começar.

2.1.1 PHP

Yii é um framework PHP. Portanto, certifique-se de ler e entender a referência da linguagem¹. Quando estiver desenvolvendo com Yii, você estará escrevendo código orientado a objetos, então, certifique-se de que está familiarizado tanto com Classes e Objetos² como com namespaces³.

2.1.2 Programação orientada a objetos

É necessário ter conhecimentos básicos de programação orientada a objetos. Se você não está familiarizado com esse tipo de programação, acesse um dos muitos tutoriais disponíveis, como este do tuts+⁴.

Note que, quanto mais complicado for seu projeto ou aplicação, mais você precisará de conceitos avançados de POO (Programação Orientada a Objetos) para ser bem sucedido em tratar essa complexidade.

2.1.3 Linha de comando and composer

Yii usa extensivamente o gerenciador de pacotes mais utilizado do PHP, o Composer⁵, então certifique-se de ler e entender seu guia⁶. Se você não tem

¹https://www.php.net/manual/pt_BR/langref.php

²https://www.php.net/manual/pt_BR/language.oop5.basic.php

³https://www.php.net/manual/pt_BR/language.namespaces.php

⁴<https://code.tutsplus.com/tutorials/object-oriented-php-for-beginners--net-12762>

⁵<https://getcomposer.org/>

⁶<https://getcomposer.org/doc/01-basic-usage.md>

familiaridade com a linha de comando é hora de começar a experimentar. Quando tiver aprendido o básico, nunca mais vai querer trabalhar sem ela.

2.2 Instalando o Yii

Você pode instalar o Yii de duas maneiras: usando o gerenciador de pacotes Composer⁷ ou baixando um arquivo compactado. O primeiro modo é o preferido, já que permite que você instale novas *extensões* ou atualize o Yii simplesmente executando um único comando.

A instalação padrão do Yii resulta no download e instalação tanto do framework quanto de um template de projetos. Um template de projeto é um projeto funcional do Yii que implementa alguns recursos básicos, tais como: autenticação, formulário de contato, etc. Este código é organizado de uma forma recomendada. Portanto, ele pode servir como ponto de partida para seus projetos.

Nesta e nas próximas seções, vamos descrever como instalar o *Template Básico de Projetos* do Yii e como implementar novas funcionalidades sobre este template. O Yii fornece ainda outro template chamado *Template Avançado de Projetos*⁸ que é melhor usado em um ambiente de desenvolvimento em equipe e para desenvolver aplicações com múltiplas camadas.

Info: O Template Básico de Projetos é adequado para o desenvolvimento de cerca de 90 Projetos principalmente em como o seu código é organizado. Se você é novo no Yii, recomendamos fortemente escolher o Template Básico de Projetos pela sua simplicidade e por manter suficientes funcionalidades.

2.2.1 Instalando via Composer

Instalando o Composer

Se você ainda não tem o Composer instalado, você pode instalá-lo seguindo as instruções em getcomposer.org⁹. No Linux e no Mac OS X, você executará os seguintes comandos:

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

No Windows, você baixará e executará o *Composer-Setup.exe*¹⁰.

⁷<https://getcomposer.org/>

⁸<https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-pt-BR/README.md>

⁹<https://getcomposer.org/download/>

¹⁰<https://getcomposer.org/Composer-Setup.exe>

Por favor, consulte a seção de Resolução de Problemas do Composer¹¹ se você encontrar dificuldades. Se você é novo no assunto, nós também recomendamos que leia pelo menos a seção Uso Básico¹² na documentação do Composer.

Neste guia, todos os comandos do composer assumem que você o tem instalado globalmente¹³ de modo que ele seja acessível através do comando `composer`. Se em vez disso estiver usando o `composer.phar` no diretório local, você tem que ajustar os comandos de exemplo de acordo.

Se você já tem o Composer instalado, certifique-se de usar uma versão atualizada. Você pode atualizar o Composer executando o comando `composer self-update`.

Note: Durante a instalação do Yii, o Composer precisará solicitar muitas informações da API do Github. A quantidade de solicitações depende do número de dependências que sua aplicação possui e pode extrapolar a **taxa limite da API do Github**. Se você atingir esse limite, o Composer pode pedir a você suas credenciais de login para obter um token de acesso à API Github. Em conexões rápidas você pode atingir esse limite antes que o Composer consiga lidar com a situação, então, recomendamos configurar um token de acesso antes de instalar o Yii. Por favor, consulte a documentação do Composer sobre tokens da API Github¹⁴ para instruções de como fazer isso.

Instalando o Yii

Com o Composer instalado, você pode instalar o Yii executando o seguinte comando em um diretório acessível pela Web:

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

Isto vai instalar a versão estável mais recente do Yii em um diretório chamado `basico`. Você pode especificar um nome de diretório diferente se quiser.

Info: Se o comando `composer create-project` falhar, você pode consultar a seção de Resolução de Problemas na documentação do Composer¹⁵ para verificar erros comuns. Quando tiver corrigido o erro, você pode continuar a instalação abortada por executar o comando `composer update` dentro do diretório `basico`.

¹¹<https://getcomposer.org/doc/articles/troubleshooting.md>

¹²<https://getcomposer.org/doc/01-basic-usage.md>

¹³<https://getcomposer.org/doc/00-intro.md#globally>

¹⁴<https://getcomposer.org/doc/articles/troubleshooting.md#api-rate-limit-and-oauth-tokens>

¹⁵<https://getcomposer.org/doc/articles/troubleshooting.md>

Tip: Se em disso você quiser instalar a versão em desenvolvimento mais recente do Yii, use o comando a seguir que adiciona uma opção de estabilidade¹⁶:

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

Note que a versão do Yii em desenvolvimento não deve ser usada em produção visto que pode quebrar seu código funcional.

2.2.2 Instalando a partir de um Arquivo Compactado

A instalação do Yii a partir de um arquivo compactado envolve três passos:

1. Baixe o arquivo compactado em yiiframework.com¹⁷.
2. Descompacte o arquivo baixado em um diretório acessível pela Web.
3. Modifique o arquivo `config/web.php` informando uma chave secreta no item de configuração `cookieValidationKey` (isto é feito automaticamente se você instalar o Yii pelo Composer):

```
`php // !!! Informe a chave secreta no item a seguir (se estiver vazio) -  
isto é requerido para a validação do cookie 'cookieValidationKey' =>  
'enter your secret key here',`
```

2.2.3 Outras Opções de Instalação

As instruções de instalação acima mostram como instalar o Yii, que também cria uma aplicação Web básica que funciona imediatamente sem qualquer configuração ou modificação (*out of the box*). Esta abordagem é um bom ponto de partida para a maioria dos projetos, seja ele pequeno ou grande. É especialmente adequado se você acabou de começar a aprender Yii.

No entanto, existem outras opções de instalação disponíveis:

- Se você só quer instalar o núcleo (*core*) do framework e gostaria de construir uma aplicação inteira do zero, você pode seguir as instruções em [Construindo uma Aplicação a Partir do Zero](#).
- Se você quiser começar com uma aplicação mais sofisticada, mais adequada ao ambiente de desenvolvimento em equipe, você pode considerar instalar o Template Avançado de Projetos¹⁸.

¹⁶<https://getcomposer.org/doc/04-schema.md#minimum-stability>

¹⁷<https://www.yiiframework.com/download/>

¹⁸<https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-pt-BR/README.md>

2.2.4 Instalando Recursos Estáticos (Assets)

Yii utiliza os pacotes Bower¹⁹ e/ou NPM²⁰ para a instalação das bibliotecas de recursos estáticos (CSS and JavaScript). Ele usa composer para obter essas bibliotecas, permitindo que versões de pacotes PHP, CSS e Javascript possam ser definidas/instaladas ao mesmo tempo. Isto é possível por usar ou asset-packagist.org²¹ ou [composer asset plugin](https://github.com/fxpiao/composer-asset-plugin)²². Por favor, consulta a [documentação sobre Assets](#) para mais detalhes.

Você pode querer gerenciar assets através de clientes nativos do Bower ou NPM, pode querer utilizar CDNs ou até evitar completamente a instalação de recursos estáticos. Para evitar que recursos estáticos sejam instalados via Composer, adicione o seguinte código ao seu `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: caso a instalação de recursos estáticos via Composer seja evitada, caberá a você instalar e resolver conflitos de versão ao instalar recursos estáticos (assets). Esteja preparado para possíveis inconsistências entre arquivos de recursos estáticos de diferentes extensões.

2.2.5 Verificando a Instalação

Após a instalação ser concluída, você pode tanto configurar seu servidor web (veja na próxima seção) como usar o servidor web embutido do PHP²³ executando o seguinte comando de console no diretório `web`:

```
php yii serve
```

Note: Por padrão o servidor HTTP vai ouvir na porta 8080. Contudo, se essa porta já estiver em uso ou se você pretende servir múltiplas aplicações desta forma, você pode querer especificar qual porta será usada. Para isso, basta adicionar o argumento `--port`:

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

¹⁹<https://bower.io/>

²⁰<https://www.npmjs.com/>

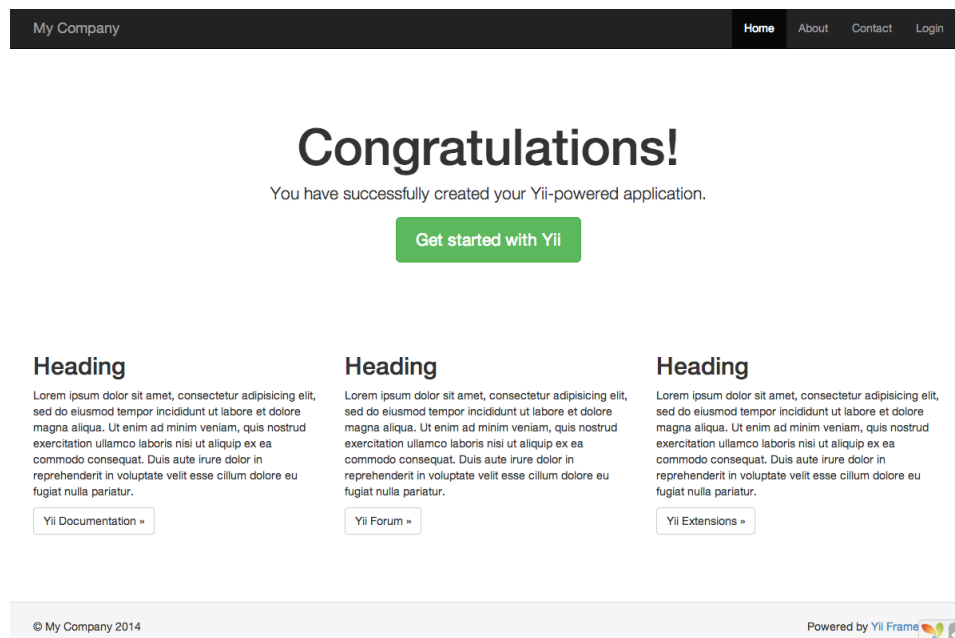
²¹<https://asset-packagist.org>

²²<https://github.com/fxpiao/composer-asset-plugin>

²³https://www.php.net/manual/pt_BR/features.commandline.webserver.php

Você pode usar seu navegador para acessar a aplicação instalada por meio da seguinte URL:

`http://localhost:8080/`



Você deverá ver a página de parabenização acima em seu navegador. Se não a vir, por favor, verifique se sua instalação PHP satisfaz os requisitos do Yii. Você pode verificar se os requisitos mínimos são atingidos usando uma das seguintes abordagens:

- Copiar `/requirements.php` para `/web/requirements.php` e então usar um navegador para acessá-lo por meio da URL `http://localhost/requirements.php`
- Executar os seguintes comandos:

```
cd basico
php requirements.php
```

Você deve configurar sua instalação PHP de forma a atingir os requisitos mínimos do Yii. A versão mínima do PHP que você deve ter é a 5.4. Mas o ideal seria utilizar a versão mais recente, PHP 7. Se sua aplicação precisa de um banco de dados, você também deve instalar a Extensão PDO PHP²⁴ e o driver de banco de dados correspondente (tal como `pdo_mysql` para bancos de dados MySQL).

2.2.6 Configurando Servidores Web

Info: Você pode pular essa subseção por enquanto se estiver fazendo somente um test drive do Yii sem a intenção de publicá-lo em um servidor de produção.

²⁴https://www.php.net/manual/pt_BR/pdo.installation.php

A aplicação instalada de acordo com as instruções acima deve funcionar imediatamente com um Servidor HTTP Apache²⁵ ou um Servidor HTTP Nginx²⁶, no Windows, Mac OS X ou Linux usando PHP 5.4 ou superior. O Yii 2.0 também é compatível com o HHVM²⁷ do Facebook. No entanto, existem alguns casos extremos em que o HHVM se comporta diferentemente do PHP nativo, então você terá que ter um cuidado extra quando usar o HHVM.

Em um servidor de produção, você pode querer configurar o seu servidor Web de modo que a aplicação possa ser acessada pela URL `https://www.example.com/index.php` ao invés de `https://www.example.com/basico/web/index.php`. Tal configuração requer que você aponte a raiz dos documentos de seu servidor Web para o diretório `basico/web`. Você também pode querer ocultar o `index.php` da URL, conforme descrito na seção [Roteamento e Criação de URL](#). Nessa sub-seção, você aprenderá como configurar o seu servidor Apache ou Nginx para atingir estes objetivos.

Info: Definindo `basico/web` como a raiz dos documentos, você também evita que usuários finais acessem o código privado de sua aplicação e os arquivos de dados sensíveis que estão armazenados em diretórios no mesmo nível de `basico/web`. Negar o acesso a estes outros diretórios é uma melhoria de segurança.

Info: Se a sua aplicação rodará em um ambiente de hospedagem compartilhada onde você não tem permissão para alterar a configuração do seu servidor Web, você ainda pode ajustar a estrutura de sua aplicação para uma melhor segurança. Por favor, consulte a seção [Ambiente de Hospedagem Compartilhada](#) para mais detalhes.

Configuração do Apache Recomendada

Use a seguinte configuração no arquivo `httpd.conf` do Apache ou em uma configuração de virtual host. Perceba que você pode deve substituir `caminho/para/basico/web` com o caminho real para `basico/web`.

```
# Torna "basico/web" a raiz de documentos
DocumentRoot "caminho/para/basico/web"

<Directory "caminho/para/basico/web">
    # Usa mod_rewrite para suporte a URLs amigáveis
    RewriteEngine on

    # Se $showScriptName for "false" no UrlManager, impede o acesso a URLs
    # que tenham o nome do script (index.php)
```

²⁵<https://httpd.apache.org/>

²⁶<https://nginx.org/>

²⁷<https://hhvm.com/>

```

RewriteRule ^index.php/ - [L,R=404]

# Se um arquivo ou diretório existe, usa a solicitação diretamente
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# Caso contrário, redireciona para index.php
RewriteRule . index.php

# ... outras configurações ...
</Directory>

```

Configuração do Nginx Recomendada

Para usar o Nginx²⁸, você deve ter instalado o PHP como um FPM SAPI²⁹. Use a seguinte configuração do Nginx, substituindo `caminho/para/basico/web` com o caminho real para `basico/web` e `mysite.test` com o nome de host real a servir.

```

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        /caminho/para/basico/web;
    index       index.php;

    access_log  /caminho/para/basico/log/access.log;
    error_log   /caminho/para/basico/log/error.log;

    location / {
        # Redireciona tudo que não é um arquivo real para index.php
        try_files $uri $uri/ /index.php$is_args$args;
    }

    # Descomente para evitar processar chamadas feitas pelo Yii a arquivos
    # estáticos não existentes
    #location ~ \.(js/css/png/jpg/gif/swf/ico/pdf/mov/fla/zip/rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.html;

    # Nega acesso a arquivos php no diretório /assets
    location ~ ^/assets/.*\.php$ {
        deny all;
    }
}

```

²⁸<https://wiki.nginx.org/>

²⁹https://www.php.net/manual/pt_BR/install.fpm.php

```
location ~ /\.php$ {
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    fastcgi_pass 127.0.0.1:9000;
    #fastcgi_pass unix:/var/run/php5-fpm.sock;
    try_files $uri =404;
}

location ~* /\. {
    deny all;
}
}
```

Ao usar esta configuração, você também deve definir `cgi.fix_pathinfo=0` no arquivo `php.ini` de modo a evitar muitas chamadas desnecessárias ao comando `stat()` do sistema.

Também perceba que ao rodar um servidor HTTPS, você precisa adicionar `fastcgi_param HTTPS on`;, de modo que o Yii possa detectar adequadamente se uma conexão é segura.

2.3 Executando Aplicações

Após instalar o Yii, você tem uma aplicação Yii funcional que pode ser acessada pela URL `https://hostname/basico/web/index.php` ou `https://hostname/index.php`, dependendo de sua configuração. Esta seção introduzirá a funcionalidade embutida da aplicação, como o código é organizado e como a aplicação trata as requisições em geral.

Info: Por questões de simplicidade, por todo este tutorial de “Primeiros Passos” assume-se que você definiu `basico/web` como a raiz de documentos do seu servidor Web e configurou a URL de acesso de sua aplicação como `https://hostname/index.php` ou algo semelhante. Por favor, ajuste as URLs em nossas descrições conforme necessário.

Observe que, ao contrário do framework em si, após o template de projeto ser instalado, ele é todo seu. Você está livre para adicionar ou remover código e modificar o template conforme precisar.

2.3.1 Funcionalidade

O template básico de projetos instalado contém quatro páginas:

- A página inicial, exibida quando você acessa a URL `https://hostname/index.php`,
- a página “About” (Sobre),
- a página “Contact” (Contato), que exibe um formulário de contato que permite que usuários finais entrem em contato com você via e-mail,

- e a página “Login”, que exibe um formulário de login que pode ser usado para autenticar usuários finais. Tente fazer o login com “admin/admin”, e você perceberá que o item do menu principal “Login” mudará para “Logout”.

Essas páginas compartilham o mesmo cabeçalho e rodapé. O cabeçalho contém uma barra de menu principal que permite a navegação entre as diferentes páginas.

Você também deverá ver uma barra de ferramentas no rodapé da janela do navegador. Essa é uma ferramenta de depuração muito útil fornecida pelo Yii para registrar e exibir várias informações de depuração, tais como: mensagens de logs, status de respostas, as consultas de banco de dados executadas, e assim por diante.

Além da aplicação Web, existe um script de console chamado `yii`, que está localizado no diretório raiz da aplicação. Esse script pode ser usado para executar rotinas em segundo plano e tarefas de manutenção da aplicação que são descritas na seção [Comandos de Console](#).

2.3.2 Estrutura da Aplicação

Os diretórios e arquivos mais importantes em sua aplicação, assumindo que o diretório raiz dela é o `basico`, são:

<code>basico/</code>	caminho base de sua aplicação
<code>composer.json</code>	usado pelo Composer, descreve informações de pacotes
<code>config/</code>	contém as configurações da aplicação e outras
<code>console.php</code>	a configuração da aplicação de console
<code>web.php</code>	a configuração da aplicação Web
<code>commands/</code>	contém classes de comandos do console
<code>controllers/</code>	contém classes de controllers (controladores)
<code>models/</code>	contém classes de models (modelos)
<code>runtime/</code>	contém arquivos gerados pelo Yii durante o tempo de
execução, tais como	logs e arquivos de cache
<code>vendor/</code>	contém os pacotes do Composer instalados, incluindo
o próprio Yii framework	
<code>views/</code>	contém arquivos de views (visões)
<code>web/</code>	raiz da aplicação Web, contém os arquivos acessíveis
pela Web	
<code>assets/</code>	contém os arquivos de assets (javascript e css)
publicados pelo Yii	
<code>index.php</code>	o script de entrada (ou bootstrap) para a aplicação
<code>yii</code>	o script de execução dos comandos de console do Yii

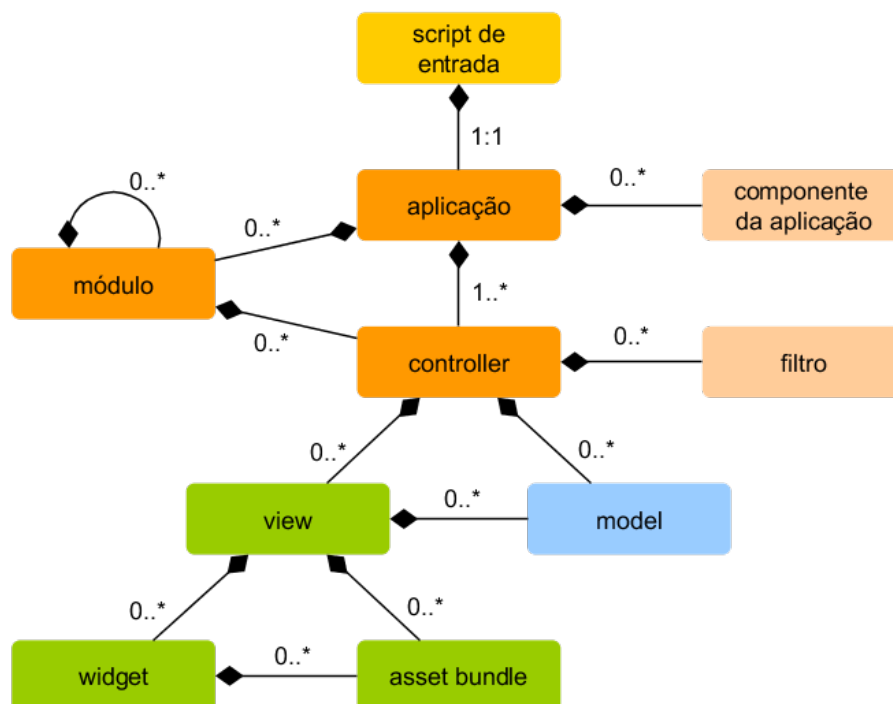
Em geral, os arquivos na aplicação podem ser divididos em dois tipos: aqueles em `basico/web` e aqueles em outros diretórios. Os primeiros podem ser acessados diretamente via HTTP (ou seja, em um navegador), enquanto os demais não podem e deveriam ser acessados.

O Yii implementa o padrão de arquitetura modelo-visão-controlador (MVC)³⁰,

³⁰<https://wikipedia.org/wiki/Model-view-controller>

que se reflete na organização de diretórios acima. O diretório `models` contém todas as [classes de modelos](#), o diretório `views` contém todos os [scripts de visões](#), e o diretório `controllers` contém todas as [classes de controladores](#).

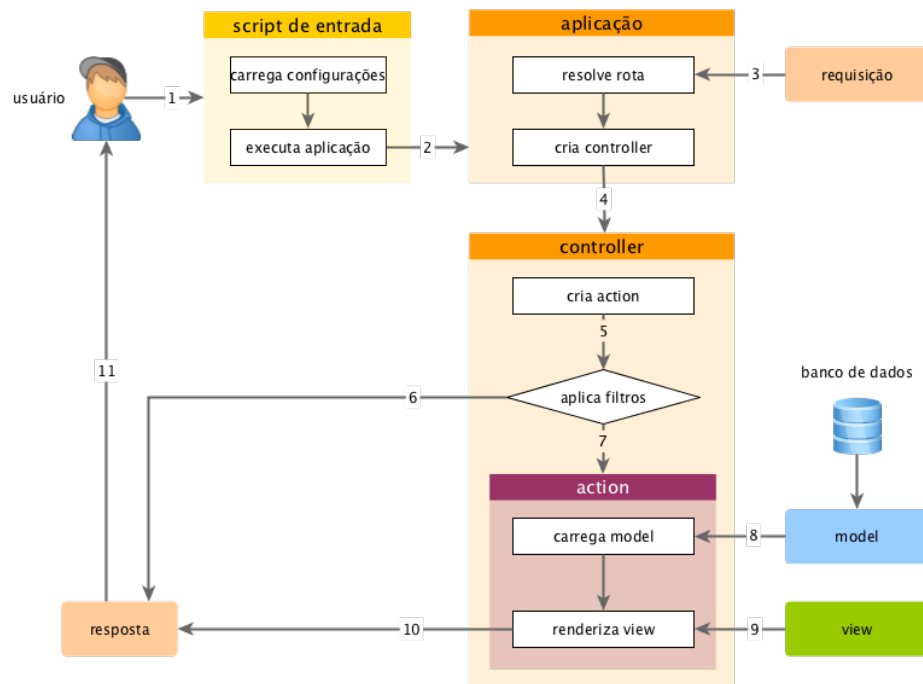
O diagrama a seguir demonstra a estrutura estática de uma aplicação.



Cada aplicação tem um script de entrada `web/index.php` que é o único script PHP acessível pela Web na aplicação. O script de entrada recebe uma requisição e cria uma instância de [aplicação](#) para tratar a requisição. A [aplicação](#) resolve ("traduz") a requisição com a ajuda de seus [componentes](#) e despacha a requisição para os elementos do MVC. São usados [Widgets](#) nas [views](#) para ajudar a construir elementos de interface de usuário complexos e dinâmicos.

2.3.3 Ciclo de Vida da Requisição

O diagrama a seguir demonstra como uma aplicação trata uma requisição.



1. Um usuário faz uma requisição ao **script de entrada** `web/index.php`.
2. O script de entrada carrega a **configuração** da aplicação e cria uma instância de **aplicação** para tratar a requisição.
3. A aplicação resolve ("traduz") a **rota** solicitada com a ajuda do componente **request** da aplicação.
4. A aplicação cria uma instância de um **controller** para tratar a requisição.
5. O controller cria uma instância de uma **action** (ação) e aplica os filtros para a ação.
6. Se qualquer filtro falhar, a ação é cancelada.
7. Se todos os filtros passarem, a ação é executada.
8. A ação carrega alguns modelos (models) de dados, possivelmente a partir de um banco de dados.
9. A ação renderiza uma view, passando a ela os modelos de dados.
10. O resultado renderizado é retornado pelo componente **response** (resposta) da aplicação.
11. O componente response envia o resultado renderizado para o navegador do usuário.

2.4 Dizendo “Olá!”

Esta seção descreve como criar uma nova página de “Olá!” em sua aplicação. Para atingir este objetivo, você criará uma [action](#) e uma [view](#):

- A aplicação enviará a requisição de página para a action
- e a action, por sua vez, renderizará a view que mostra a palavra “Olá!” ao usuário final.

Através deste tutorial, você aprenderá três coisas:

1. Como criar uma [action](#) para responder às requisições,
2. como criar uma [view](#) para compor o conteúdo da resposta, e
3. como uma aplicação envia requisições às [actions](#).

2.4.1 Criando uma Action

Para a tarefa “Olá!”, você criará uma [action](#) `cumprimentar` que lê um parâmetro `mensagem` da requisição e exibe essa mensagem de volta para o usuário. Se a requisição não fornecer um parâmetro `mensagem`, a action exibirá a mensagem padrão “Olá!”.

Info: [Actions](#) são os objetos que usuários finais podem solicitar diretamente para execução. Actions são agrupadas nos [controllers](#). O resultado da execução de uma action é a resposta que o usuário final receberá.

As actions devem ser declaradas em [controllers](#). Para manter a simplicidade, você pode declarar a action `cumprimentar` na classe já existente `SiteController`. Esse controller está definido no arquivo `controllers/SiteController.php`. Segue aqui o início da nova action:

```
<?php

namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    // ...código existente...

    public function actionCumprimentar($mensagem = 'Olá!')
    {
        return $this->render('cumprimentar', ['mensagem' => $mensagem]);
    }
}
```

No código acima, a action `cumprimentar` está definida como um método chamado `actionCumprimentar` na classe `SiteController`. O Yii usa o prefixo `action` para diferenciar os métodos de actions dos métodos que não são de actions em uma classe de controller. O nome após o prefixo `action` é mapeado como o ID da action.

Quando se trata de dar nome às suas actions, você deveria entender como o Yii trata os IDs de actions. Os IDs de actions são sempre referenciados em minúsculo. Se o ID de uma action necessitar de múltiplas palavras, elas serão concatenadas por hífens (por exemplo, `criar-comentario`). Os IDs de actions são convertidos em nomes de actions removendo-se os hífens dos IDs, colocando em maiúscula a primeira letra de cada palavra, e prefixando o resultado com a palavra `action`. Por exemplo, o ID de action `criar-comentario` corresponde ao método de action `actionCriarComentario`.

O método da action em nosso exemplo recebe um parâmetro `$mensagem`, cujo valor padrão é “Olá!” (exatamente da mesma forma que você define um valor padrão para qualquer argumento de função ou método no PHP). Quando a aplicação recebe a requisição e determina que a action `cumprimentar` é responsável por tratar a requisição, a aplicação vai preencher esse parâmetro com o parâmetro que tiver o mesmo nome na requisição. Em outras palavras, se a requisição inclui um parâmetro `mensagem` com o valor “Adeus!”, a variável `$mensagem` na action receberá esse valor.

Dentro do método da action, `render()` é chamado para renderizar um arquivo de `view` chamado `cumprimentar`. O parâmetro `mensagem` também é passado para a view de modo que ele possa ser usado por ela. O resultado da renderização da view é retornado pelo método da action. Esse resultado será recebido pela aplicação e exibido para o usuário final no navegador (como parte de uma página HTML completa).

2.4.2 Criando uma View

As `views` são scripts que você escreve para gerar o conteúdo de uma resposta. Para a tarefa “Olá!”, você criará uma view `cumprimentar` que exibe o parâmetro `mensagem` recebido do método da action:

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

A view `cumprimentar` deve ser salva no arquivo `views/site/cumprimentar.php`. Quando o método `render()` é chamado em uma action, ele procurará o arquivo PHP em `views/IDdoController/NomeDaView.php`.

Perceba que no código acima o parâmetro `mensagem` é codificado como HTML antes de ser impresso. Isso é necessário, já que o parâmetro vem de um

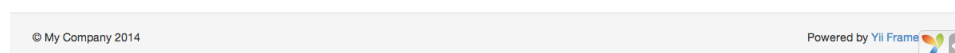
usuário final, tornando-o vulnerável a ataques de cross-site scripting (XSS)³¹ por embutir código JavaScript malicioso no parâmetro.

Naturalmente, você pode colocar mais conteúdo na view `cumprimentar`. O conteúdo pode consistir de tags HTML, texto puro, ou até mesmo instruções de PHP. De fato, a view `cumprimentar` é apenas um script PHP que é executado pelo método `render()`. O conteúdo impresso pelo script da view será retornado à aplicação como o resultado da resposta. A aplicação, por sua vez, retornará esse resultado para o usuário final.

2.4.3 Conferindo

Após criar a action e a view, você pode acessar a nova página através da seguinte URL:

`https://hostname/index.php?r=site/cumprimentar&mensagem=Olá+Mundo!`



Essa URL resultará em uma página exibindo “Olá Mundo!”. Essa página compartilha o mesmo cabeçalho e rodapé das outras páginas da aplicação.

Se você omitir o parâmetro `mensagem` na URL, você verá a página exibindo somente “Olá!”. Isso ocorre por que `mensagem` é passado como um parâmetro para o método `actionCumprimentar()` e, quando ele é omitido, o valor padrão “Olá!” é usado em seu lugar.

Info: A nova página compartilha o mesmo cabeçalho e rodapé de outras páginas porque o método `render()` vai automaticamente

³¹https://en.wikipedia.org/wiki/Cross-site_scripting

incluir o resultado da view `cumprimentar` em um [layout](#) que neste caso está localizado em `views/layouts/main.php`.

O parâmetro `r` na URL acima requer mais explicação. Ele significa [rota](#), um ID único e amplo de uma aplicação que se refere a uma action. O formato da rota é `IDdoController/IDdaAction`. Quando a aplicação recebe uma requisição, ela verificará esse parâmetro e usará a parte `IDdoController` para determinar qual classe de controller deve ser instanciada para tratar a requisição. Então o controller usará a parte `IDdaAction` para determinar qual action deverá ser instanciada para fazer o trabalho. No caso deste exemplo, a rota `site/cumprimentar` será resolvida como a classe de controller `SiteController` e a action `cumprimentar`. Como resultado, o método `SiteController::actionCumprimentar()` será chamado para tratar a requisição.

Info: Assim como as actions, os controllers também possuem IDs que os identificam de maneira única em uma aplicação. IDs de controllers seguem as mesmas regras de nomenclatura dos IDs de actions. Os nomes das classes de controllers derivam dos IDs de controllers removendo-se os hífens dos IDs, convertendo a primeira letra de cada palavra em maiúscula, e adicionando o sufixo `Controller`. Por exemplo, o ID de controller `comentario-de-artigo` corresponde ao nome de classe de controller `ComentarioDeArtigoController`.

2.4.4 Resumo

Nesta seção, você teve uma introdução sobre as partes controller e view do padrão de arquitetura MVC. Você criou uma action como parte de um controller para tratar uma requisição específica. E você também criou uma view para compor o conteúdo da resposta. Nesse exemplo simples, nenhum modelo (model) foi utilizado, já que o único dado exibido foi o parâmetro `mensagem`.

Você também aprendeu sobre as rotas no Yii, que agem como a ponte entre as requisições de usuário e as actions de controllers.

Na próxima seção, você aprenderá como criar um modelo (model) e adicionar uma nova página contendo um formulário HTML.

2.5 Trabalhando com Formulários

Esta seção descreve como criar uma nova página com um formulário para receber dados dos usuários. A página exibirá um formulário com um campo para o nome e outro para o e-mail. Depois de obter essas duas informações do usuário, a página exibirá os valores inseridos de volta para confirmação.

Para alcançar esse objetivo, além de criar uma [action](#) e duas [views](#), você também criará um [model](#).

No decorrer deste tutorial, você aprenderá como:

- criar um [model](#) para representar os dados que o usuário insere por meio de um formulário
- declarar regras (rules) para validar os dados inseridos
- criar um formulário HTML em uma [view](#)

2.5.1 Criando um Model

Os dados a serem solicitados do usuário serão representados por uma classe model `FormularioDeRegistro`, como visto a seguir, que será salva no arquivo `models/FormularioDeRegistro.php`. Por favor, consulte a seção [Autoloading de Classes](#) para mais detalhes sobre convenção de nomenclatura de arquivos de classes.

```
<?php

namespace app\models;

use Yii;
use yii\base\Model;

class FormularioDeRegistro extends Model
{
    public $nome;
    public $e_mail;

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

A classe estende de `yii\base\Model`, uma classe base fornecida pelo Yii comumente usada para representar dados de formulários.

Info: `yii\base\Model` é usado como pai das classes de models que *não* são associadas com tabelas de bancos de dados. Enquanto `yii\db\ActiveRecord`, como pai das classes de models que correspondem a tabelas de bancos de dados.

A classe `FormularioDeRegistro` contém dois atributos públicos, `nome` e `e_mail`, que são usados para armazenar os dados fornecidos pelo usuário. Ele também contém um método chamado `rules()`, que retorna um conjunto de regras para validação dos dados. As regras de validação declaradas no código acima declaram que:

- tanto o `nome` quanto o `e_mail` são obrigatórios
- o `e_mail` deve ser preenchido com um e-mail sintaticamente válido (por exemplo, um valor sem `@` não pode ser considerado válido para um e-mail, etc.)

Se você tem um objeto `FormularioDeRegistro` preenchido com dados fornecidos pelo usuário, você pode chamar seu método `validate()` para iniciar as rotinas de validação dos dados. Se a validação de dados falhar, a propriedade `hasErrors` será definida como `true` e você pode saber quais erros de validação ocorreram por consultar `errors`.

```
<?php
$model = new FormularioDeRegistro();
$model->nome = 'Fulano';
$model->e_mail = 'emailruim';
if ($model->validate()) {
    // Bom!
} else {
    // Falhou!
    // Utilize $model->getErrors()
}
```

2.5.2 Criando uma Action

Em seguida, você precisará criar uma action `registro` no controller `site` que usará o novo model. O processo de criação e utilização de ações foi explicado na seção [Dizendo “Olá!”](#).

```
<?php

namespace app\controllers;

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

class SiteController extends Controller
{
    // ...código existente...

    public function actionRegistro()
    {
        $model = new FormularioDeRegistro();

        if ($model->load(Yii::$app->request->post()) && $model->validate())
        {
            // dados válidos recebidos no $model

            // faça alguma coisa significativa com o $model aqui ...

            return $this->render('confirmar-registro', ['model' => $model]);
        } else {
```

```

        // Ou a página esta sendo exibida inicial ou houve algum erro de
        // validação
        return $this->render('registro', ['model' => $model]);
    }
}
}

```

A primeira action cria um objeto `FormularioDeRegistro`. Ele, então, tenta preencher o model com os dados de `$_POST`, fornecidos no Yii por `yii\web\Request::post()`. Se o model for preenchido (observe o método `load()`) com sucesso, ou seja, se o usuário enviou o formulário HTML, a action chamará o `validate()` para se certificar de que os valores fornecidos são válidos.

Info: A expressão `Yii::$app` representa a instância da aplicação, que é um “singleton” globalmente acessível. Ao mesmo tempo, é um **service locator** que fornece componentes tais como `request`, `response`, `db`, etc., para permitir funcionalidades específicas. No código acima, o componente `request` da instância da aplicação é usado para acessar os dados do `$_POST`.

Se tudo estiver certo, a action renderizará a view chamada `confirmar-registro` para confirmar ao usuário que os dados foram enviados corretamente. Se nenhum dado foi enviado ou se tiverem erros, a view `registro` será renderizada novamente e seu formulário HTML voltará a ser exibido mas, dessa vez, juntamente com as mensagens de erro de validação.

Nota: Neste exemplo muito simples, renderizamos a página de confirmação somente se os dados enviados eram válidos. Na prática, você deve considerar usar `refresh()` ou `redirect()` para evitar problemas de reenvio de formulário³².

2.5.3 Criando Views

Por fim, crie dois arquivos de views chamados `confirmar-registro` e `registro`. Essas serão as views renderizadas pela action `registro`, como acabamos de descrever acima.

A view `confirmar-registro` simplesmente exibe os dados dos campos `nome` e `e_mail` e deve ser salva no arquivo `views/site/confirmar-registro.php`.

```

<?php
use yii\helpers\Html;
?>
<p>Você enviou as seguintes informações:</p>

<ul>
    <li><label>Nome</label>: <?= Html::encode($model->nome) ?></li>

```

³²<https://en.wikipedia.org/wiki/Post/Redirect/Get>

```
<li><label>E-mail</label>: <?= Html::encode($model->e_mail) ?></li>
</ul>
```

A view `registro` exibe um formulário HTML e deve ser salva no arquivo `views/site/registro.php`.

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

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

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

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

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

A view usa um poderoso `widget` chamado `ActiveForm` para construir o formulário HTML. Os métodos `begin()` e `end()` do widget renderizam as tags de abertura e de fechamento do formulário. Entre as duas chamadas de método, campos são criados pelo método `field()`. O primeiro campo é para o nome e o segundo é para o e-mail. Após os campos, o método `yii\helpers\Html::submitButton()` é chamado para criar um botão de envio do formulário (submit).

2.5.4 Conferindo

Para ver como funciona, utilize seu navegador para acessar a seguinte URL:

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

Você verá uma página exibindo um formulário com dois campos. Na frente de cada campo, um *label* indica quais dados devem ser inseridos. Se você clicar no botão Enviar sem informar nenhum dado, ou se você não fornecer um e-mail válido, você verá uma mensagem de erro próxima a cada campo com problema.

The screenshot shows a web form for 'My Company'. At the top is a dark navigation bar with 'My Company' on the left and links for 'Home', 'About', 'Contact', and 'Login' on the right. Below the navigation bar, there are two input fields. The first is labeled 'Nome' and has a red border with a red error message below it: '*Nome* não pode ficar em branco.' The second is labeled 'Email' and also has a red border with a red error message below it: '*Email* não pode ficar em branco.' Below the email field is a blue button labeled 'Enviar'. At the bottom of the form is a light gray footer bar containing '© My Company 2014' on the left and 'Powered by Yii Frame' with a logo on the right.

Após informar um nome e e-mail válidos e clicar no botão de enviar, você verá uma nova página exibindo os dados informados por você.

The screenshot shows the result of submitting the form. It features the same 'My Company' navigation bar at the top. Below the navigation bar, the text 'Você enviou as seguintes informações:' is displayed. Underneath this text is a bulleted list showing the submitted data: '• Nome: Fulano' and '• E-mail: fulano@exemplo.com'. At the bottom is the same light gray footer bar with '© My Company 2014' and 'Powered by Yii Frame' with a logo.

Mágica Explicada

Você talvez se pergunte como o formulário HTML trabalha por trás das cortinas, já que parece quase mágica exibir um *label* para cada campo e mostrar mensagens de erro sem recarregar a página quando você não informa

os dados corretamente.

Sim, a validação de dados inicialmente é feita no cliente usando JavaScript e, posteriormente, realizada no servidor via PHP. O `yii\widgets\ActiveForm` é esperto o suficiente para extrair as regras de validação que você declarou no `FormularioDeRegistro`, transformá-las em código JavaScript executável e usar esse código JavaScript para realizar a validação de dados. Caso você tenha desabilitado o JavaScript em seu navegador, a validação ainda será realizada no servidor, como mostrado no método `actionRegistro()`. Isso garante a validade dos dados em todas as circunstâncias.

Warning: (Alerta!) A validação feita no cliente é uma conveniência que fornece uma melhor experiência para o usuário. A validação feita no servidor é sempre necessária, quer a validação no cliente aconteça, quer não.

Os *labels* dos campos são gerados pelo método `field()`, usando os nomes das propriedades do model. Por exemplo, um *label* chamado `Nome` será gerado para a propriedade `nome`.

Você pode personalizar um *label* em uma view utilizando o seguinte código:

```
<?= $form->field($model, 'nome')->label('Seu Nome') ?>
<?= $form->field($model, 'e_mail')->label('Seu E-mail') ?>
```

Info: O Yii fornece muitos desses widgets para ajudá-lo a construir rapidamente views dinâmicas e complexas. Conforme você vai aprender mais tarde, escrever um novo widget também é extremamente fácil. Você talvez queira transformar grande parte do código de suas views em widgets reutilizáveis para simplificar o desenvolvimento de views no futuro.

2.5.5 Resumo

Nesta seção do guia, você teve uma introdução à última parte do padrão de arquitetura MVC. Você aprendeu como criar uma classe model para representar os dados do usuário e validá-los.

Também aprendeu como obter os dados enviados pelos usuários e como exibi-los de volta no navegador. Essa é uma tarefa que poderia tomar muito tempo ao desenvolver uma aplicação, mas o Yii fornece widgets poderosos que tornam o processo muito simples.

Na próxima seção, você aprenderá como trabalhar com bancos de dados, que são necessários em quase todas as aplicações.

2.6 Trabalhando com Bancos de Dados

Esta seção descreverá como criar uma nova página que exibe informações de países obtidos de uma tabela de banco de dados chamada `pais`. Para isso, você configurará uma conexão com o banco de dados, criará uma classe de [Active Record](#), definirá uma [action](#) e criará uma [view](#).

Ao longo deste tutorial, você aprenderá como:

- configurar uma conexão de BD
- definir uma classe Active Record
- consultar dados usando a classe de Active Record
- exibir dados em uma view de forma paginada

Perceba que para terminar essa seção, você deve ter conhecimento e experiência básicos em bancos de dados. Em particular, você deve saber como criar um banco de dados e como executar instruções SQL usando uma ferramenta cliente de bancos de dados.

2.6.1 Preparando o Banco de Dados

Para começar, crie um banco de dados chamado `yii2basico`, de onde você obterá os dados em sua aplicação. Você pode criar um banco de dados SQLite, MySQL, PostgreSQL, MSSQL ou Oracle, já que o Yii tem suporte embutido a vários gerenciadores de bancos de dados. Por questões de simplicidade, será assumido o uso do MySQL na descrição a seguir.

Info: O MariaDB costumava ser um substituto transparente do MySQL. Isto já não é mais totalmente verdade. Caso você queira usar recursos avançados como suporte a JSON no MariaDB, por favor, consulte a extensão do MariaDB listada mais à frente.

Em seguida, crie uma tabela chamada `pais` no banco de dados e insira alguns dados de exemplo. Você pode rodar as seguintes declarações SQL para fazer isso:

```
CREATE TABLE `pais` (  
  `codigo` CHAR(2) NOT NULL PRIMARY KEY,  
  `nome` CHAR(52) NOT NULL,  
  `populacao` INT(11) NOT NULL DEFAULT '0'  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
INSERT INTO `pais` VALUES ('AU','Austrália',24016400);  
INSERT INTO `pais` VALUES ('BR','Brasil',205722000);  
INSERT INTO `pais` VALUES ('CA','Canadá',35985751);  
INSERT INTO `pais` VALUES ('CN','China',1375210000);  
INSERT INTO `pais` VALUES ('DE','Alemanha',81459000);  
INSERT INTO `pais` VALUES ('FR','França',64513242);  
INSERT INTO `pais` VALUES ('GB','Reino Unido',65097000);  
INSERT INTO `pais` VALUES ('IN','Índia',1285400000);  
INSERT INTO `pais` VALUES ('RU','Rússia',146519759);  
INSERT INTO `pais` VALUES ('US','Estados Unidos',322976000);
```

Neste ponto, você tem um banco de dados chamado `yii2basico` e dentro dele uma tabela `pais` com três colunas, contendo dez linhas de dados.

2.6.2 Configurando uma Conexão de BD

Antes de prosseguir, certifique-se de que você possui instalados tanto a extensão PDO³³ do PHP quanto o driver PDO para o gerenciador de banco de dados que você está usando (por exemplo, `pdo_mysql` para o MySQL). Este é um requisito básico se a sua aplicação usa um banco de dados relacional.

Tendo esses instalados, abra o arquivo `config/db.php` e mude os parâmetros conforme seu banco de dados. Por padrão, o arquivo contém o seguinte:

```
<?php

return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basico',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

O arquivo `config/db.php` é uma típica ferramenta de [configuração](#) baseada em arquivo. Este arquivo de configuração em particular especifica os parâmetros necessários para criar e inicializar uma instância `yii\db\Connection` por meio da qual você pode fazer consultas SQL ao banco de dados subjacente.

A conexão configurada acima pode ser acessada no código da aplicação através da expressão `Yii::$app->db`.

Info: O arquivo `config/db.php` será absorvido (inclusive) pela configuração principal da aplicação `config/web.php`, que especifica como a instância da [aplicação](#) deve ser inicializada. Para mais informações, por favor, consulte a seção [Configurações](#).

Se você precisa trabalhar com bancos de dados para os quais não há suporte nativo no Yii, consulte as seguintes extensões:

- Informix³⁴
- IBM DB2³⁵
- Firebird³⁶
- MariaDB³⁷

³³https://www.php.net/manual/pt_BR/book.pdo.php

³⁴<https://github.com/edgardmessias/yii2-informix>

³⁵<https://github.com/edgardmessias/yii2-ibm-db2>

³⁶<https://github.com/edgardmessias/yii2-firebird>

³⁷<https://github.com/sam-it/yii2-mariadb>

2.6.3 Criando um Active Record

Para representar e buscar os dados da tabela `pais`, crie uma classe que deriva de `Active Record` chamada `Pais` e salve-a no arquivo `models/Pais.php`.

```
<?php

namespace app\models;

use yii\db\ActiveRecord;

class Pais extends ActiveRecord
{
}
```

A classe `Pais` estende de `yii\db\ActiveRecord`. Você não precisa escrever nenhum código nela! Só com o código acima, o Yii descobrirá o nome da tabela associada a partir do nome da classe.

Info: Se não houver nenhuma correspondência direta do nome da classe com o nome da tabela, você pode sobrescrever o método `yii\db\ActiveRecord::tableName()` para especificar explicitamente o nome da tabela associada.

Usando a classe `Pais`, você pode manipular facilmente os dados na tabela `pais`, conforme é demonstrado nos fragmentos de código a seguir:

```
use app\models\Pais;

// obtém todas as linhas da tabela pais e as ordena pela coluna "nome"
$países = Pais::find()->orderBy('nome')->all();

// obtém a linha cuja chave primária é "BR"
$pais = Pais::findOne('BR');

// exibe "Brasil"
echo $pais->nome;

// altera o nome do país para "Brazil" e o salva no banco de dados
$pais->nome = 'Brazil';
$pais->save();
```

Info: O `Active Record` é uma maneira poderosa de acessar e manipular dados do banco de dados de uma forma orientada a objetos. Você pode encontrar informações mais detalhadas na seção `[Active Record](db-active-record.md)`. Alternativamente, você também pode interagir com o banco de dados usando um método de acesso a dados em baixo nível chamado `Objeto de Acesso a Dados (Data Access Objects)`.

2.6.4 Criando uma Action

Para disponibilizar os dados de países aos usuários finais, você precisa criar uma nova action. Em vez de colocar a nova action no controller `site` como você fez nas seções anteriores, faz mais sentido criar um novo controller especificamente para todas as actions relacionadas aos dados de países. Chame este novo controller de `PaisController`, e crie uma action `index` nele, conforme o exemplo a seguir:

```
<?php

namespace app\controllers;

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

class PaisController extends Controller
{
    public function actionIndex()
    {
        $query = Pais::find();

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

        $paises = $query->orderBy('nome')
            ->offset($paginacao->offset)
            ->limit($paginacao->limit)
            ->all();

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

Salve o código acima no arquivo `controllers/PaisController.php`.

A action `index` chama `Pais::find()`. Este método do Active Record constrói uma consulta ao BD e retorna todos os dados da tabela `pais`. Para limitar o número de países retornados a cada requisição, a consulta é paginada com a ajuda de um objeto `yii\data\Pagination`. O objeto `Pagination` serve para dois propósitos:

- Define as cláusulas `offset` e `limit` da declaração SQL representada pela query (consulta) de modo que apenas retorne uma única página de dados por vez (no exemplo, no máximo 5 linhas por página).
- É usado na view para exibir um paginador que consiste de uma lista de botões de páginas, conforme será explicado na próxima subseção.

No final do código, a action `index` renderiza uma view chamada `index` e envia a ela os dados dos países e as informações de paginação.

2.6.5 Criando uma View

Dentro do diretório `views`, primeiro crie um subdiretório chamado `pais`. Esta pasta será usada para guardar todas as views renderizadas pelo controller `PaisController`. Dentro do diretório `views/pais`, crie um arquivo `index.php` contendo o seguinte:

```
<?php
use yii\helpers\Html;
use yii\widgets\LinkPager;
?>
<h1>Países</h1>
<ul>
<?php foreach ($países as $pais): ?>
    <li>
        <? = Html::encode("{$pais->nome} ({$pais->codigo})") ?>:
        <? = $pais->populacao ?>
    </li>
<?php endforeach; ?>
</ul>

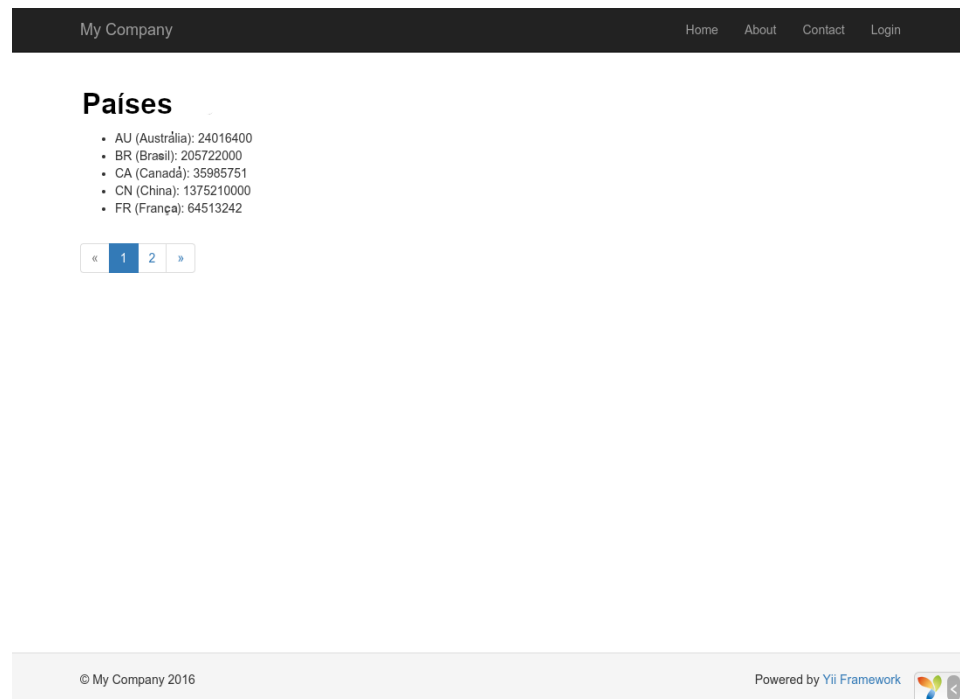
<? = LinkPager::widget(['pagination' => $paginacao]) ?>
```

A view tem duas seções relativas à exibição dos dados dos países. Na primeira parte, os dados de países fornecidos são percorridos e renderizados como uma lista HTML. Na segunda parte, um widget `yii\widgets\LinkPager` é renderizado usando as informações de paginação passadas pela action. O widget `LinkPager` exibe uma lista de botões de páginas. Clicar em qualquer um deles vai atualizar os dados dos países conforme a página correspondente.

2.6.6 Conferindo

Para ver se todo os códigos acima funcionam, use o seu navegador para acessar a seguinte URL:

<https://hostname/index.php?r=pais/index>



Primeiramente, você verá uma lista exibindo cinco países. Abaixo dos países, você verá um paginador com quatro botões. Se você clicar no botão “2”, você verá a página exibir outros cinco países do banco de dados: a segunda página de registros. Observe mais cuidadosamente e você perceberá que a URL no browser mudou para

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

Por trás das cortinas, **Pagination** está fornecendo toda a funcionalidade necessária para paginar um conjunto de dados:

- Inicialmente, **Pagination** representa a primeira página, que reflete a consulta `SELECT` de países com a cláusula `LIMIT 5 OFFSET 0`. Como resultado, os primeiros cinco países serão buscados e exibidos.
- O widget **LinkPager** renderiza os botões das páginas usando as URLs criadas pelo **Pagination**. As URLs conterão um parâmetro `page`, que representa os diferentes números de páginas.
- Se você clicar no botão de página “2”, uma nova requisição para a rota `pais/index` será disparada e tratada. **Pagination** lê o parâmetro `page` da URL e define o número da página atual como sendo 2. A nova consulta de países então terá a cláusula `LIMIT 5 OFFSET 5` e retornará os próximos cinco países para a exibição.

2.6.7 Resumo

Nesta seção, você aprendeu como trabalhar com um banco de dados. Você também aprendeu como buscar e exibir dados em páginas com a ajuda do

`yii\data\Pagination` e do `yii\widgets\LinkPager`.

Na próxima seção, você aprenderá como usar a poderosa ferramenta de geração de códigos, chamada Gii, para ajudá-lo a implementar rapidamente algumas funcionalidades comumente necessárias, tais como as operações CRUD (Criar-Ler-Atualizar-Excluir) para trabalhar com os dados de uma tabela do banco de dados. Na verdade, todo o código que você acabou de escrever pode ser gerado automaticamente no Yii usando a ferramenta Gii.

2.7 Gerando Código com Gii

Essa seção irá descrever como usar o Gii³⁸ para gerar automaticamente código que implementa algumas funcionalidades comuns de sites. Usar o Gii para gerar código é simplesmente uma questão de informar os dados corretos conforme as instruções mostradas nas páginas do Gii.

Através desse tutorial, você irá aprender a:

- habilitar o Gii em sua aplicação,
- usar o Gii para gerar uma classe Active Record,
- usar o Gii para gerar código que implementa as operações CRUD para uma tabela do banco de dados, e
- personalizar o código gerado pelo Gii.

2.7.1 Começando a usar o Gii

O Gii³⁹ é fornecido como um **módulo** do Yii. Você pode habilitar o Gii ao configurá-lo na propriedade `modules` da aplicação. Dependendo de como você criou sua aplicação, você pode encontrar o seguinte código já pronto no arquivo de configuração `config/web.php`:

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

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

A configuração acima declara que, quando estiver usando o **ambiente de desenvolvimento**, a aplicação deve incluir um módulo chamado `gii`, da classe `yii\gii\Module`.

Se você verificar o **script de entrada** `web/index.php` da sua aplicação, você encontrará a seguinte linha que, basicamente, torna `YII_ENV_DEV` verdadeira.

³⁸<https://github.com/yiisoft/yii2-gii/blob/master/docs/guide-pt-BR/README.md>

³⁹<https://github.com/yiisoft/yii2-gii/blob/master/docs/guide-pt-BR/README.md>

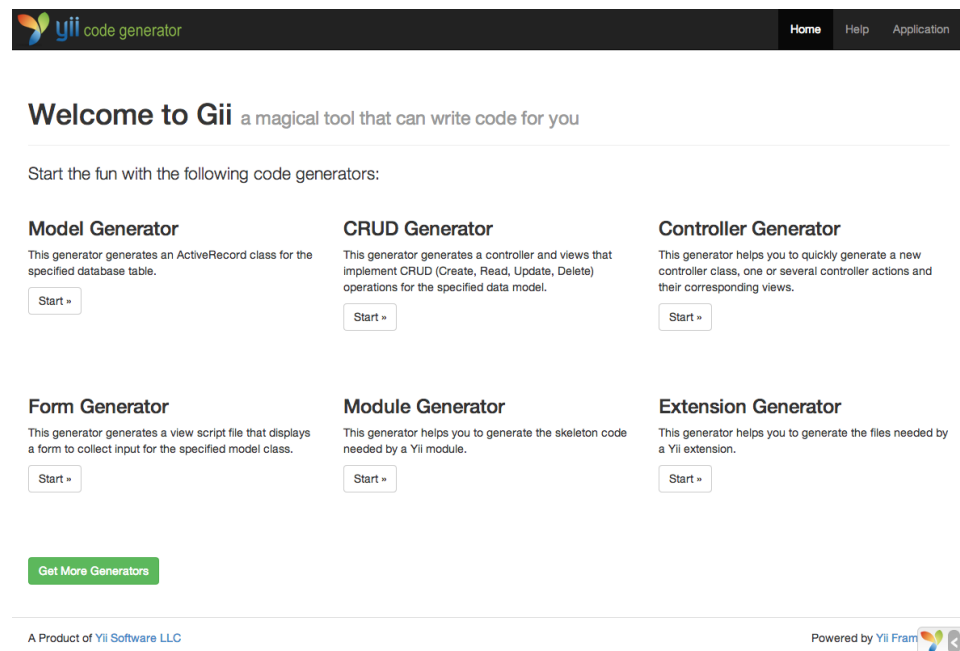
```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

Graças a essa linha, sua aplicação está em modo de desenvolvimento e terá o Gii habilitado, devido a configuração mais acima. Agora você pode acessar o Gii pela URL:

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

Note: Se você está acessando o Gii por um endereço IP que não seja o localhost, o acesso será negado por padrão, por questões de segurança. Você pode configurar o Gii adicionando endereços IP permitidos como mostrado a seguir:

```
'gii' => [
    'class' => 'yii\gii\Module',
    'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*',
        '192.168.178.20'] // ajuste de acordo com suas necessidades
],
```



2.7.2 Gerando uma classe Active Record

Para gerar uma classe Active Record usando o Gii, selecione o “Model Generator” clicando no link na página inicial do Gii. Então, preencha o formulário como indicado abaixo:

- Nome da tabela: `pais`
- Classe do modelo: `Pais`

Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name

country

Model Class

Country

Namespace

app\models

Base Class

yii\db\ActiveRecord

Database Connection ID

db

☐ Use Table Prefix

☒ Generate Relations

☐ Generate Labels from DB Comments

☐ Enable I18N

Code Template

default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/model/default)

[Preview](#)

A Product of [Yii Software LLC](#)

Powered by [Yii Framework](#)

Em seguida, clique no botão “Preview”. Você verá o `models/Pais.php` listado como arquivo a ser criado. Você pode clicar no nome do arquivo para pré-visualizar seu conteúdo.

Ao usar o Gii, se você já havia criado o mesmo arquivo e pretende sobrescrevê-lo, clique no botão `diff` próximo ao nome do arquivo para ver as diferenças entre o código a ser gerado e a versão já existente.

Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name
country

Model Class
Country

Namespace
app\models

Base Class
yii\db\ActiveRecord

Database Connection ID
db

☐ Use Table Prefix

☒ Generate Relations

☐ Generate Labels from DB Comments

☐ Enable I18N

Code Template
default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/model/default)

[Preview](#) [Generate](#)

Click on the above **Generate** button to generate the files selected below:

Code File	Action
models/Country.php	overwrite

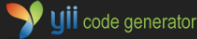
Quando estiver sobrescrevendo um arquivo, marque a caixa próxima a “overwrite” (sobrescrever) e clique no botão “Generate”. Se estiver criando um novo arquivo, apenas clique em “Generate”.

Em seguida, você verá uma página de confirmação indicando que o código foi gerado com sucesso. Se você já tinha um arquivo, também verá uma mensagem indicando que ele foi sobrescrito pelo novo código.

2.7.3 Gerando código CRUD

CRUD significa a Create, Read, Update, and Delete (criar, ler, atualizar e apagar), representando as quatro tarefas comuns feitas com os dados na maioria das aplicações. Para criar funcionalidades CRUD usando o Gii, selecione “CRUD Generator” clicando no link na página inicial do Gii. Seguindo o exemplo “pais”, preencha o formulário com as seguintes informações:

- Model Class: `app\models\Pais`
- Search Model Class: `app\models\PaisSearch`
- Controller Class: `app\controllers\PaisController`

 Home Help Application

Model Generator >

CRUD Generator >

Controller Generator >

Form Generator >

Module Generator >

Extension Generator >

CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.

Model Class

Search Model Class

Controller Class

View Path

Base Controller Class

yii\web\Controller

Widget Used in Index Page

GridView

☐ **Enable I18N**

Code Template

default (C:\dev\yii2\extensions\yii\generators\crud\default)

[Preview](#)

A Product of [Yii Software LLC](#)

Powered by [Yii Framework](#)

Em seguida, clique no botão “Preview”. Você verá uma lista de arquivos a serem gerados, como mostrado abaixo.

The screenshot shows the 'yii code generator' interface. The configuration is as follows:

- Base Controller Class:** yiiweb\Controller
- Widget Used in Index Page:** GridView
- Enable I18N:** ☐
- Code Template:** default (C:\dev\yii2\extensions\gii\generators\crud\default)

Buttons: Preview, Generate

Click on the above **Generate** button to generate the files selected below:

Legend: ☒ Create ☐ Unchanged ☒ Overwrite

Code File	Action	
controllers/CountryController.php	create	<input checked="" type="checkbox"/>
models/CountrySearch.php	create	<input checked="" type="checkbox"/>
views/country/_form.php	create	<input checked="" type="checkbox"/>
views/country/_search.php	create	<input checked="" type="checkbox"/>
views/country/create.php	create	<input checked="" type="checkbox"/>
views/country/index.php	create	<input checked="" type="checkbox"/>
views/country/update.php	create	<input checked="" type="checkbox"/>
views/country/view.php	create	<input checked="" type="checkbox"/>

Se você criou anteriormente os arquivos `controllers/PaisController.php` e `views/pais/index.php` (na seção de banco de dados deste guia), marque a caixa “overwrite” para substituí-los. (As versões anteriores não tinham suporte completo às operações CRUD.)

2.7.4 Conferindo

Para ver como ficou, use seu navegador para acessar a seguinte URL:

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

Você verá uma tabela mostrando os países do seu banco de dados. Você pode ordená-los ou filtrá-los inserindo condições nos cabeçalhos das colunas.

Para cada país exibido na tabela, você pode visualizar detalhes, editar ou excluir. Você também pode clicar no botão “Create Pais” no topo da tabela para ter acesso a um formulário para cadastrar um novo país.

My Company































HomeAboutContactLogin

Home / Pais

Pais

Create Pais

Showing 1-10 of 10 items.

#	Codigo	Nome	Populacao	
	<input type="text"/>	<input type="text"/>	<input type="text"/>	
1	AU	Australia	18886000	  
2	BR	Brazil	170115000	  
3	CA	Canada	1147000	  
4	CN	China	1277558000	  
5	DE	Germany	82164700	  
6	FR	France	59225700	  
7	GB	United Kingdom	59623400	  
8	IN	India	1013662000	  
9	RU	Russia	146934000	  
10	US	United States	278357000	  

< 1 >

© My Company 2014

Powered by Yii Frame

My Company

HomeAboutContactLogin

Home / Pais / Brasil / Update

Update Pais : Brasil

Codigo

Nome

Populacao

Update

© My Company 2014

Powered by Yii Frame

Esta é uma lista de arquivos gerados pelo Gii, caso você queira investigar como essas funcionalidades estão implementadas ou ajustá-las:

- Controller: `controllers/PaisController.php`
- Modelo: `models/Pais.php` e `models/PaisSearch.php`

- Views: `views/pais/*.php`

Info: o Gii é projetado para ser uma ferramenta altamente adaptável e extensível. Usando-o sabiamente, você pode acelerar bastante o desenvolvimento de sua aplicação. Para mais detalhes, por favor, consulte a seção Gii⁴⁰.

2.7.5 Resumo

Nessa seção, você aprendeu a usar o Gii para gerar código que implementa todas as operações CRUD para o conteúdo armazenado em uma tabela de banco de dados.

2.8 Seguindo em Frente

Se você leu toda a seção “Primeiros Passos”, você criou uma aplicação Yii completa. No processo, você aprendeu como implementar algumas funcionalidades comumente necessárias, tais como obter dados de usuários através de um formulário HTML, consultar dados de um banco de dados e exibir os dados de forma paginada. Você também aprendeu a usar o Gii para gerar código automaticamente. Usar o Gii para a geração de código torna a carga do seu processo de desenvolvimento Web uma tarefa tão simples quanto preencher alguns formulários.

Esta seção resume recursos Yii disponíveis que o ajudarão a ser mais produtivo usando o framework.

- Documentação
 - O Guia Definitivo⁴¹: Conforme o nome indica, o guia define precisamente como o Yii deve funcionar e fornece orientações gerais sobre como usá-lo. É o tutorial mais importante e que você deveria ler antes de escrever qualquer código Yii.
 - A Referência de Classes⁴²: Especifica o uso de todas as classes fornecidas pelo Yii. Deve ser principalmente usada quando você estiver escrevendo código e quiser entender o uso de uma classe, método ou propriedade específicos. O uso da referência de classes é idealmente melhor somente depois de um entendimento contextual do framework como um todo.
 - Os Artigos Wiki⁴³: Os artigos wiki são escritos por usuários do Yii baseados em suas próprias experiências. A maioria deles são

⁴⁰<https://github.com/yiisoft/yii2-gii/blob/master/docs/guide-pt-BR/README.md>

⁴¹<https://www.yiiframework.com/doc-2.0/guide-README.html>

⁴²<https://www.yiiframework.com/doc-2.0/index.html>

⁴³<https://www.yiiframework.com/wiki/?tag=yii2>

escritos como receitas de bolo e mostram como resolver problemas específicos usando o Yii. Ainda que a qualidade desses artigos possa não ser tão boa quanto a do Guia Definitivo, eles são úteis porque são abrangem assuntos adicionais e frequentemente fornecem soluções prontas para serem usadas.

- Livros⁴⁴
- Extensões⁴⁵: O Yii ostenta uma biblioteca de milhares de extensões contribuídas por usuários que podem facilmente ser plugadas em suas aplicações, tornando o seu desenvolvimento ainda mais rápido e mais fácil.
- Comunidade
 - Fórum: <https://forum.yiiframework.com/>
 - Chat IRC: O canal #yii na rede Libera (<ircs://irc.libera.chat:6697/yii>)
 - Canal Slack: <https://yii.slack.com>
 - Chat Gitter: <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>

⁴⁴<https://www.yiiframework.com/books>

⁴⁵<https://www.yiiframework.com/extensions/>

Capítulo 3

Estrutura da Aplicação

3.1 Visão Geral

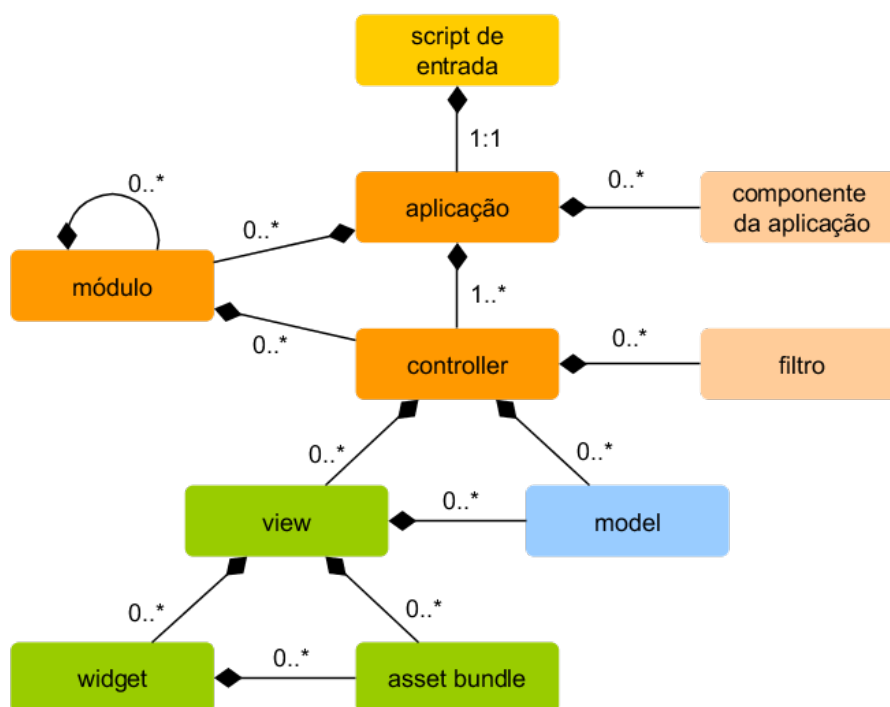
As aplicações do Yii são organizadas de acordo com o padrão de projeto model-view-controller (MVC)¹ (modelo-visão-controlador). Os **models** representam dados, lógica e regras de negócio; as **views** são a representação da saída dos modelos; e os **controllers** recebem entradas e as convertem em comandos para os **models** e as **views**.

Além do MVC, as aplicações do Yii também possuem as seguintes entidades:

- **scripts de entrada**: são scripts PHP que são diretamente acessíveis aos usuários finais. São responsáveis por iniciar o ciclo de tratamento de uma requisição.
- **aplicações**: são objetos globalmente acessíveis que gerenciam os componentes da aplicação e os coordenam para atender às requisições.
- **componentes da aplicação**: são objetos registrados com as aplicações e fornecem vários serviços para atender às requisições.
- **módulos**: são pacotes auto-contidos que contém um MVC completo por si sós. Uma aplicação pode ser organizada em termos de múltiplos módulos.
- **filtros**: representam código que precisa ser chamado pelos controllers antes e depois do tratamento propriamente dito de cada requisição.
- **widgets**: são objetos que podem ser embutidos em **views**. Podem conter lógica de controller e podem ser reutilizados em diferentes views.

O diagrama a seguir demonstra a estrutura estática de uma aplicação:

¹<https://pt.wikipedia.org/wiki/MVC>



3.2 Scripts de Entrada

Scripts de entrada são o primeiro passo no processo de inicialização da aplicação. Uma aplicação (seja uma aplicação Web ou uma aplicação console) possui um único script de entrada. Os usuários finais fazem requisições nos scripts de entrada que criam as instâncias da aplicação e redirecionam as requisições para elas.

Os scripts de entrada para aplicações Web devem estar armazenados em diretórios acessíveis pela Web, de modo que eles possam ser acessados pelos usuários finais. Frequentemente são chamados de `index.php`, mas também podem usar outros nomes, desde que os servidores Web consigam localizá-los.

Os scripts de entrada para aplicações do console são geralmente armazenados no `caminho base` das aplicações e são chamados de `yii` (com o sufixo `.php`). Eles devem ser tornados executáveis para que os usuários possam executar aplicações do console através do comando `./yii <rota> [argumentos] [opções]`.

O trabalho principal dos scripts de entrada é o seguinte:

- Definir constantes globais;
- Registrar o autoloader do Composer²;

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

- Incluir o arquivo da classe `Yii`;
- Carregar a configuração da aplicação;
- Criar e configurar uma instância da aplicação;
- Chamar `yii\base\Application::run()` para processar as requisições que chegam.

3.2.1 Aplicações Web

Este é o código no script de entrada para o [Template Básico de Projetos](#).

```
<?php

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

// registra o autoloader do Composer
require __DIR__ . '/../vendor/autoload.php';

// inclui o arquivo da classe Yii
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

// carrega a configuração da aplicação
$config = require __DIR__ . '/../config/web.php';

// cria, configura e executa a aplicação
(new yii\web\Application($config))->run();
```

3.2.2 Aplicações Console

De forma semelhante, o seguinte é o código do script de entrada de uma aplicação do 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);

// registra o autoloader do Composer
require __DIR__ . '/vendor/autoload.php';

// inclui o arquivo da classe Yii
require __DIR__ . '/vendor/yiisoft/yii2/Yii.php';

// carrega a configuração da aplicação
$config = require __DIR__ . '/config/console.php';
```

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

3.2.3 Definindo Constantes

Os scripts de entrada são o melhor lugar para definir as constantes globais. O Yii suporta as seguintes três constantes:

- `YII_DEBUG`: especifica se a aplicação está rodando no modo de depuração. No modo de depuração, uma aplicação manterá mais informações de log, e revelará stacks de chamadas de erros detalhadas se forem lançadas exceções. Por este motivo, o modo de depuração deveria ser usado principalmente durante o desenvolvimento. O valor padrão de `YII_DEBUG` é `false`.
- `YII_ENV`: especifica em qual ambiente a aplicação está rodando. Isso foi descrito em maiores detalhes na seção [Configurações](#). O valor padrão de `YII_ENV` é `'prod'`, significando que a aplicação está executando em ambiente de produção.
- `YII_ENABLE_ERROR_HANDLER`: especifica se deve ativar o manipulador de erros fornecido pelo Yii. O valor padrão desta constante é `true`.

Ao definir uma constante, frequentemente usamos código como o a seguir:

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

que é equivalente ao seguinte código:

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

Claramente o primeiro é mais sucinto e fácil de entender.

A definição de constantes deveria ser feita logo no início de um script de entrada, de modo que obtenha efeito quando outros arquivos PHP estiverem sendo inclusos.

3.3 Aplicações

Aplicações são objetos que regem a estrutura e ciclo de vida gerais de aplicações em Yii. Cada aplicação contém um único objeto `Application` que é criado no [script de entrada](#) e que pode ser acessado globalmente pela expressão `\Yii::$app`.

Informação: Dependendo do contexto, quando dizemos “uma aplicação”, pode significar tanto um objeto `Application` quanto um sistema.

Existem dois tipos de aplicações: `aplicações Web` e `aplicações console`. Como o próprio nome indica, o primeiro manipula requisições Web enquanto o segundo trata requisições de comandos do console.

3.3.1 Configurações da Aplicação

Quando um `script de entrada` cria uma aplicação, ele carregará uma `configuração` e a aplicará à aplicação, da seguinte forma:

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

// carrega a configuração da aplicação
$config = require __DIR__ . '/../config/web.php';

// instancia e configura a aplicação
(new yii\web\Application($config))->run();
```

Tal como `configurações` normais, as configurações da aplicação especificam como inicializar as propriedades de objetos `Application`. Uma vez que geralmente são muito complexas, elas normalmente são mantidas em `arquivos de configuração`, como o arquivo `web.php` no exemplo acima.

3.3.2 Propriedades da Aplicação

Existem muitas propriedades importantes da aplicação que deveriam ser configuradas. Essas propriedades tipicamente descrevem o ambiente em que as aplicações estão rodando. Por exemplo, as aplicações precisam saber como carregar os `controllers`, onde armazenar os arquivos temporários, etc. A seguir resumiremos essas propriedades.

Propriedades Obrigatórias

Em qualquer aplicação, você deve pelo menos configurar duas propriedades: `id` e `basePath`.

id A propriedade `id` especifica um ID único que diferencia uma aplicação das outras. É usado principalmente programaticamente. Apesar de não ser obrigatório, para melhor interoperabilidade recomenda-se que você só use caracteres alfanuméricos ao especificar um ID de aplicação.

basePath A propriedade `basePath` especifica o diretório raiz de um sistema. É o diretório que contém todo o código fonte protegido de um sistema. Sob este diretório, você normalmente verá subdiretórios tais como `models`, `views` e `controllers`, que contém o código fonte correspondente ao padrão MVC.

Você pode configurar a propriedade `basePath` usando um [alias de caminho](#). Em ambas as formas, o diretório correspondente precisa existir, doutra forma será lançada uma exceção. O caminho será normalizado chamando-se a função `realpath()`.

A propriedade `basePath` frequentemente é usada para derivar outros caminhos importantes (por exemplo, o diretório de runtime). Por esse motivo, um alias de caminho `@app` é pré-definido para representar esse caminho. Assim os caminhos derivados podem ser formados usando esse alias (por exemplo, `@app/runtime` para referenciar o diretório runtime).

Propriedades Importantes

As propriedades descritas nesta subseção frequentemente precisam ser configuradas porque elas variam em diferentes aplicações.

aliases Esta propriedade permite que você defina um conjunto de [aliases](#) em termos de um array. As chaves do array representam os nomes de alias, e os valores são as definições correspondentes. Por exemplo:

```
[
    'aliases' => [
        '@name1' => 'path/to/path1',
        '@name2' => 'path/to/path2',
    ],
]
```

Esta propriedade é fornecida para que você possa definir aliases na configuração da aplicação ao invés de chamar o método `Yii::setAlias()`.

bootstrap Esta é uma propriedade muito útil. Ela permite que você especifique um array de componentes que devem ser executados durante o [processo de inicialização](#) da aplicação. Por exemplo, se você quer que um [módulo](#) personalize as [regras de URL](#), você pode listar seu ID como um elemento nesta propriedade.

Cada componente listado nesta propriedade deve ser especificado em um dos seguintes formatos:

- o ID de um componente de aplicação conforme especificado via `components`.
- o ID de um módulo conforme especificado via `modules`.
- o nome de uma classe.
- um array de configuração.
- uma função anônima que cria e retorna um componente.

Por exemplo:

```
[
    'bootstrap' => [
```



```

        // o ID de uma aplicação ou de um módulo
        'demo',

        // um nome de classe
        'app\components\Profiler',

        // um array de configuração
        [
            'class' => 'app\components\Profiler',
            'level' => 3,
        ],

        // uma função anônima
        function () {
            return new app\components\Profiler();
        }
    ],
]

```

Informação: Se o ID de um módulo é o mesmo que o ID de um componente da aplicação, o componente será usado durante o processo de inicialização. Se você quiser usar o módulo ao invés dele, você pode especificá-lo usando uma função anônima conforme a seguir: `php [

```

function () {
    return Yii::$app->getModule('user');
},

] `

```

Durante o processo de inicialização, cada componente será instanciado. Se a classe do componente implementa `yii\base\BootstrapInterface`, seu método `bootstrap()` também será chamado.

Outro exemplo prático está na configuração do [Template Básico de Projetos](#), onde os módulos `debug` e `gii` estão configurados como componentes de inicialização quando a aplicação está rodando no ambiente de desenvolvimento:

```

if (YII_ENV_DEV) {
    // ajustes da configuração para o ambiente 'dev'
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';

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

```

Observação: Colocar componentes demais em `bootstrap` degradará o desempenho de sua aplicação, porque para cada requisição o mesmo conjunto de componentes precisará ser carregado. Desta forma, use os componentes de inicialização com juízo.

catchAll Essa propriedade só é suportada por aplicações Web. Ela especifica uma [action de um controller](#) que deve manipular todas as requisições. Isso é geralmente usado quando a aplicação está em modo de manutenção e precisa tratar todas as requisições através de uma única action.

A configuração é um array, cujo primeiro elemento especifica a rota para a action. O restante dos elementos do array (pares de chave-valor) especificam os parâmetros que devem ser atrelados à action. Por exemplo:

```
[
    'catchAll' => [
        'offline/notice',
        'param1' => 'value1',
        'param2' => 'value2',
    ],
]
```

components Essa é a propriedade mais importante. Ela permite que você registre uma lista de componentes chamados [componentes de aplicação](#) que você pode usar em outros lugares. Por exemplo:

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

Cada componente da aplicação é especificado como um par de chave-valor em um array. A chave representa o ID do componente, enquanto o valor representa o nome ou a [configuração](#) da classe do componente.

Você pode registrar qualquer componente com uma aplicação, e o componente depois poderá ser acessado globalmente através da expressão `\Yii::$app->IDdoComponente`.

Por favor leia a seção [Componentes de Aplicação](#) para mais detalhes.

controllerMap Essa propriedade permite que você mapeie um ID de um controller a uma classe de controller arbitrária. Por padrão, o Yii mapeia os IDs de controllers a classes de controllers baseado em uma convenção (por exemplo, o ID `post` será mapeado para `app\controllers\PostController`). Ao configurar essa propriedade, você pode quebrar a convenção de controllers em específico. No exemplo a seguir, `account` será mapeado para `app\controllers\UserController`, enquanto `article` será mapeado para `app\controllers\PostController`.

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

As chaves do array dessa propriedade representam os IDs dos controllers, enquanto os valores do array representam o nome ou as [configurações](#) da classe do controller.

controllerNamespace Essa propriedade especifica o namespace padrão sob o qual as classes dos controllers deverão ser localizadas. Seu valor padrão é `app\controllers`. Se um ID de um controller for `post`, por convenção o nome da classe de controller correspondente (sem namespace) seria `PostController`, e o nome da classe completo e qualificado seria `app\controllers\PostController`.

As classes de controllers também podem estar localizadas em subdiretórios do diretório correspondente ao namespace. Por exemplo, dado um ID de controller `admin/post`, a classe completa e qualificada correspondente seria `app\controllers\admin\PostController`.

É importante que as classes completas e qualificadas possam ser [carregadas automaticamente](#) e que o namespace das suas classes de controller correspondam ao valor dessa propriedade. Doutra forma, você receberia um erro de “Página Não Encontrada” ao acessar a aplicação.

Caso você queira quebrar a convenção conforme descrito acima, você pode configurar a propriedade `controllerMap`.

language Essa propriedade especifica o idioma no qual a aplicação deve exibir o conteúdo aos usuários finais. O valor padrão dessa propriedade é `en`, significando inglês. Você deve configurar essa propriedade se a sua aplicação suportar múltiplos idiomas.

O valor dessa propriedade determina vários aspectos da [internacionalização](#), incluindo tradução de mensagens, formato de datas, formato de números, etc. Por exemplo, o widget `yii\jui\DatePicker` usará o valor dessa propriedade por padrão para determinar em qual idioma o calendário deverá ser exibido e como a data deve ser formatada.

Recomenda-se que você especifique um idioma em termos de um código de idioma IETF³. Por exemplo, `en` corresponde ao inglês, enquanto `en-US` significa inglês dos Estados Unidos.

Mais detalhes sobre essa propriedade podem ser encontrados na seção [Internacionalização](#).

³https://en.wikipedia.org/wiki/IETF_language_tag

modules Essa propriedade especifica os [módulos](#) que uma aplicação contém.

A propriedade recebe um array de classes de módulos ou [configurações](#) com as chaves do array sendo os IDs dos módulos. Por exemplo:

```
[
    'modules' => [
        // um módulo "booking" especificado com a classe do módulo
        'booking' => 'app\modules\booking\BookingModule',

        // um módulo "comment" especificado com um array de configurações
        'comment' => [
            'class' => 'app\modules\comment\CommentModule',
            'db' => 'db',
        ],
    ],
]
```

Por favor consulte a seção [Módulos](#) para mais detalhes.

name Essa propriedade especifica o nome da aplicação que pode ser exibido aos usuários finais. Ao contrário da propriedade `id` que deveria receber um valor único, o valor desta propriedade serve principalmente para fins de exibição e não precisa ser único.

Você nem sempre precisa configurar essa propriedade se nenhuma parte do código a estiver usando.

params Essa propriedade especifica um array de parâmetros da aplicação que podem ser acessados globalmente. Ao invés de usar números e strings fixos espalhados por toda parte no seu código, é uma boa prática defini-los como parâmetros da aplicação em um único lugar e usá-los nos lugares onde for necessário. Por exemplo, você pode definir o tamanho de uma miniatura de imagem como um parâmetro conforme a seguir:

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

Então no seu código onde você precisar usar o valor do tamanho, você pode simplesmente usar o código conforme a seguir:

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

Mais tarde, se você decidir mudar o tamanho da miniatura, você só precisa modificá-lo na configuração da aplicação sem tocar em quaisquer códigos dependentes.

sourceLanguage Essa propriedade especifica o idioma no qual o código da aplicação foi escrito. O valor padrão é `'en-US'`, significando inglês dos Estados Unidos. Você deve configurar essa propriedade se o conteúdo do texto no seu código não estiver em inglês.

Conforme a propriedade `language`, você deve configurar essa propriedade em termos de um código de idioma IETF⁴. Por exemplo, `en` corresponde ao inglês, enquanto `en-US` significa inglês dos Estados Unidos.

Mais detalhes sobre essa propriedade podem ser encontrados na seção [Internacionalização](#).

timeZone Essa propriedade é disponibilizada como uma maneira alternativa de definir a timezone do PHP em tempo de execução. Ao configurar essa propriedade, você está essencialmente chamando a função `date_default_timezone_set()`⁵ do PHP. Por exemplo:

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

version Essa propriedade especifica a versão da aplicação. Seu valor padrão é `'1.0'`. Você não precisa configurar esta propriedade se nenhuma parte do seu código estiver utilizando-a.

Propriedades Úteis

As propriedades descritas nesta subseção não são comumente configuradas porque seus valores padrão estipulam convenções comuns. No entanto, você pode ainda configurá-las no caso de querer quebrar as convenções.

charset Essa propriedade especifica o charset que a aplicação usa. O valor padrão é `'UTF-8'`, que deveria ser mantido como está para a maioria das aplicações, a menos que você esteja trabalhando com sistemas legados que usam muitos dados que não são unicode.

defaultRoute Essa propriedade especifica a [rota](#) que uma aplicação deveria usar quando uma requisição não especifica uma. A rota pode consistir de um ID de módulo, ID de controller e/ou ID de action. Por exemplo, `help`, `post/create`, `admin/post/create`. Se não for passado um ID de action, ele assumirá o valor conforme especificado em `yii\base\Controller::$defaultAction`.

Para aplicações Web, o valor padrão dessa propriedade é `'site'`, o que significa que deve usar o controller `SiteController` e sua action padrão. Como

⁴https://en.wikipedia.org/wiki/IETF_language_tag

⁵<https://www.php.net/manual/en/function.date-default-timezone-set.php>

resultado disso, se você acessar a aplicação sem especificar uma rota, ele exibirá o resultado de `app\controllers\SiteController::actionIndex()`.

Para aplicações do console, o valor padrão é `'help'`, o que significa que deve usar o comando do core `yii\console\controllers\HelpController::actionIndex()`. Como resultado, se você executar o comando `yii` sem fornecer quaisquer argumentos, ele exibirá a informação de ajuda.

extensions Essa propriedade especifica a lista de [extensões](#) que estão instaladas e são usadas pela aplicação. Por padrão, ela receberá o array retornado pelo arquivo `@vendor/yiisoft/extensions.php`. O arquivo `extensions.php` é gerado e mantido automaticamente quando você usa o Composer⁶ para instalar extensões. Então na maioria dos casos você não precisa configurar essa propriedade.

No caso especial de você querer manter extensões manualmente, você pode configurar essa propriedade da seguinte forma:

```
[
    'extensions' => [
        [
            'name' => 'extension name',
            'version' => 'version number',
            'bootstrap' => 'BootstrapClassName', // opcional, também pode
            ser um array de configuração
            'alias' => [ // opcional
                '@alias1' => 'to/path1',
                '@alias2' => 'to/path2',
            ],
        ],
        // ... mais extensões conforme acima ...
    ],
]
```

Como você pode ver, a propriedade recebe um array de especificações de extensões. Cada extensão é especificada com um array composto pelos elementos `name` e `version`. Se uma extensão precisa executar durante o [processo de inicialização](#), um elemento `bootstrap` pode ser especificado com um nome de uma classe de inicialização ou um array de [configuração](#). Uma extensão também pode definir alguns [alias](#)es.

layout Essa propriedade especifica o nome do layout padrão que deverá ser usado ao renderizar uma [view](#). O valor padrão é `'main'`, significando que o arquivo de layout `main.php` sob o caminho dos layouts deverá ser usado. Se tanto o caminho do layout quanto o caminho da view estiverem recebendo

⁶<https://getcomposer.org>

os valores padrão, o arquivo de layout padrão pode ser representado como o alias de caminho `@app/views/layouts/main.php`.

Você pode configurar esta propriedade como `false` se você quiser desativar o layout por padrão, embora isso seja muito raro.

layoutPath Essa propriedade especifica o caminho onde os arquivos de layout devem ser procurados. O valor padrão é o subdiretório `layouts` dentro do diretório do caminho das views. Se o caminho das views estiver recebendo seu valor padrão, o caminho padrão dos layouts pode ser representado como o alias de caminho `@app/views/layouts`.

Você pode configurá-la como um diretório ou um [alias](#) de caminho.

runtimePath Essa propriedade especifica o caminho onde os arquivos temporários, tais como arquivos de log e de cache, podem ser gerados. O valor padrão é o diretório representado pelo alias `@app/runtime`.

Você pode configurá-la como um diretório ou [alias](#) de caminho. Perceba que o caminho de runtime precisa ter permissão de escrita para o processo que executa a aplicação. E o caminho deveria ser protegido para não ser acessado pelos usuários finais, porque os arquivos temporários dentro dele podem conter informações sensíveis.

Para simplificar o acesso a esse caminho, o Yii possui um alias de caminho pré-definido chamado `@runtime` para ele.

viewPath Essa propriedade especifica o diretório raiz onde os arquivos de views estão localizados. O valor padrão do diretório é representado pelo alias `@app/views`. Você pode configurá-lo como um diretório ou [alias](#) de caminho.

vendorPath Essa propriedade especifica o diretório vendor gerenciado pelo Composer⁷. Ele contém todas as bibliotecas de terceiros usadas pela sua aplicação, incluindo o framework do Yii. O valor padrão é o diretório representado pelo alias `@app/vendor`.

Você pode configurar essa propriedade como um diretório ou [alias](#) de caminho. Quando você modificar essa propriedade, assegure-se de ajustar a configuração do Composer de acordo.

Para simplificar o acesso a esse caminho, o Yii tem um alias de caminho pré-definido para ele chamado de `@vendor`.

enableCoreCommands Essa propriedade só é suportada por aplicações do console. Ela especifica se os comandos do core inclusos no pacote do Yii devem estar ativos. O valor padrão é `true`.

⁷<https://getcomposer.org>

3.3.3 Eventos da Aplicação

Uma aplicação dispara muitos eventos durante o ciclo de vida de manipulação de uma requisição. Você pode vincular manipuladores a esses eventos nas configurações da aplicação do seguinte modo,

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

A sintaxe de uso de `on eventName` é descrita na seção [Configurações](#).

Alternativamente, você pode vincular manipuladores de evento durante o processo de inicialização após a instância da aplicação ser criada. Por exemplo:

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

EVENT_BEFORE_REQUEST

Este evento é disparado *antes* de uma aplicação manipular uma requisição. O nome do evento é `beforeRequest`.

Quando esse evento é disparado, a instância da aplicação foi configurada e inicializada. Então é um bom lugar para inserir código personalizado por meio do mecanismo de eventos para interceptar o processo de tratamento da requisição. Por exemplo, no manipulador de eventos, você pode definir dinamicamente a propriedade `\yii\base\Application::$language` baseado em alguns parâmetros.

EVENT_AFTER_REQUEST

Este evento é disparado *depois* que uma aplicação finaliza o tratamento da requisição, mas *antes* de enviar a resposta. O nome do evento é `afterRequest`.

Quando este evento é disparado, o tratamento da requisição está completo e você pode aproveitar essa ocasião para fazer um pós-processamento da requisição ou personalizar a resposta.

Perceba que o componente `response` também dispara alguns eventos enquanto está enviando conteúdo de resposta para os usuários finais. Esses eventos são disparados *depois* deste evento.

EVENT_BEFORE_ACTION

Este evento é disparado *antes* de executar cada *action de controller*. O nome do evento é `beforeAction`.

O parâmetro do evento é uma instância de `yii\base\ActionEvent`. Um manipulador de evento pode definir o valor da propriedade `yii\base\ActionEvent::$isValid` como `false` para interromper a execução da action. Por exemplo:

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

Perceba que o mesmo evento `beforeAction` também é disparado pelos `módulos` e `controllers`. Os objetos `Application` são os primeiros a disparar este evento, seguidos pelos módulos (se houver algum) e finalmente pelos controllers. Se um manipulador de evento definir `yii\base\ActionEvent::$isValid` como `false`, todos os eventos seguintes NÃO serão disparados.

EVENT_AFTER_ACTION

Este evento é disparado *depois* de executar cada `action de controller`. O nome do evento é `afterAction`.

O parâmetro do evento é uma instância de `yii\base\ActionEvent`. Através da propriedade `yii\base\ActionEvent::$result`, um manipulador de evento pode acessar ou modificar o resultado da action. Por exemplo:

```
[
    'on afterAction' => function ($event) {
        if (alguma condição) {
            // modifica $event->result
        } else {
        }
    },
]
```

Perceba que o mesmo evento `afterAction` também é disparado pelos `módulos` e `controllers`. Estes objetos disparam esse evento na order inversa da do `beforeAction`. Ou seja, os controllers são os primeiros objetos a disparar este evento, seguidos pelos módulos (se houver algum) e finalmente pelos objetos `Application`.

3.3.4 Ciclo de Vida da Aplicação

Quando um `script de entrada` estiver sendo executado para manipular uma requisição, uma aplicação passará pelo seguinte ciclo de vida:

1. O script de entrada carrega a configuração da aplicação como um array.

2. O script de entrada cria uma nova instância da aplicação:
 - `preInit()` é chamado, que configura algumas propriedades da aplicação de alta prioridade, tais como `basePath`.
 - Registra o **manipulador de erros**.
 - Configura as propriedades da aplicação.
 - `init()` é chamado, que por sua vez chama `bootstrap()` para rodar os componentes de inicialização.
3. O script de entrada chama `yii\base\Application::run()` para executar a aplicação:
 - Dispara o evento `EVENT_BEFORE_REQUEST`.
 - Trata a requisição: resolve a requisição em uma **rota** e os parâmetros associados; cria os objetos do módulo, do controller e da action conforme especificado pela rota; e executa a action.
 - Dispara o evento `EVENT_AFTER_REQUEST`.
 - Envia a resposta para o usuário final.
4. O script de entrada recebe o status de saída da aplicação e completa o processamento da requisição.

3.4 Componentes de Aplicação

Aplicações são **service locators**. Elas hospedam um conjunto de assim chamados *componentes de aplicação* que fornecem diferentes serviços para o processamento de requisições. Por exemplo, o componente `urlManager` é responsável pelo roteamento de requisições Web aos controllers adequados; o componente `db` fornece serviços relacionados a bancos de dados; e assim por diante.

Cada componente de aplicação tem um ID que o identifica de maneira única dentre os outros componentes de uma mesma aplicação. Você pode acessar um componente de aplicação através da expressão

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

Por exemplo, você pode usar `\Yii::$app->db` para obter a **conexão do BD**, e `\Yii::$app->cache` para obter o **cache primário** registrado com a aplicação.

Um componente de aplicação é criado na primeira vez em que é acessado através da expressão acima. Quaisquer acessos posteriores retornarão a mesma instância do componente.

Componentes de aplicação podem ser quaisquer objetos. Você pode registrá-los configurando a propriedade `yii\base\Application::$components` nas **configurações da aplicação**. Por exemplo,

```
[
    'components' => [
        // registra o componente "cache" usando um nome de classe
        'cache' => 'yii\caching\ApcCache',

        // registra o componente "db" usando um array de configuração
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],

        // registra o componente "search" usando uma função anônima
        'search' => function () {
            return new app\components\SolrService;
        },
    ],
]
```

Informação: Embora você possa registrar quantos componentes de aplicação você quiser, você deveria fazer isso com juízo. Componentes de aplicação são como variáveis globais. Usar componentes de aplicação demais pode tornar seu código potencialmente mais difícil de testar e manter. Em muitos casos, você pode simplesmente criar um componente local e utilizá-lo quando necessário.

3.4.1 Components de Inicialização

Conforme mencionado acima, um componente de aplicação só será instanciado quando ele estiver sendo acessado pela primeira vez. Se ele nunca for acessado durante uma requisição, ele não será instanciado. No entanto, algumas vezes você pode querer instanciar um componente de aplicação em todas as requisições, mesmo que ele não seja explicitamente acessado. Para fazê-lo, você pode listar seu ID na propriedade `bootstrap` da aplicação.

Por exemplo, a configuração de aplicação a seguir assegura-se que o componente `log` sempre esteja carregado:

```
[
    'bootstrap' => [
        'log',
    ],
    'components' => [
        'log' => [
            // configuração para o componente "log"
        ],
    ],
]
```

3.4.2 Componentes de Aplicação do Core

O yii define um conjunto de componentes de aplicação do **core** com IDs fixos e configurações padrão. Por exemplo, o componente **request** é usado para coletar as informações sobre uma requisição do usuário e resolvê-la em uma **rota**; o componente **db** representa uma conexão do banco de dados através da qual você pode realizar consultas. É com a ajuda destes componentes de aplicação do core que as aplicações Yii conseguem tratar as requisições dos usuários.

Segue abaixo uma lista dos componentes de aplicação pré-definidos do core. Você pode configurá-los e personalizá-los como você faz com componentes de aplicação normais. Quando você estiver configurando um componente de aplicação do core, se você não especificar sua classe, a padrão será utilizada.

- **assetManager**: gerencia os asset bundles e a publicação de assets. Por favor consulte a seção [Gerenciando Assets](#) para mais detalhes.
- **db**: representa uma conexão do banco de dados através da qual você poderá realizar consultas. Perceba que quando você configura esse componente, você precisa especificar a classe do componente bem como as outras propriedades obrigatórias, tais como `yii\db\Connection::$dsn`. Por favor consulte a seção [Data Access Objects](#) (Objeto de Acesso a Dados) para mais detalhes.
- **errorHandler**: manipula erros e exceções do PHP. Por favor consulte a seção [Tratamento de Erros](#) para mais detalhes.
- **formatter**: formata dados quando são exibidos aos usuários finais. Por exemplo, um número pode ser exibido com um separador de milhares, uma data pode ser formatada em um formato longo. Por favor consulte a seção [Formatação de Dados](#) para mais detalhes.
- **i18n**: suporta a tradução e formatação de mensagens. Por favor consulte a seção [Internacionalização](#) para mais detalhes.
- **log**: gerencia alvos de logs. Por favor consulte a seção [Gerenciamento de Logs](#) para mais detalhes.
- **yii\swiftmailer\Mailer**: suporta a composição e envio de e-mails. Por favor consulte a seção [Enviando E-mails](#) para mais detalhes.
- **response**: representa a resposta sendo enviada para os usuários finais. Por favor consulte a seção [Respostas](#) para mais detalhes.
- **request**: representa a requisição recebida dos usuários finais. Por favor consulte a seção [Requisições](#) para mais detalhes.
- **session**: representa as informações da sessão. Esse componente só está disponível em [aplicações Web](#). Por favor consulte a seção [Sessões e Cookies](#) para mais detalhes.
- **urlManager**: suporta a análise e criação de URLs. Por favor consulte a seção [Análise e Geração de URLs](#) para mais detalhes.
- **user**: representa as informações de autenticação do usuário. Esse com-

ponente só está disponível em aplicações Web. Por favor consulte a seção [Autenticação](#) para mais detalhes.

- **view**: suporta a renderização de views. Por favor consulte a seção [Views](#) para mais detalhes.

3.5 Controllers (Controladores)

Os controllers (controladores) fazem parte da arquitetura MVC⁸. São objetos de classes que estendem de `yii\base\Controller` e são responsáveis pelo processamento das requisições e por gerar respostas. Em particular, após assumir o controle de [applications](#), controllers analisarão os dados de entrada obtidos pela requisição, passarão estes dados para os [models](#) (modelos), incluirão os resultados dos models (modelos) nas [views](#) (visões) e finalmente gerarão as respostas de saída.

3.5.1 Actions (Ações)

Os controllers são compostos por unidades básicas chamadas de *ações* que podem ser tratados pelos usuários finais a fim de realizar a sua execução.

No exemplo a seguir mostra um controller `post` com duas ações: `view` e `create`:

```
namespace app\controllers;

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

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

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

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

⁸<https://pt.wikipedia.org/wiki/MVC>

```

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

```

Na ação `view` (definido pelo método `actionView()`), o primeiro código carrega o `model` conforme o ID solicitado; Se o model for devidamente carregado, a ação irá exibi-lo utilizando a `view` chamada de `view`. Caso contrário, a ação lançará uma exceção.

Na ação `create` (definido pelo método `actionCreate()`), o código é parecido. Primeiro ele tenta popular o `model` usando os dados da requisição em seguida os salva. Se ambos forem bem sucedidos, a ação redirecionará o navegador para a ação `view` com o novo ID criado pelo model. Caso contrário, a ação exibirá a `view create` na qual os usuário poderão fornecer os dados necessários.

3.5.2 Routes (Rotas)

Os usuários finais abordarão as ações por meio de *rotas*. Uma rota é uma string composta pelas seguintes partes:

- um ID do módulo: serve apenas se o controller pertencer a um `módulo` que não seja da aplicação;
- um ID do controller: uma string que identifica exclusivamente o controller dentre todos os controllers da mesma aplicação (ou do mesmo módulo, caso o controller pertença a um módulo);
- um ID da ação: uma string que identifica exclusivamente uma ação dentre todas as ações de um mesmo controller.

As rotas seguem o seguinte formato:

IDdoController/IDdoAction

ou o seguinte formato se o controller estiver em um módulo:

IDdoModule/IDdoController/IDdoAction

Portanto, se um usuário fizer uma requisição com a URL `https://hostname/index.php?r=site/index`, a ação `index` do controller `site` será executada. Para mais detalhes sobre como as ações são resolvidas pelas rotas, por favor consulte a seção [Roteamento e Criação de URL](#).

3.5.3 Criando Controllers

Em aplicações Web, os controllers devem estender de `yii\web\Controller` ou de suas classes filhas. De forma semelhante, em aplicações console, os controllers devem estender de `yii\console\Controller` ou de suas classes filhas. O código a seguir define um controller `site`:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
}
```

IDs de Controllers

Normalmente, um controller é projetado para tratar as requisições relativos a um determinado tipo de recurso. Por esta razão, os IDs dos controllers geralmente são substantivos que referenciam-se ao tipo de recurso que será tratado. Por exemplo, você pode usar o `article` como o ID do um controller para tratar dados de artigos.

Por padrão, os IDs dos controllers devem conter apenas esses caracteres: letras inglesas em caixa baixa, números, underscores (underline), hífens e barras. Por exemplo, `article` e `post-comment` são ambos IDs de controllers válidos, enquanto `article?`, `PostComment`, `admin\post` não são.

Um ID de controller também pode conter um prefixo para o subdiretório. Por exemplo, `admin/article` representa um controller `article` em um subdiretório `admin` sob o namespace do controller. Os caracteres válidos para os prefixos de subdiretórios incluem: letras inglesas em caixa alto ou caixa baixa, números, underscores (underline) e barras, onde as barras são usadas para separar os níveis dos subdiretórios (por exemplo, `panels/admin`).

Nomenclatura da Classe do Controller

Os nomes das classes dos controllers podem ser derivadas dos IDs dos controllers de acordo com as seguintes procedimentos:

1. Colocar em caixa alta a primeira letra de cada palavra separadas por traço. Observe que se o ID do controller possuir barras, a regra é aplicada apenas na parte após a última barra no ID.
2. Remover os traços e substituir todas as barras por barras invertidas.
3. Adicionar `Controller` como sufixo.
4. Preceder ao namespace do controller.

Segue alguns exemplos, assumindo que o namespace do controller tenha por padrão o valor `app\controllers`:

- `article` torna-se `app\controllers\ArticleController`;
- `post-comment` torna-se `app\controllers\PostCommentController`;
- `admin/post-comment` torna-se `app\controllers\admin\PostCommentController`;
- `adminPanels/post-comment` torna-se `app\controllers\adminPanels\PostCommentController`.

As classes dos controllers devem ser [autoloadable](#). Por esta razão, nos exemplos anteriores, o controller `article` deve ser salvo no arquivo cuja [alias](#) é `@app/controllers/ArticleController.php`; enquanto o controller `admin/post-comment` deve ser salvo no `@app/controllers/admin/PostCommentController.php`.

Informação: No último exemplo `admin/post-comment`, mostra como você pode colocar um controller em um subdiretório do namespace `controller`. Isto é útil quando você quiser organizar seus controllers em diversas categorias e não quiser usar [módulos](#).

Mapeando Controllers

Você pode configurar um mapeamento de controllers para superar as barreiras impostas pelos IDs de controllers e pelos nomes de classes descritos acima. Isto é útil principalmente quando quiser esconder controllers de terceiros na qual você não tem controle sobre seus nomes de classes.

Você pode configurar o mapeamento de controllers na [configuração da aplicação](#). Por exemplo:

```
[
    'controllerMap' => [
        // declara o controller "account" usando um nome de classe
        'account' => 'app\controllers\UserController',

        // declara o controller "article" usando uma configuração em array
        'article' => [
            'class' => 'app\controllers\PostController',
            'enableCsrfValidation' => false,
        ],
    ],
]
```

Controller Padrão

Cada aplicação tem um controller padrão que é especificado pela propriedade `yii\base\Application::$defaultRoute`. Quando uma requisição não especificar uma rota, será utilizada a rota especificada pela propriedade. Para as aplicações Web, este valor é `'site'`, enquanto para as aplicações console é `help`. Portanto, se uma URL for `https://hostname/index.php`, o controller `site` será utilizado nesta requisição.

Você pode alterar o controller padrão como a seguinte [configuração da aplicação](#):


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

3.5.4 Criando Ações

Criar ações pode ser tão simples como a definição dos chamados *métodos de ação* em uma classe controller. Um método de ação é um método *público* cujo nome inicia com a palavra *action*. O valor de retorno representa os dados de resposta a serem enviados aos usuário finais. O código a seguir define duas ações, `index` e `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';  
    }  
}
```

IDs de Actions

Uma ação muitas vezes é projetada para realizar uma manipulação em particular sobre um recurso. Por esta razão, os IDs das ações geralmente são verbos, tais como `view`, `update`, etc.

Por padrão, os IDs das ações devem conter apenas esses caracteres: letras inglesas em caixa baixa, números, underscores (underline) e traços. Os traços em um ID da ação são usados para separar palavras. Por exemplo, `view`, `update2` e `comment-post` são IDs válidos, enquanto `view?` e `Update` não são.

Você pode criar ações de duas maneiras: ações inline (em sequência) e ações standalone (autônomas). Uma ação inline é definida pelo método de uma classe controller, enquanto uma ação standalone é uma classe que estende de `yii\base\Action` ou de uma classe-filha. As ações inline exigem menos esforço para serem criadas e muitas vezes as preferidas quando não se tem a intenção de reutilizar estas ações. Ações standalone, por outro lado, são criados principalmente para serem utilizados em diferentes controllers ou para serem distribuídos como [extensions](#).

Ações Inline

As ações inline referem-se a os chamados métodos de ação, que foram descritos anteriormente.

Os nomes dos métodos de ações são derivadas dos IDs das ações de acordo com os seguintes procedimentos:

1. Colocar em caixa alta a primeira letra de cada palavra do ID da ação;
2. Remover os traços;
3. Adicionar o prefixo `action`.

Por exemplo, `index` torna-se `actionIndex` e `hello-world` torna-se `actionHelloWorld`.

Observação: Os nomes dos métodos de ações são *case-sensitive*. Se você tiver um método chamado `ActionIndex`, não será considerado como um método de ação e como resultado, o pedido para a ação `index` lançará uma exceção. Observe também que os métodos de ações devem ser públicas. Um método privado ou protegido NÃO será definido como ação inline.

As ações inline normalmente são as mais utilizadas pois demandam pouco esforço para serem criadas. No entanto, se você deseja reutilizar algumas ações em diferentes lugares ou se deseja distribuir uma ação, deve considerá-la como uma *ação standalone*.

Ações Standalone

Ações standalone são definidas por classes de ações que estendem de `yii\base\Action` ou de uma classe-filha. Por exemplo, nas versões do Yii, existe a `yii\web\ViewAction` e a `yii\web>ErrorAction`, ambas são ações standalone.

Para usar uma ação standalone, você deve *mapear as ações* sobrescrevendo o método `yii\base\Controller::actions()` em suas classes controllers como o seguinte:

```
public function actions()
{
    return [
        // declara a ação "error" usando um nome de classe
        'error' => 'yii\web>ErrorAction',

        // declara a ação "view" usando uma configuração em array
        'view' => [
            'class' => 'yii\web\ViewAction',
            'viewPrefix' => '',
        ],
    ];
}
```

Como pode ver, o método `actions()` deve retornar um array cujas chaves são os IDs das ações e os valores correspondentes ao nome da classe da ação ou configurações. Ao contrário das ações inline, os IDs das ações standalone podem conter caracteres arbitrários desde que sejam mapeados no método `actions()`.

Para criar uma classe de ação standalone, você deve estender de `yii\base\Action` ou de duas classes filhas e implementar um método público chamado `run()`. A regra para o método `run()` é semelhante ao de um método de ação. Por exemplo,

```
<?php
namespace app\components;

use yii\base\Action;

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

Resultados da Ação

O valor de retorno do método de ação ou do método `run()` de uma ação standalone são importantes. Eles representam o resultado da ação correspondente.

O valor de retorno pode ser um objeto de `resposta` que será enviado como resposta aos usuários finais.

- Para **aplicações Web**, o valor de retorno também poder ser algum dado arbitrário que será atribuído à propriedade `yii\web\Response::$data` e ainda ser convertido em uma string para representar o corpo da resposta.
- Para **aplicações console**, o valor de retorno também poder ser um inteiro representando o `exit status` (status de saída) da execução do comando.

Nos exemplos acima, todos os resultados são strings que serão tratados como o corpo das respostas para serem enviados aos usuários finais. No exemplo a seguir, mostra como uma ação pode redirecionar o navegador do usuário para uma nova URL retornando um objeto de resposta (o método `redirect()` retorna um objeto de resposta):

```
public function actionForward()
{
    // redireciona o navegador do usuário para https://example.com
    return $this->redirect('https://example.com');
}
```

Parâmetros da Ação

Os métodos de ações para as ações inline e os métodos `run()` para as ações standalone podem receber parâmetros, chamados *parâmetros da ação*. Seus valores são obtidos a partir das requisições. Para **aplicações Web**, o valor de cada parâmetro da ação são obtidos pelo `$_GET` usando o nome do parâmetro como chave; para **aplicações console**, eles correspondem aos argumentos da linha de comando.

No exemplo a seguir, a ação `view` (uma ação inline) possui dois parâmetros declarados: `$id` e `$version`.

```
namespace app\controllers;

use yii\web\Controller;

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

A seguir, os parâmetros da ação serão populados em diferentes requisições:

- `https://hostname/index.php?r=post/view&id=123`: o parâmetro `$id` receberá o valor `'123'`, enquanto o `$version` continuará com o valor nulo porque não existe o parâmetro `version` na URL.
- `https://hostname/index.php?r=post/view&id=123&version=2`: os parâmetros `$id` e `$version` serão receberão os valores `'123'` e `'2'`, respectivamente.
- `https://hostname/index.php?r=post/view`: uma exceção `yii\web\BadRequestHttpException` será lançada porque o parâmetro obrigatório `$id` não foi informado na requisição.
- `https://hostname/index.php?r=post/view&id[]=123`: uma exceção `yii\web\BadRequestHttpException` será lançada porque o parâmetro `$id` foi informado com um valor array `['123']` na qual não era esperado.

Se você quiser que um parâmetro da ação aceite valores arrays, deverá declara-lo explicitamente com `array`, como mostro a seguir:

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

Agora, se a requisição for `https://hostname/index.php?r=post/view&id[]=123`, o parâmetro `$id` receberá o valor `['123']`. Se a requisição for `https://hostname/index.php?r=post/view&id=123`, o parâmetro `$id` ainda receberá um array como valor pois o valor escalar `'123'` será convertido automaticamente em um array.

Os exemplo acima mostram, principalmente, como os parâmetros da ação trabalham em aplicações Web. Para aplicações console, por favor, consulte a seção [Comandos de Console](#) para mais detalhes.

Default Action

Cada controller tem uma ação padrão especificado pela propriedade `yii\base\Controller::$defaultAction`. Quando uma rota contém apenas o ID do controller, implica que a ação padrão do controller seja solicitada.

Por padrão, a ação padrão é definida como `index`. Se quiser alterar o valor padrão, simplesmente sobrescreva esta propriedade na classe controller, como o seguinte:

```
namespace app\controllers;

use yii\web\Controller;

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

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

3.5.5 Ciclo de Vida do Controller

Ao processar uma requisição, a aplicação criará um controller baseada na rota solicitada. O controller, então, se submeterá ao seguinte ciclo de vida para concluir a requisição:

1. O método `yii\base\Controller::init()` é chamado após o controller ser criado e configurado.
2. O controller cria um objeto da ação baseada no ID da ação solicitada:
 - Se o ID da ação não for especificado, o ID da ação padrão será utilizada.
 - Se o ID da ação for encontrada no mapeamento das ações, uma ação standalone será criada;
 - Se o ID da ação for encontrada para corresponder a um método de ação, uma ação inline será criada;
 - Caso contrário, uma exceção `yii\base\InvalidRouteException` será lançada.
3. De forma sequencial, o controller chama o método `beforeAction()` da aplicação, o módulo (se o controller pertencer a um módulo) e o controller.

- Se uma das chamadas retornar `false`, o restante dos métodos subsequentes `beforeAction()` serão ignoradas e a execução da ação será cancelada.
 - Por padrão, cada método `beforeAction()` desencadeia a execução de um evento chamado `beforeAction` na qual você pode associar a uma função (handler).
4. O controller executa a ação:
 - Os parâmetros da ação serão analisados e populados a partir dos dados obtidos pela requisição;
 5. De forma sequencial, o controller chama o método `afterAction()` do controller, o módulo (se o controller pertencer a um módulo) e a aplicação.
 - Por padrão, cada método `afterAction()` desencadeia a execução de um evento chamado `afterAction` na qual você pode associar a uma função (handler).
 6. A aplicação obterá o resultado da ação e irá associá-lo na *resposta*.

3.5.6 Boas Práticas

Em uma aplicação bem projetada, frequentemente os controllers são bem pequenos na qual cada ação possui poucas linhas de códigos. Se o controller for um pouco complicado, geralmente indica que terá que refazê-lo e passar algum código para outro classe.

Segue algumas boas práticas em destaque. Os controllers:

- podem acessar os dados de uma *requisição*;
- podem chamar os métodos dos *models* e outros componentes de serviço com dados da requisição;
- podem usar as *views* para compor as respostas;
- NÃO devem processar os dados da requisição - isto deve ser feito na *camada model (modelo)*;
- devem evitar inserir códigos HTML ou outro código de apresentação - é melhor que sejam feitos nas *views*.

3.6 Models (Modelos)

Os models (modelos) fazem parte da arquitetura MVC⁹. Eles representam os dados, as regras e a lógica de negócio.

Você pode criar uma classe model estendendo de `yii\base\Model` ou de seus filhos. A classe base `yii\base\Model` suporta muitos recursos úteis:

⁹<https://pt.wikipedia.org/wiki/MVC>

- Atributos: representa os dados de negócio e podem ser acessados normalmente como uma propriedade de objeto ou como um elemento de array;
- Labels dos atributos: especifica os labels de exibição dos atributos;
- Atribuição em massa: suporta popular vários atributos em uma única etapa;
- Regras de validação: garante que os dados de entrada sejam baseadas nas regras de validação que foram declaradas;
- Data Exporting: permite que os dados de model a serem exportados em array possuam formatos personalizados.

A classe `Model` também é a classe base para models mais avançados, como o [Active Record](#). Por favor, consulte a documentação relevante para mais detalhes sobre estes models mais avançados.

Informação: Você não é obrigado basear suas classe model em `yii\base\Model`. No entanto, por existir muitos componentes do Yii construídos para suportar o `yii\base\Model`, normalmente é a classe base preferível para um model.

3.6.1 Atributos

Os models representam dados de negócio por meio de *atributos*. Cada atributo é uma propriedade publicamente acessível de um model. O método `yii\base\Model::attributes()` especifica quais atributos de uma classe model possuirá.

Você pode acessar um atributo como fosse uma propriedade normal de um objeto:

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

// "name" é um atributo de ContactForm
$model->name = 'example';
echo $model->name;
```

Você também pode acessar os atributos como elementos de um array, graças ao suporte de `ArrayAccess`¹⁰ e `ArrayIterator`¹¹ pelo `yii\base\Model`:

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

// acessando atributos como elementos de array
$model['name'] = 'example';
echo $model['name'];

// iterando sobre os atributos
foreach ($model as $name => $value) {
    echo "$name: $value\n";
}
```

¹⁰<https://www.php.net/manual/en/class.arrayaccess.php>

¹¹<https://www.php.net/manual/en/class.arrayiterator.php>

Definindo Atributos

Por padrão, se a classe model estender diretamente de `yii\base\Model`, todas as suas variáveis públicas e não estáticas serão atributos. Por exemplo, a classe model `ContactForm` a seguir possui quatro atributos: `name`, `email`, `subject` e `body`. O model `ContactForm` é usado para representar os dados de entrada obtidos a partir de um formulário HTML.

```
namespace app\models;

use yii\base\Model;

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

Você pode sobrescrever o método `yii\base\Model::attributes()` para definir atributos de uma forma diferente. Este método deve retornar os nomes dos atributos em um model. Por exemplo, o `yii\db\ActiveRecord` faz com que o método retorne os nomes das colunas da tabela do banco de dados como nomes de atributos. Observe que também poderá sobrescrever os métodos mágicos tais como `__get()` e `__set()`, para que os atributos possam ser acessados como propriedades normais de objetos.

Labels dos Atributos

Ao exibir valores ou obter dados de entrada dos atributos, muitas vezes é necessário exibir alguns labels associados aos atributos. Por exemplo, dado um atributo chamado `firstName`, você pode querer exibir um label `First Name` que é mais amigável quando exibido aos usuários finais como em formulários e mensagens de erro.

Você pode obter o label de um atributo chamando o método `yii\base\Model::getAttributeLabel()`. Por exemplo,

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

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

Por padrão, os labels dos atributos automaticamente serão gerados com os nomes dos atributos. Isto é feito pelo método `yii\base\Model::generateAttributeLabel()`. Ele transforma os nomes camel-case das variáveis em várias palavras, colocando em caixa alta a primeira letra de cada palavra. Por exemplo, `username` torna-se `Username`, enquanto `firstName` torna-se `First Name`.

Se você não quiser usar esta geração automática do labels, poderá sobrescrever o método `yii\base\Model::attributeLabels()` declarando explicitamente os labels dos atributos. Por exemplo,

```
namespace app\models;

use yii\base\Model;

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

    public function attributeLabels()
    {
        return [
            'name' => 'Your name',
            'email' => 'Your email address',
            'subject' => 'Subject',
            'body' => 'Content',
        ];
    }
}
```

Para aplicações que suportam vários idiomas, você pode querer traduzir os labels dos atributos. Isto também é feito no método `attributeLabels()`, conforme o exemplo a seguir:

```
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'),
    ];
}
```

Você pode até definir condicionalmente os labels dos atributos. Por exemplo, baseado no cenário que o model estiver utilizando, você pode retornar diferentes labels para o mesmo atributo.

Informação: Estritamente falando, os labels dos atributos fazem parte das [views](#) (visões). Mas ao declarar os labels em models (modelos), frequentemente tornam-se mais convenientes e podem resultar um código mais limpo e reutilizável.

3.6.2 Cenários

Um model (modelo) pode ser usado em diferentes *cenários*. Por exemplo, um model `User` pode ser usado para obter dados de entrada de login, mas também pode ser usado com a finalidade de registrar o usuário. Em diferentes cenários, um model pode usar diferentes regras e lógicas de negócio. Por exemplo, um atributo `email` pode ser obrigatório durante o cadastro do usuário, mas não durante ao login.

Um model (modelo) usa a propriedade `yii\base\Model::$scenario` para identificar o cenário que está sendo usado. Por padrão, um model (modelo) suporta apenas um único cenário chamado `default`. O código a seguir mostra duas formas de definir o cenário de um model (modelo):

```
// o cenário é definido pela propriedade
$model = new User;
$model->scenario = 'login';

// o cenário é definido por meio de configuração
$model = new User(['scenario' => 'login']);
```

Por padrão, os cenários suportados por um model (modelo) são determinados pelas regras de validação declaradas no próprio model (modelo). No entanto, você pode personalizar este comportamento sobrescrevendo o método `yii\base\Model::scenarios()`, conforme o exemplo a seguir:

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    public function scenarios()
    {
        return [
            'login' => ['username', 'password'],
            'register' => ['username', 'email', 'password'],
        ];
    }
}
```

Informação: Nos exemplos anteriores, as classes model (modelo) são estendidas de `yii\db\ActiveRecord` por usarem diversos cenários para auxiliarem as classes `Active Record` classes.

O método `scenarios()` retorna um array cujas chaves são os nomes dos cenários e os valores que correspondem aos *active attributes* (atributo ativo). Um atributo ativo podem ser atribuídos em massa e é sujeito a validação. No exemplo anterior, os atributos `username` e `password` são ativos no cenário

`login`; enquanto no cenário `register`, além dos atributos `username` e `password`, o atributo `email` passará a ser ativo.

A implementação padrão do método `scenarios()` retornará todos os cenários encontrados nas regras de validação declaradas no método `yii\base\Model::rules()`. Ao sobrescrever o método `scenarios()`, se quiser introduzir novos cenários, além dos cenários padrão, poderá escrever um código conforme o exemplo a seguir:

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    public function scenarios()
    {
        $scenarios = parent::scenarios();
        $scenarios['login'] = ['username', 'password'];
        $scenarios['register'] = ['username', 'email', 'password'];
        return $scenarios;
    }
}
```

O recurso de cenários são usados principalmente para validação e para atribuição em massa. Você pode, no entanto, usá-lo para outros fins. Por exemplo, você pode declarar diferentes labels para os atributos baseados no cenário atual.

3.6.3 Regras de Validação

Quando os dados para um model (modelo) são recebidos de usuários finais, devem ser validados para garantir que satisfazem as regras (*regras de validação*, também conhecidos como *regras de negócio*). Por exemplo, considerando um model (modelo) `ContactForm`, você pode querer garantir que todos os atributos não sejam vazios e que o atributo `email` contenha um e-mail válido. Se o valor de algum atributo não satisfizer a regra de negócio correspondente, mensagens apropriadas de erros serão exibidas para ajudar o usuário a corrigi-los.

Você pode chamar o método `yii\base\Model::validate()` para validar os dados recebidos. O método usará as regras de validação declaradas em `yii\base\Model::rules()` para validar todos os atributos relevantes. Se nenhum erro for encontrado, o método retornará `true`. Caso contrário, o método irá manter os erros na propriedade `yii\base\Model::$errors` e retornará `false`. Por exemplo,

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

```
// os atributos do model serão populados pelos dados fornecidos pelo
usuário
$model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // todos os dados estão válidos
} else {
    // a validação falhou: $errors é um array contendo as mensagens de erro
    $errors = $model->errors;
}
}
```

Para declarar as regras de validação em um model (modelo), sobrescreva o método `yii\base\Model::rules()` retornando as regras que os atributos do model (modelo) devem satisfazer. O exemplo a seguir mostra as regras de validação sendo declaradas no model (modelo) `ContactForm`:

```
public function rules()
{
    return [
        // os atributos name, email, subject e body são obrigatórios
        [['name', 'email', 'subject', 'body'], 'required'],

        // o atributo email deve ter um e-mail válido
        ['email', 'email'],
    ];
}
```

Uma regra pode ser usada para validar um ou vários atributos e, um atributo pode ser validado por uma ou várias regras. Por favor, consulte a seção [Validação de Dados](#) para mais detalhes sobre como declarar regras de validação.

Às vezes, você pode querer que uma regra se aplique apenas em determinados cenários. Para fazer isso, você pode especificar a propriedade `on` de uma regra, como o seguinte:

```
public function rules()
{
    return [
        // os atributos username, email e password são obrigatórios no
        cenário "register"
        [['username', 'email', 'password'], 'required', 'on' => 'register'],

        // os atributos username e password são obrigatórios no cenário
        "login"
        [['username', 'password'], 'required', 'on' => 'login'],
    ];
}
```

Se você não especificar a propriedade `on`, a regra será aplicada em todos os cenários. Uma regra é chamada de *active rule* (regra ativa), se ela puder ser aplicada no cenário atual.

Um atributo será validado, se e somente se, for um atributo ativo declarado no método `scenarios()` e estiver associado a uma ou várias regras declaradas no método `rules()`.

3.6.4 Atribuição em Massa

Atribuição em massa é a forma conveniente para popular um model (modelo) com os dados de entrada do usuário usando uma única linha de código. Ele popula os atributos de um model (modelo) atribuindo os dados de entrada diretamente na propriedade `yii\base\Model::$attributes`. Os dois códigos a seguir são equivalentes, ambos tentam atribuir os dados do formulário enviados pelos usuários finais para os atributos do model (modelo) `ContactForm`. Evidentemente, a primeira forma, que utiliza a atribuição em massa, é a mais limpa e o menos propenso a erros do que a segunda forma:

```
$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;
```

Atributos Seguros

A atribuição em massa só se aplica aos chamados *safe attributes* (atributos seguros), que são os atributos listados no `yii\base\Model::$scenarios()` para o cenário atual de um model (modelo). Por exemplo, se o model (modelo) `User` declarar o cenário como o código a seguir, quando o cenário atual for `login`, apenas os atributos `username` e `password` podem ser atribuídos em massa. Todos os outros atributos permanecerão inalterados.

```
public function scenarios()
{
    return [
        'login' => ['username', 'password'],
        'register' => ['username', 'email', 'password'],
    ];
}
```

Informação: A razão da atribuição em massa só se aplicar para os atributos seguros é para que você tenha o controle de quais atributos podem ser modificados pelos dados dos usuário finais. Por exemplo, se o model (modelo) tiver um atributo `permission` que determina a permissão atribuída ao usuário, você gostará que apenas os administradores possam modificar este atributo através de uma interface backend.

Como a implementação do método `yii\base\Model::scenarios()` retornará todos os cenários e atributos encontrados em `yii\base\Model::rules()`, se não quiser sobrescrever este método, isto significa que um atributo é seguro desde que esteja mencionado em uma regra de validação ativa.

Por esta razão, uma alias especial de validação chamada `safe`, será fornecida para que você possa declarar um atributo seguro, sem ser validado. Por exemplo, a declaração da regra a seguir faz com que tanto o atributo `title` quanto o `description` sejam seguros.

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

Atributos não Seguros

Como descrito anteriormente, o método `yii\base\Model::scenarios()` serve para dois propósitos: determinar quais atributos devem ser validados e quais atributos são seguros. Em alguns casos raros, você pode querer validar um atributo sem marca-lo como seguro. Para fazer isto, acrescente um ponto de exclamação `!` como prefixo do nome do atributo ao declarar no método `scenarios()`, como o que foi feito no atributo `secret` no exemplo a seguir:

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

Quando o model (modelo) estiver no cenário `login`, todos os três atributos serão validados. No entanto, apenas os atributos `username` e `password` poderão ser atribuídos em massa. Para atribuir um valor de entrada no atributo `secret`, terá que fazer isto explicitamente da seguinte forma:

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

3.6.5 Exportação de Dados

Muitas vezes os models (modelos) precisam ser exportados em diferentes tipos de formatos. Por exemplo, você pode querer converter um conjunto de models (modelos) no formato JSON ou Excel. O processo de exportação pode ser dividido em duas etapas independentes. Na primeira etapa, os models (modelos) serão convertidos em arrays; na segunda etapa, os arrays serão convertidos em um determinado formato. Se concentre apenas na primeira

etapa, uma vez que a segunda etapa pode ser alcançada por formataadores de dados genéricos, tais como o `yii\web\JsonResponseFormatter`.

A maneira mais simples de converter um model (modelo) em um array consiste no uso da propriedade `yii\base\Model::$attributes`. Por exemplo,

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

Por padrão, a propriedade `yii\base\Model::$attributes` retornará os valores de todos os atributos declarados no método `yii\base\Model::attributes()`.

Uma maneira mais flexível e poderosa de converter um model (modelo) em um array é através do método `yii\base\Model::toArray()`. O seu comportamento padrão é o mesmo do `yii\base\Model::$attributes`. No entanto, ele permite que você escolha quais itens de dados, chamados de *fields* (campos), devem ser mostrados no array resultante e como eles devem vir formatados. Na verdade, é a maneira padrão de exportação de models (modelos) no desenvolvimento de Web services RESTful, como descrito na seção [Formatando Respostas](#).

Campos

Um campo é simplesmente um elemento nomeado no array obtido pela chamada do método `yii\base\Model::toArray()` de um model (modelo).

Por padrão, os nomes dos campos são iguais aos nomes dos atributos. No entanto, você pode alterar este comportamento sobrescrevendo os métodos `fields()` e/ou `extraFields()`. Ambos os métodos devem retornar uma lista dos campos definidos. Os campos definidos pelo método `fields()` são os campos padrão, o que significa que o `toArray()` retornará estes campos por padrão. O método `extraFields()` define, de forma adicional, os campos disponíveis que também podem ser retornados pelo `toArray()`, contanto que sejam especificados através do parâmetro `$expand`. Por exemplo, o código a seguir retornará todos os campos definidos em `fields()` incluindo os campos `prettyName` e `fullAddress`, a menos que estejam definidos no `extraFields()`.

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

Você poderá sobrescrever o método `fields()` para adicionar, remover, renomear ou redefinir os campos. O valor de retorno do `fields()` deve ser um array. As chaves do array não os nomes dos campos e os valores correspondem ao nome do atributo definido, na qual, podem ser tanto os nomes de propriedades/atributos quanto funções anônimas que retornam o valor dos campos correspondentes. Em um caso especial, quando o nome do campo for igual ao nome do atributo definido, você poderá omitir a chave do array. Por exemplo,

```

// usar uma lista explicita de todos os campos lhe garante que qualquer
mudança
// em sua tabela do banco de dados ou atributos do model (modelo) não altere
os
// nomes de seus campos (para manter compatibilidade com versões anterior da
API).
public function fields()
{
    return [
        // o nome do campos é igual ao nome do atributo
        'id',

        // o nome do campo é "email", o nome do atributo correspondente é
        "email_address"
        'email' => 'email_address',

        // o nome do campo é "name", o seu valor é definido por uma função
        call-back do PHP
        'name' => function () {
            return $this->first_name . ' ' . $this->last_name;
        },
    ];
}

// filtra alguns campos, é bem usado quando você quiser herdar a
implementação
// da classe pai e remover alguns campos delicados.
public function fields()
{
    $fields = parent::fields();

    // remove os campos que contém informações delicadas
    unset($fields['auth_key'], $fields['password_hash'],
    $fields['password_reset_token']);

    return $fields;
}

```

Atenção: Como, por padrão, todos os atributos de um model (modelo) serão >incluídos no array exportado, você deve examinar seus dados para ter certeza >que não possuem informações delicadas. Se existir, deverá sobrescrever o método >fields() para remove-los. No exemplo anterior, nós decidimos remover os >campos auth_key, password_hash e password_reset_token.

3.6.6 Boas Práticas

A representação dos dados, regras e lógicas de negócios estão centralizados nos models (modelos). Muitas vezes precisam ser reutilizadas em lugares diferentes. Em um aplicativo bem projetado, models (modelos) geralmente são muitos maiores que os `controllers`

Em resumo, os models (modelos):

- podem conter atributos para representar os dados de negócio;
- podem conter regras de validação para garantir a validade e integridade dos dados;
- podem conter métodos para implementar lógicas de negócio;
- NÃO devem acessar diretamente as requisições, sessões ou quaisquer dados do ambiente do usuário. Os models (modelos) devem receber estes dados a partir dos **controllers** (controladores);
- devem evitar inserir HTML ou outros códigos de apresentação – isto deve ser feito nas **views** (visões);
- devem evitar ter muitos cenários em um único model (modelo).

Você deve considerar em utilizar com mais frequência a última recomendação acima quando desenvolver sistemas grandes e complexos. Nestes sistemas, os models (modelos) podem ser bem grandes, pois são usados em muitos lugares e podendo, assim, conter muitas regras e lógicas de negócio. Nestes casos, a manutenção do código de um model (modelo) pode se transformar em um pesadelo, na qual uma simples mudança no código pode afetar vários lugares diferentes. Para desenvolver um model (modelo) manutenível, você pode seguir a seguinte estratégia:

- Definir um conjunto de classes model (modelo) base que são compartilhados por diferentes **aplicações** ou **módulos**. Estas classes model (modelo) base deve conter um conjunto mínimo de regras e lógicas de negócio que são comuns entre os locais que as utilizem.
- Em cada **aplicação** ou **módulo** que usa um model (modelo), deve definir uma classe model (modelo) concreta que estenderá a classe model (modelo) base que a corresponde. A classe model (modelo) concreta irá conter apenas as regras e lógicas que são específicas de uma aplicação ou módulo.

Por exemplo, no Template Avançado de Projetos, você pode definir uma classe model (modelo) base `common\models\Post`. Em seguida, para a aplicação front-end, você define uma classe model (modelo) concreta `frontend\models\Post` que estende de `common\models\Post`. E de forma similar para a aplicação back-end, você define a `backend\models\Post`. Com essa estratégia, você garantirá que o `frontend\models\Post` terá apenas códigos específicos da aplicação front-end e, se você fizer qualquer mudança nele, não precisará se preocupar se esta mudança causará erros na aplicação back-end.

3.7 Visões (Views)

As views fazem parte da arquitetura MVC¹². São a parte do código responsável por apresentar dados aos usuários finais. Em um aplicação Web, views geralmente são criadas em termos de *view templates* (modelos de view)

¹²<https://pt.wikipedia.org/wiki/MVC>

que são arquivos PHP contendo principalmente códigos HTML e códigos PHP de apresentação. Os modelos de view são gerenciados pelo [componente da aplicação view](#) que fornece métodos comumente utilizados para facilitar a montagem e a renderização da view em si. Para simplificar, geralmente chamamos os modelos de view ou seus arquivos simplesmente de view.

3.7.1 Criando Views

Conforme já mencionado, uma view é simplesmente um arquivo PHP composto por HTML ou códigos PHP. O código a seguir representa uma view que exibe um formulário de login. Como você pode ver, o código PHP é utilizado para gerar as partes de conteúdo dinâmicas, tais como o título da página e o formulário, enquanto o código HTML dispõe os itens na página de uma forma apresentável.

```
<?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>Por favor, preencha os seguintes campos para entrar:</p>

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

Em uma view, você pode acessar a variável `$this` que referencia o [componente view](#) responsável por gerenciar e renderizar a view em questão.

Além de `$this`, pode haver outras variáveis predefinidas na view, tal como `$model` no exemplo acima. Essas variáveis representam os dados que foram enviados à view por meio dos [controllers](#) ou de outros objetos que desencadeiam a renderização da view.

Dica: As variáveis predefinidas são listadas em um bloco de comentário no início de uma view para que possam ser reconhecidas pelas IDEs. Além de ser também uma ótima maneira de documentar suas views.

Segurança

Ao criar views que geram páginas HTML, é importante que você codifique e/ou filtre os dados que vêm de usuários antes de exibí-los. Caso contrário, sua aplicação poderá estar sujeita a um ataque de cross-site scripting¹³.

Para exibir um texto simples, codifique-o antes por chamar o método `yii\helpers\Html::encode()`. Por exemplo, o código a seguir codifica o nome do usuário antes de exibí-lo:

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

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

Para exibir conteúdo HTML, use `yii\helpers\HtmlPurifier` para filtrar o conteúdo primeiro. Por exemplo, o código a seguir filtra o conteúdo de `$post->text` antes de exibí-lo:

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

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

Dica: Embora o `HTMLPurifier` faça um excelente trabalho em tornar a saída de dados segura, ele não é rápido. Você deveria considerar guardar em [cache](#) o resultado filtrado se sua aplicação precisa de alta performance.

Organizando as Views

Assim como para os [controllers](#) e para os [models](#), existem convenções para organizar as views.

- Views renderizadas por um controller deveriam ser colocadas sob o diretório `@app/views/IDdoController` por padrão, onde `IDdoController` refere-se ao [ID do controller](#). Por exemplo, se a classe do controller for `PostController`, o diretório será `@app/views/post`; se for `PostCommentController`, o diretório será `@app/views/post-comment`. Caso o controller pertença a um módulo, o diretório seria `views/IDdoController` sob o diretório do módulo.

¹³https://pt.wikipedia.org/wiki/Cross-site_scripting

- Views renderizadas em um [widget](#) deveriam ser colocadas sob o diretório `WidgetPath/views` por padrão, onde `WidgetPath` é o diretório o arquivo da classe do widget.
- Para views renderizadas por outros objetos, é recomendado que você siga a convenção semelhante à dos widgets.

Você pode personalizar os diretórios padrões das views sobrescrevendo o método `yii\base\ViewContextInterface::getViewPath()` dos controllers ou dos widgets.

3.7.2 Renderizando Views

Você pode renderizar views em [controllers](#), em [widgets](#) ou em qualquer outro lugar chamando os métodos de renderização da view. Esses métodos compartilham uma assinatura similar, como a seguir:

```
/**
 * @param string $view nome da view ou caminho do arquivo, dependendo do
 * método de renderização
 * @param array $params os dados passados para a view
 * @return string resultado da renderização
 */
methodName($view, $params = [])
```

Renderização em Controllers

Nos [controllers](#), você pode chamar os seguintes métodos para renderizar as views:

- `render()`: renderiza uma view nomeada e aplica um layout ao resultado da renderização.
- `renderPartial()`: renderiza uma view nomeada sem qualquer layout.
- `renderAjax()`: renderiza uma view nomeada sem qualquer layout e injeta todos os arquivos JS/CSS registrados. É geralmente utilizado em respostas de requisições Web Ajax.
- `renderFile()`: renderiza uma view a partir de um caminho de arquivo ou a partir de um [alias](#).
- `renderContent()`: renderiza um conteúdo estático que será incorporado no layout selecionado. Este método está disponível desde a versão 2.0.1.

Por exemplo,

```
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 NotFoundException;
        }

        // renderiza uma view chamada "exibir" e aplica um layout a ela
        return $this->render('exibir', [
            'model' => $model,
        ]);
    }
}

```

Renderização em Widgets

Nos `widgets`, você pode chamar os seguintes métodos do widget para renderizar views:

- `render()`: renderiza uma view nomeada.
- `renderFile()`: renderiza uma view a partir de um caminho de arquivo ou a partir de um `alias`.

Por exemplo,

```

namespace app\components;

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

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

    public function run()
    {
        // renderiza uma view chamada "listar"
        return $this->render('listar', [
            'items' => $this->items,
        ]);
    }
}

```

Renderização em Views

Você pode renderizar uma view dentro de outra chamando um dos seguintes métodos fornecidos pelo componente `view`:

- `render()`: renderiza uma view nomeada.
- `renderAjax()`: renderiza uma view nomeada sem qualquer layout e injeta todos os arquivos JS/CSS registrados. É geralmente utilizado em respostas de requisições Web Ajax.

- `renderFile()`: renderiza uma view a partir de um caminho de arquivo ou a partir de um *alias*.

Por exemplo, no código a seguir, uma view qualquer renderiza outro arquivo de view chamado `_visao-geral.php` que encontram-se em seu mesmo diretório. Lembre-se que `$this` na view referencia o componente `view`:

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

Renderização em Outros Lugares

Em qualquer lugar, você pode acessar o componente de aplicação `view` pela expressão `Yii::$app->view` e então chamar qualquer método mencionado anteriormente para renderizar uma view. Por exemplo,

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

Views Nomeadas

Ao renderizar uma view, você pode especificá-la usando seu nome, ou o caminho do arquivo, ou um *alias*. Na maioria dos casos, você usará a primeira maneira por ser mais concisa e flexível. Quando especificamos views por nome, chamamos essas views de *views nomeadas*.

Um nome de view é convertido no caminho de arquivo da view correspondente de acordo com as seguintes regras:

- Um nome de view pode omitir a extensão do arquivo. Neste caso, o `.php` será usado como extensão. Por exemplo, a view chamada *sobre* corresponderá ao arquivo `sobre.php`.
- Se o nome da view iniciar com barras duplas `//`, o caminho correspondente seria `@app/views/ViewName`. Ou seja, a view será localizada sob o diretório das views da aplicação. Por exemplo, `//site/sobre` corresponderá ao `@app/views/site/sobre.php`.
- Se o nome da view iniciar com uma barra simples `/`, o caminho do arquivo da view será formado pelo nome da view com o diretório da view do módulo ativo. Se não houver um módulo ativo, o `@app/views/ViewName` será usado. Por exemplo, `/usuario/criar` corresponderá a `@app/modules/user/views/usuario/criar.php` caso o módulo ativo seja `user`. Se não existir um módulo ativo, o caminho do arquivo da view será `@app/views/usuario/criar.php`.
- Se a view for renderizada com um contexto e que implemente `yii\base\ViewContextInterface`, o caminho do arquivo da view será formado por prefixar o diretório da view do contexto ao nome da view. Isto se aplica principalmente às views renderizadas em controllers e widgets. Por exemplo, *sobre* corresponderá a `@app/views/site/sobre.php` caso o contexto seja o controller `SiteController`.

- Se uma view for renderizada dentro de outra, o diretório que contém esta outra view será usado para formar o caminho de seu arquivo. Por exemplo, `item` corresponderá a `@app/views/post/item.php` se ela for renderizada dentro da view `@app/views/post/index.php`.

De acordo com as regras acima, chamar `$this->render('exibir')` em um controller `app\controllers\PostController` vai realmente renderizar o arquivo de view `@app/views/post/exibir.php` e, chamar `$this->render('_visaogeral')` nessa view (`exibir.php`) vai renderizar o arquivo de visão `@app/views/post/_visaogeral.php`.

Acessando Dados em Views

Existem duas abordagens para acessar dados em uma view : *push* e *pull*.

Ao passar os dados como o segundo parâmetro nos métodos de renderização de view, você estará usando a abordagem *push*. Os dados devem ser representados por um array com pares de nome-valor. Quando a view estiver sendo renderizada, a função `extract()` do PHP será executada sobre essa array a fim de extrair seus dados em variáveis na view. Por exemplo, o renderização da view a seguir, em um controller, disponibilizará (pela abordagem *push*) duas variáveis para a view `relatorio`: `$foo = 1` e `$bar = 2`.

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

A abordagem *pull* ativamente obtém os dados do componente `view` ou de outros objetos acessíveis nas views (por exemplo, `Yii::$app`). Usando o código a seguir como exemplo, dentro da view você pode acessar seu objeto controller usando a expressão `$this->context`. E como resultado, será possível acessar quaisquer propriedades ou métodos do controller, como o seu ID, na view `relatorio`:

```
O ID do controller é: <?= $this->context->id ?>  
?>
```

A abordagem *push* normalmente é a forma preferida de acessar dados nas views por que as torna menos dependentes de objetos de contexto. A desvantagem é que você precisa montar manualmente os dados em um array o tempo todo, o que poderia se tornar tedioso e propenso a erros se uma view for compartilhada e renderizada em lugares diferentes.

Compartilhando Dados entre as Views

O componente `view` fornece a propriedade `params` que você pode usar para compartilhar dados entre as views.

Por exemplo, em uma view `sobre`, você pode ter o seguinte código que especifica o seguimento atual do “rastros de navegação” (breadcrumbs):

```
$this->params['breadcrumbs'][] = 'Sobre nós';
```

Em seguida, no arquivo layout, que também é uma view, você pode exibir o “rastros de navegação” (breadcrumbs) usando os dados passados pela propriedade `params`:

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

3.7.3 Layouts

Layouts são um tipo especial de view que representam as partes comuns de múltiplas views. Por exemplo, as páginas da maioria das aplicações Web compartilham o mesmo cabeçalho e rodapé. Embora você possa repetir o mesmo cabeçalho e rodapé em todas as view, a melhor maneira é fazer isso apenas uma vez no layout e incorporar o resultado da renderização de uma view em um lugar apropriado no layout.

Criando Layouts

Visto que os layouts também são views, eles podem ser criados de forma semelhante às views normais. Por padrão, layouts são salvos no diretório `@app/views/layouts`. Layouts usados em um módulo devem ser salvos no diretório `views/layouts` sob o diretório do módulo. Você pode personalizar o diretório padrão de layouts configurando a propriedade `yii\base\Module::$layoutPath` da aplicação ou do módulo.

O exemplo a seguir mostra como é um layout. Observe que, para fins ilustrativos, simplificamos bastante o código do layout. Na prática, você pode querer adicionar mais conteúdos a ele, tais como tags no head, 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>Minha Empresa</header>
<?= $content ?>
<footer>&copy; 2014 por Minhas Empresa</footer>
<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

Conforme pode ver, o layout gera as tags HTML que são comuns a todas as páginas. Na seção `<body>`, o layout vai inserir a variável `$content` que representa o resultado da renderização do conteúdo das views e é enviado ao layout quando método `yii\base\Controller::render()` for chamado.

A maioria dos layouts devem chamar os métodos listados a seguir, conforme ocorreu no código acima. Estes métodos essencialmente desencadeiam eventos referentes ao processo de renderização para que scripts e tags registrados em outros lugares possam ser inseridos nos locais onde eles (os métodos) forem chamados.

- **beginPage()**: Este método deve ser chamado no início do layout. Ele dispara o evento `EVENT_BEGIN_PAGE` que indica o início de uma página.
- **endPage()**: Este método deve ser chamado no final do layout. Ele dispara o evento `EVENT_END_PAGE` que indica o fim de uma página.
- **head()**: Este método deve ser chamado na seção `<head>` de uma página HTML. Ele gera um marcador que será substituído por código HTML (por exemplo, tags `<link>` e `<meta>`) quando a página termina a renderização.
- **beginBody()**: Este método deve ser chamado no início da seção `<body>`. Ele dispara o evento `EVENT_BEGIN_BODY` e gera um marcador que será substituído por código HTML que estiver registrado para essa posição (por exemplo, algum código JavaScript).
- **endBody()**: Este método deve ser chamado no final da seção `<body>`. Ele dispara o evento `EVENT_END_BODY` e gera um marcador que será substituído por código HTML que estiver registrado para essa posição (por exemplo, algum código JavaScript).

Acessando Dados nos Layouts

Dentro de um layout, você tem acesso a duas variáveis predefinidas: `$this` e `$content`. A primeira se refere ao componente **view** como em views normais, enquanto a segunda contém o resultado da renderização do conteúdo de uma view que é gerada por chamar o método **render()** no controller.

Se você quiser acessar outros dados nos layouts, você terá de usar a abordagem *pull* conforme descrito na subseção **Acessando Dados em Views**. Se você quiser passar os dados do conteúdo da view para um layout, poderá usar o método descrito na subseção **Compartilhando Dados entre as Views**.

Usando Layouts

Conforme descrito na subseção Renderização em Controllers, quando você renderiza uma view chamando o método `render()` em um controller, será aplicado um layout ao resultado da renderização. Por padrão, o layout `@app/views/layouts/main.php` será usado.

Você pode usar um layout diferente configurando ou a propriedade `yii\base\Application::$layout` ou a `yii\base\Controller::$layout`. A primeira especifica o layout usado por todos os controllers, enquanto a segunda é usada para controllers de forma individual, sobrescrevendo a primeira. Por exemplo, o código a seguir faz com que o controller `post` usar o `@app/views/layouts/post.php` como layout quando renderizar as suas views. Outros controllers, assumindo que a propriedade `layout` da aplicação não tenha sido alterada, usarão o layout padrão `@app/views/layouts/main.php`.

```
namespace app\controllers;

use yii\web\Controller;

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

    // ...
}
```

Para os controllers que pertencem a um módulo, você também pode configurar a propriedade `layout` do módulo para usar um layout em particular para esses controllers.

Visto que a propriedade `layout` pode ser configurada em diferentes níveis (controllers, módulos, aplicação), por trás das cortinas o Yii determina, em duas etapas, qual arquivo de layout será usado por um controller em particular.

Na primeira etapa, o Yii determina o valor da propriedade do layout e o módulo de contexto:

- Se a propriedade `yii\base\Controller::$layout` do controller não for `null`, ela será usada e o módulo do controller será usado como módulo de contexto.
- Se a propriedade `layout` for `null`, o Yii pesquisará através de todos os módulos ancestrais do controller (incluindo a própria aplicação) até encontrar o primeiro módulo cuja propriedade `layout` não for `null`. O módulo encontrado será usado como módulo de contexto e o valor de sua propriedade `layout` como o layout escolhido. Se nenhum módulo for encontrado, nenhum layout será aplicado.

Na segunda etapa, o Yii determina o real arquivo de layout de acordo com o valor da propriedade `layout` e com o módulo de contexto obtidos na primeira etapa. O valor da propriedade `layout` pode ser:

- uma alias de caminho (por exemplo, `@app/views/layouts/main`).
- um caminho absoluto (por exemplo, `/main`): o valor começa com uma barra. O arquivo de layout será procurado sob o **diretório de layouts** da aplicação, cujo valor padrão é `@app/views/layouts`.
- um caminho relativo (por exemplo, `main`): o arquivo de layout será procurado sob o **diretório de layouts** do módulo de contexto, cujo valor padrão é `views/layouts` sob o **diretório do módulo**.
- um valor booleano `false`: nenhum layout será aplicado.

Se o valor da propriedade `layout` não tiver uma extensão de arquivo, será usada a extensão `.php` por padrão.

Layouts Aninhados

Às vezes, você pode querer que um layout seja usado dentro de outro. Por exemplo, em diferentes seções de um site, você pode querer usar diferentes layouts, e todos esses layouts compartilharão o mesmo layout básico a fim de produzir toda a estrutura da página HTML. Você pode fazer isso por chamar os métodos `beginContent()` e `endContent()` nos layouts filhos, como no exemplo a seguir:

```
<?php $this->beginContent('@app/views/layouts/base.php'); ?>

...conteúdo do layout filho aqui...

<?php $this->endContent(); ?>
```

Como mostrado acima, o conteúdo do layout filho deve ser inserido entre os métodos `beginContent()` e `endContent()`. O parâmetro passado para o `beginContent()` indica qual é o layout pai. Ele pode ser um arquivo de layout ou mesmo um alias.

Usando a abordagem acima, você pode aninhar os layouts em mais de um nível.

Usando Blocos

Blocos permitem que você especifique o conteúdo da view em um local e o exiba em outro. Geralmente são usados em conjunto com os layouts. Por exemplo, você pode definir um bloco no conteúdo de uma view e exibi-lo no layout.

Para definir um bloco, chame os métodos `beginBlock()` e `endBlock()`. O bloco pode então ser acessado via `$view->blocks[$blockID]`, onde o `$blockID` é o identificador único que você associou ao bloco quando o definiu.

O exemplo a seguir mostra como você pode usar blocos para personalizar as partes específicas de um layout pelo conteúdo da view.

Primeiramente, no conteúdo da view, defina um ou vários blocos:

```

...

<?php $this->beginBlock('bloco1'); ?>

...conteúdo do bloco1...

<?php $this->endBlock(); ?>

...

<?php $this->beginBlock('bloco3'); ?>

... conteúdo do bloco3...

<?php $this->endBlock(); ?>

```

Em seguida, no layout, renderize os blocos se estiverem disponíveis ou exiba um conteúdo padrão se não estiverem.

```

...

<?php if (isset($this->blocks['bloco1'])): ?>
    <?= $this->blocks['bloco1'] ?>
<?php else: ?>
    ... conteúdo padrão para o bloco1 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['bloco2'])): ?>
    <?= $this->blocks['bloco2'] ?>
<?php else: ?>
    ... conteúdo padrão para o bloco2 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['bloco3'])): ?>
    <?= $this->blocks['bloco3'] ?>
<?php else: ?>
    ... conteúdo padrão para o bloco3 ...
<?php endif; ?>

...

```

3.7.4 Usando Componentes View

Os **componentes view** fornecem muitos recursos. Embora você possa obtê-los por criar instancias individuais de `yii\base\View` ou de suas classes filhas, na maioria dos casos você usará o componente `view` da aplicação. Você pode configurar este componente nas [configurações da aplicação](#) conforme o exemplo a seguir:

```

[
    // ...

```

```

        'components' => [
            'view' => [
                'class' => 'app\components\View',
            ],
            // ...
        ],
    ],
]

```

Componentes de view fornecem úteis recursos relacionados. Cada um deles está descrito com mais detalhes em seções separadas:

- **temas**: permite que você desenvolva e altere temas para o seu site.
- **fragmento de cache**: permite que você guarde em cache um fragmento de uma página.
- **manipulação de client scripts**: permite que você registre e renderize CSS e JavaScript.
- **manipulando asset bundle**: permite que você registre e renderize recursos estáticos (asset bundles).
- **template engines alternativos**: permite que você use outros template engines, tais como o Twig¹⁴ e Smarty¹⁵.

Você também pode usar os seguintes recursos que, embora simples, são úteis quando estiver desenvolvendo suas páginas.

Configurando Títulos de Página

Cada página deve ter um título. Normalmente, a tag <title> é exibida em um layout. Mas, na prática, o título é muitas vezes determinado no conteúdo das views, em vez de nos layouts. Para resolver este problema, a classe `yii\web\View` fornece a propriedade `title` para você passar o título a partir das views para o layout.

Para fazer uso deste recurso, em cada view, você pode definir o título da página conforme o exemplo a seguir:

```

<?php
$this->title = 'Título da Minha Página';
?>

```

E, no layout, certifique-se de ter o seguinte código dentro da seção <head>:

```

<title><?= Html::encode($this->title) ?></title>

```

Registrando os Meta Tags

Páginas Web geralmente precisam gerar variadas meta tags necessárias a diversas finalidades. Assim como os títulos, as meta tags precisam estar na seção <head> e normalmente são geradas nos layouts.

¹⁴<https://twig.symfony.com/>

¹⁵<https://www.smarty.net/>

Se você quiser especificar quais meta tags gerar no conteúdo das views, poderá chamar o método `yii\web\View::registerMetaTag()` na view, conforme o exemplo a seguir:

```
<?php
$this->registerMetaTag(['name' => 'keywords', 'content' => 'yii, framework,
php']);
?>
```

O código acima registrará uma meta tag “keywords” com o componente view. A meta tag registrada será renderizada depois de o layout finalizar sua renderização. O código HTML a seguir será gerado e inserido no local onde você chama `yii\web\View::head()` no layout:

```
<meta name="keywords" content="yii, framework, php">
```

Observe que se você chamar o método `yii\web\View::registerMetaTag()` muitas vezes, ele registrará diversas meta tags, independente se forem as mesmas ou não.

Para garantir que exista apenas uma única instância de um tipo de meta tag, você pode especificar uma chave no segundo parâmetro ao chamar o método. Por exemplo, o código a seguir registra dois meta tags “description”. No entanto, apenas o segundo será renderizado.

```
$this->registerMetaTag(['name' => 'description', 'content' => 'Este é o meu
website feito com Yii!'], 'descricao');
$this->registerMetaTag(['name' => 'description', 'content' => 'Este website
é sobre coisas divertidas.'], 'descricao');
```

Registrando Tags Link

Assim como as meta tags, as tags link são úteis em muitos casos, tais como a personalização do favicon, apontamento para feed RSS ou delegação do OpenID para outros servidores. Você pode trabalhar com as tags link de forma similar às meta tags, usando o método `yii\web\View::registerLinkTag()`. Por exemplo, na view, você pode registrar uma tag link como segue:

```
$this->registerLinkTag([
    'title' => 'Notícias sobre o Yii',
    'rel' => 'alternate',
    'type' => 'application/rss+xml',
    'href' => 'https://www.yiiframework.com/rss.xml/',
]);
```

O código acima resultará em

```
<link title="Notícias sobre o Yii" rel="alternate"
type="application/rss+xml" href="https://www.yiiframework.com/rss.xml/">
```

Assim como no método `registerMetaTags()`, você também pode especificar uma chave quando chamar o método `registerLinkTag()` para evitar a criação de tags link repetidas.

3.7.5 Eventos da View

Componentes `view` disparam vários eventos durante o processo de renderização da `view`. Você pode usar estes eventos para inserir conteúdo nas `views` ou processar os resultados da renderização antes de serem enviados para os usuários finais.

- `EVENT_BEFORE_RENDER`: disparado no início da renderização de um arquivo em um `controller`. Funções registradas para esse evento (handlers) podem definir a propriedade `yii\base\ViewEvent::$isValid` como `false` para cancelar o processo de renderização.
- `EVENT_AFTER_RENDER`: disparado depois da renderização de um arquivo pela chamada de `yii\base\View::afterRender()`. Funções registradas para esse evento (handlers) podem capturar o resultado da renderização por meio da propriedade `yii\base\ViewEvent::$output` e podem modificá-lo para alterar o resultado final.
- `EVENT_BEGIN_PAGE`: disparado pela chamada do método `yii\base\View::beginPage()` nos layouts.
- `EVENT_END_PAGE`: disparado pela chamada do método `yii\base\View::endPage()` nos layouts.
- `EVENT_BEGIN_BODY`: disparado pela chamada do método `yii\web\View::beginBody()` nos layouts.
- `EVENT_END_BODY`: disparado pela chamada do método `yii\web\View::endBody()` nos layouts.

Por exemplo, o código a seguir insere a data atual no final do corpo da página:

```
\Yii::$app->view->on(View::EVENT_END_BODY, function () {
    echo date('Y-m-d');
});
```

3.7.6 Renderizando Páginas Estáticas

Páginas estáticas referem-se a páginas cujo principal conteúdo é, na maior parte, estático, sem a necessidade de acessar dados dinâmicos provenientes dos `controllers`.

Você pode retornar páginas estáticas por colocar seu código na `view` e então, em um `controller`, usar o código a seguir:

```
public function actionAbout()
{
    return $this->render('about');
}
```

Se o site contiver muitas páginas estáticas, seria tedioso repetir os códigos similares muitas vezes. Para resolver este problema, você pode inserir uma `action` “externa” (`standalone action`) chamando a classe `yii\web\ViewAction` em um `controller`. Por exemplo:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'page' => [
                'class' => 'yii\web\ViewAction',
            ],
        ];
    }
}
```

Agora, se você criar uma view chamada `sobre` no diretório `@app/views/site/pages`, poderá exibir por meio da seguinte URL:

`http://localhost/index.php?r=site/page&view=sobre`

O parâmetro `view` passado via GET informa à classe `yii\web\ViewAction` qual view foi solicitada. A action, então, irá procurar essa view informada dentro do diretório `@app/views/site/pages`. Você pode configurar a propriedade `yii\web\ViewAction::$viewPrefix` para alterar o diretório onde as views serão procuradas.

3.7.7 Boas Práticas

Views são responsáveis por apresentar models (modelos) no formato que os usuários finais desejam. Em geral, views:

- devem conter principalmente código de apresentação, tal como o HTML, e trechos simples de PHP para percorrer, formatar e renderizar dados.
- não devem conter código de consulta ao banco de dados. Consultas assim devem ser feitas nos models.
- devem evitar acessar diretamente os dados da requisição, tais como `$_GET` e `$_POST` pois essa tarefa cabe aos controllers. Se os dados da requisição forem necessários, deverão ser fornecidos às views pelos controllers.
- podem ler as propriedades dos models, mas não devem alterá-las.

Para tornar as views mais gerenciáveis, evite criar views muito complexas ou que contenham muito código redundante. Você pode usar as seguintes técnicas para atingir este objetivo:

- use layouts para representar as seções de apresentação comuns (por exemplo, cabeçalho e rodapé).
- divida uma view complicada em várias outras menores. As views menores podem ser renderizadas e montadas em uma maior usando os métodos descritos anteriormente.

- crie e use [widgets](#) como blocos de construção das views.
- crie e use as classes helper (auxiliares) para transformar e formatar os dados nas views.

3.8 Módulos

Os módulos são unidades independentes de software que são compostos de [models](#) (modelos), [views](#) (visões), [controllers](#) (controladores) e outros componentes de apoio. Os usuários finais podem acessar os controllers (controladores) de um módulo caso esteja implementado na [aplicação](#). Por estas razões, os módulos são muitas vezes enxergados como mini-aplicações. Os módulos diferem das [aplicações](#) pelo fato de não poderem ser implementados sozinhos e que devem residir dentro das aplicações.

3.8.1 Criando Módulos

Um módulo é organizado como um diretório que é chamado de `caminho base` do módulo. Dentro deste diretório, existem subdiretório, como o `controllers`, `models`, `views` que mantêm os controllers (controladores), models (modelos), views (visões) e outros códigos assim como em uma aplicação. O exemplo a seguir mostra o conteúdo dentro de um módulo:

<code>forum/</code>	
<code>Module.php</code>	o arquivo da classe do módulo
<code>controllers/</code>	contém os arquivos da classe controller
<code>DefaultController.php</code>	o arquivo da classe controller padrão
<code>models/</code>	contém os arquivos da classe model
<code>views/</code>	contém a view do controller e os arquivos
<code>de layout</code>	
<code>layouts/</code>	contém os arquivos de layout
<code>default/</code>	contém os arquivos de view para o
<code>DefaultController</code>	
<code>index.php</code>	o arquivo de view index

Classe do Módulo

Cada módulo deve ter uma única classe que estende de `yii\base\Module`. Esta classe deve estar localizada diretamente sob o `caminho base` do módulo e deve ser [autoloadable](#). Quando um módulo estiver sendo acessado, uma única instância da classe módulo correspondente será criada. Assim como as [instâncias da aplicação](#), as instâncias do módulo são usadas para compartilhar dados e componentes para os códigos dentro dos módulos.

Segue um exemplo de uma classe de módulo:

```
namespace app\modules\forum;

class Module extends \yii\base\Module
```

```

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

        $this->params['foo'] = 'bar';
        // ... outros códigos de inicialização ...
    }
}

```

Se o método `init()` inicializar muitas propriedades do módulo, você poderá salva-los em um arquivo de [configuração](#) e carregá-los como mostro no código a seguir no método `init()`:

```

public function init()
{
    parent::init();
    // inicializa o módulo com as configurações carregadas de config.php
    \Yii::configure($this, require __DIR__ . '/config.php');
}

```

O arquivo de configuração `config.php` pode conter o conteúdo a seguir, que é semelhante ao de uma [configuração de aplicação](#).

```

<?php
return [
    'components' => [
        // lista de configuração de componentes
    ],
    'params' => [
        // lista de parâmetros
    ],
];

```

Controllers em Módulos

Ao criar controllers (controladores) em um módulo, uma convenção é colocar as classes dos controllers (controladores) sob o sub-namespace `controllers` do namespace do módulo da classe. Isto também significa que os arquivos da classe controller (controlador) devem ser colocadas no diretório `controllers` dentro do `caminho base` do módulo. Por exemplo, para criar um controller (controlador) `post` no módulo `forum` mostrado na última subseção, você deve declarar a classe controller (controlador) conforme o seguinte exemplo:

```

namespace app\modules\forum\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    // ...
}

```

Você pode personalizar o namespace da classe do controller (controlador) configurando a propriedade `yii\base\Module::$controllerNamespace`. No caso de alguns controllers (controladores) estiverem fora do namespace, você poderá torná-los acessíveis pela configuração da propriedade `yii\base\Module::$controllerMap`, de forma similar ao que você fez na aplicação.

Views em Módulos

As views (visões) devem ser colocadas no diretório `views` dentro do `caminho base` do módulo. Para as views (visões) renderizadas por um controller (controlador) no módulo, devem ser colocadas sob o diretório `views/IDdoController`, onde o `IDdoController` refere-se ao [ID do controller \(controlador\)](#). Por exemplo, se a classe do controller (controlador) for `PostController`, o diretório será `views/post` dentro do `caminho base` do módulo.

Um módulo pode especificar um [layout](#) que será aplicado pelas views (visões) renderizadas pelos controllers (controladores) do módulo. Por padrão, o layout deve ser colocado no diretório `views/layouts` e deve configurar a propriedade `yii\base\Module::$layout` para apontar o nome do layout. Se você não configurar a propriedade `layout`, o layout da aplicação será usada em seu lugar.

3.8.2 Usando os Módulos

Para usar um módulo em uma aplicação, basta configurar a aplicação, listando o módulo na propriedade `modules` da aplicação. O código da [configuração da aplicação](#) a seguir faz com que o módulo `forum` seja aplicado:

```
[
    'modules' => [
        'forum' => [
            'class' => 'app\modules\forum\Module',
            // ... outras configurações do módulo ...
        ],
    ],
]
```

A propriedade `modules` é composto por um array de configurações de módulos. Cada chave do array representa um *ID do módulo* que identifica exclusivamente o módulo entre todos módulos da aplicação e o valor do array correspondente é uma [configuração](#) para a criação do módulo.

Rotas

Assim como acessar os controllers (controladores) em uma aplicação, as [rotas](#) são usadas para tratar os controllers (controladores) em um módulo. Uma rota para um controller (controlador) dentro de um módulo deve iniciar com o ID do módulo seguido pelo ID do controller (controlador) e pelo ID da

ação. Por exemplo, se uma aplicação usar um módulo chamado `forum`, então a rota `forum/post/index` representará a ação `index` do controller (controlador) `post` no módulo. Se a rota conter apenas o ID do módulo, então a propriedade `yii\base\Module::$defaultRoute`, na qual o valor padrão é `default`, determinará qual controller/action deverá ser usado. Isto significa que a rota `forum` representará o controller (controlador) `default` no módulo `forum`.

Acessando os Módulos

Dentro de um módulo, você poderá precisar muitas vezes obter a instância do módulo da classe para que você possa acessar o ID, os parâmetros, os componentes e etc do módulo. Você pode fazer isso usando a seguinte declaração:

```
$module = MyModuleClass::getInstance();
```

O `MyModuleClass` refere-se ao nome da classe do módulo que você está interessado. O método `getInstance()` retornará a instância que foi solicitada pela requisição. Se o módulo não for solicitado pela requisição, o método retornará `null`. Observe que você não vai querer criar uma nova instância manualmente da classe do módulo pois será diferente do criado pelo Yii em resposta a uma requisição.

Informação: Ao desenvolver um módulo, você não deve assumir que o módulo usará um ID fixo. Isto porque um módulo pode ser associado a um ID arbitrário quando usado em uma aplicação ou dentro de outro módulo. A fim de obter o ID do módulo, você deve usar a abordagem anterior para obter primeiramente a instância do módulo e em seguida obter o ID através de `$module->id`.

Você também pode acessar a instância do módulo usando as seguintes abordagens:

```
// obter o módulo cujo ID é "forum"
$module = \Yii::$app->getModule('forum');

// obter o módulo pelo controller solicitado pela requisição
$module = \Yii::$app->controller->module;
```

A primeira abordagem só é útil quando você sabe o ID do módulo, enquanto a segunda abordagem é melhor utilizada quando você sabe sobre os controllers (controladores) que está sendo solicitado.

Uma vez tendo a instância do módulo, você pode acessar as propriedades e componentes registrados no módulo. Por exemplo,

```
$maxPostCount = $module->params['maxPostCount'];
```

Inicializando os Módulos

Alguns módulos precisam ser executados a cada requisição. O módulo `yii\debug\Module` é um exemplo desta necessidade. Para isto, você deverá listar os IDs deste módulos na propriedade `bootstrap` da aplicação.

Por exemplo, a configuração da aplicação a seguir garante que o módulo `debug` seja sempre carregado:

```
[
    'bootstrap' => [
        'debug',
    ],

    'modules' => [
        'debug' => 'yii\debug\Module',
    ],
]
```

3.8.3 Módulos Aninhados

Os módulos podem ser aninhados em níveis ilimitados. Isto é, um módulo pode conter um outro módulo que pode conter ainda um outro módulo. Nós chamamos dos anteriores de *módulo parente* enquanto os próximos de *módulo filho*. Os módulos filhos devem ser declarados na propriedade `modules` de seus módulos parentes. Por exemplo,

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->modules = [
            'admin' => [
                // você pode considerar em usar um namespace mais curto
                // aqui!
                'class' => 'app\modules\forum\modules\admin\Module',
            ],
        ];
    }
}
```

Para um controller (controlador) dentro de um módulo aninhado, a rota deve incluir os IDs de todos os seus módulos ancestrais. Por exemplo, a rota `forum/admin/dashboard/index` representa a ação `index` do controller (controlador) `dashboard` no módulo `admin` que é um módulo filho do módulo `forum`.

Informação: O método `getModule()` retorna apenas o módulo filho diretamente pertencente ao seu módulo parente. A propriedade `yii\base\Application::$loadedModules` mantém uma

lista dos módulos carregados, incluindo os módulos filhos e parentes aninhados, indexados pelos seus nomes de classes.

3.8.4 Boas Práticas

Os módulos são melhores usados em aplicações de larga escala, cujas características podem ser divididas em vários grupos, cada um constituída por um conjunto de recursos relacionados. Cada grupo de recurso pode ser desenvolvido em um módulo que é desenvolvido e mantido por um desenvolvedor ou equipe específica.

Módulos também são uma boa maneira de reutilizar códigos no nível de grupo de recurso. Algumas recursos comumente utilizados, tais como gerenciamento de usuários, gerenciamento de comentários, podem ser todos desenvolvidos em módulos, para que possam ser utilizados em projetos futuros.

3.9 Filtros

Os filtros são objetos que são executados antes e/ou depois das [ações do controller \(controlador\)](#). Por exemplo, um filtro de controle de acesso pode ser executado antes das ações para garantir que um determinado usuário final tenha autorização de acessá-lo; um filtro de compressão de conteúdo pode ser executado depois das ações para comprimir o conteúdo da resposta antes de enviá-los aos usuários finais.

Um filtro pode ser composto por um pré-filtro (lógicas de filtragem que são aplicadas *antes* que as ações) e/ou um pós-filtro (lógica aplicada *depois* que as ações).

3.9.1 Usando os Filtros

Os filtros são, essencialmente, um tipo especial de [behaviors \(comportamento\)](#). No entanto, o uso dos filtros é igual ao [uso dos behaviors](#). Você pode declarar os filtros em uma classe controller (controlador) sobrescrevendo o método `behaviors()` conforme o exemplo a seguir:

```
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');
            },
        ],
    ],
}
```

```
    1;  
}
```

Por padrão, os filtros declarados em uma classe controller (controlador) serão aplicados em *todas* as ações deste controller (controlador). Você pode, no entanto, especificar explicitamente em quais ações os filtros serão aplicados pela configuração da propriedade **only**. No exemplo anterior, o filtro `HttpCache` só se aplica às ações `index` e `view`. Você também pode configurar a propriedade **except** para montar um blacklist, a fim de barrar algumas ações que estão sendo filtradas.

Além dos controllers (controladores), você também poderá declarar filtros em um **módulo** ou na **aplicação**. Quando você faz isso, os filtros serão aplicados em *todos* as ações do controller (controlador) que pertençam a esse módulo ou a essa aplicação, a menos que você configure as propriedades **only** e **except** do filtro conforme descrito anteriormente.

Observação: Ao declarar os filtros em módulos ou em aplicações, você deve usar **rotas** ao invés de IDs das ações nas propriedades **only** e **except**. Isto porque os IDs das ações não podem, por si só, especificar totalmente as ações no escopo de um módulo ou de uma aplicação.

Quando muitos filtros são configurados para uma única ação, devem ser aplicados de acordo com as seguintes regras:

- Pré-filtragem:
 - Aplica os filtros declarados na aplicação na ordem que foram listados no método `behaviors()`.
 - Aplica os filtros declarados no módulo na ordem que foram listados no método `behaviors()`.
 - Aplica os filtros declarados no controller (controlador) na ordem que foram listados no método `behaviors()`.
 - Se qualquer um dos filtros cancelarem a execução da ação, os filtros (tanto os pré-filtros quanto os pós-filtros) subsequentes não serão aplicados.
- Executa a ação se passar pela pré-filtragem.
- Pós-filtragem
 - Aplica os filtros declarados no controller (controlador) na ordem inversa ao que foram listados no método `behaviors()`.
 - Aplica os filtros declarados nos módulos na ordem inversa ao que foram listados no método `behaviors()`.
 - Aplica os filtros declarados na aplicação na ordem inversa ao que foram listados no método `behaviors()`.

3.9.2 Criando Filtros

Para criar um novo filtro de ação, deve estender a classe `yii\base\ActionFilter` e sobrescrever os métodos `beforeAction()` e/ou `afterAction()`. O primeiro método será executado antes que uma ação seja executada enquanto o outro método será executado após uma ação seja executada. O valor de retorno no método `beforeAction()` determina se uma ação deve ser executada ou não. Se retornar `false`, os filtros subsequentes serão ignorados e a ação não será executada.

O exemplo a seguir mostra um filtro que guarda o log do tempo de execução das ações:

```
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);
    }
}
```

3.9.3 Filtros Nativos

O Yii fornece um conjunto de filtros que normalmente são usados, localizados sob o namespace `yii\filters`. A seguir, iremos realizar uma breve apresentação destes filtros.

AccessControl

O filtro `AccessControl` fornece um controle de acesso simples, baseado em um conjunto de regras. Em particular, antes que uma ação seja executada, o `AccessControl` analisará as regras listadas e localizará o primeiro que corresponda às variáveis do contexto atual (como o IP do usuário, o status do login, etc). A regra correspondente determinará se vai permitir ou não a

execução da ação solicitada. Se nenhuma regra for localizada, o acesso será negado.

O exemplo a seguir mostra como faz para permitir aos usuários autenticados acessarem as ações `create` e `update` enquanto todos os outros não autenticados não consigam acessá-las.

```
use yii\filters\AccessControl;

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::class,
            'only' => ['create', 'update'],
            'rules' => [
                // permite aos usuários autenticados
                [
                    'allow' => true,
                    'roles' => ['@'],
                ],
                // todos os outros usuários são negados por padrão
            ],
        ],
    ];
}
```

De modo geral, para mais detalhes sobre o controle de acesso, por favor, consulte a seção [Autorização](#).

Métodos de Autenticação por Filtros

O método de autenticação por filtros são usados para autenticar um usuário usando vários métodos, tais como HTTP Basic Auth¹⁶, OAuth 2¹⁷. Todas estas classes de filtros estão localizadas sob o namespace `yii\filters\auth`.

O exemplo a seguir mostra como você pode usar o filtro `yii\filters\auth\HttpBasicAuth` para autenticar um usuário usando um acesso baseado em token pelo método HTTP Basic Auth. Observe que, para isto funcionar, sua classe de identidade do usuário deve implementar o método `findIdentityByAccessToken()`.

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    return [
        'basicAuth' => [
            'class' => HttpBasicAuth::class,
```

¹⁶https://en.wikipedia.org/wiki/Basic_access_authentication

¹⁷<https://oauth.net/2/>

```

    ],
  ];
}

```

Os métodos de autenticação por filtros geralmente são utilizados na implementação de APIs RESTful. Para mais detalhes, por favor, consulte a seção RESTful [Autenticação](#).

ContentNegotiator

O filtro ContentNegotiator suporta a identificação de formatos de respostas e o idioma da aplicação. Este filtro tenta determinar o formato de resposta e o idioma analisando os parâmetros GET e o Accept do cabeçalho HTTP.

No exemplo a seguir, o ContentNegotiator está sendo configurado para suportar os formatos de resposta JSON e XML, e os idiomas Inglês (Estados Unidos) e Alemão.

```

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',
            ],
        ],
    ];
}

```

Os formatos de resposta e os idiomas muitas vezes precisam ser determinados muito mais cedo no [ciclo de vida da aplicação](#). Por este motivo, o ContentNegotiator foi projetado para ser usado de outras formas, onde pode ser usado tanto como um [componente de inicialização](#) quanto um filtro. Por exemplo, você pode configurá-lo na [configuração da aplicação](#) conforme o exemplo a seguir:

```

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',
        ],
    ],
];

```

Informação: Nos casos do formato de resposta e do idioma não serem determinados pela requisição, o primeiro formato e idioma listados em `formats` e `languages` serão utilizados.

HttpCache

O filtro `HttpCache` implementa no lado do cliente (client-side) o cache pela utilização dos parâmetros `Last-Modified` e `Etag` do cabeçalho HTTP. Por exemplo,

```

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');
            },
        ],
    ];
}

```

Por favor, consulte a seção [Cache HTTP](#) para mais detalhes sobre o uso do `HttpCache`.

PageCache

O filtro `PageCache` implementa no lado do servidor (server-side) o cache das páginas. No exemplo a seguir, o `PageCache` é aplicado para a ação `index` guardar o cache da página inteira por no máximo 60 segundos ou até que a quantidade de registros na tabela `post` seja alterada. Este filtro também guarda diferentes versões da página, dependendo do idioma da aplicação escolhido.

```
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,
            ]
        ],
    ];
}
```

Por favor, consulte a seção [Cache de Página](#) para mais detalhes sobre o uso do PageCache.

RateLimiter

O filtro RateLimiter implementa um limitador de acesso baseado no algoritmo do balde furado (leaky bucket)¹⁸. É usado principalmente na implementação de APIs RESTful. Por favor, consulte a seção [Limitador de Acesso](#) para mais detalhes sobre o uso deste filtro.

VerbFilter

O filtro VerbFilter verifica se os métodos de requisição HTTP são permitidos para as ações solicitadas. Se não for, será lançada uma exceção HTTP 405. No exemplo a seguir, o VerbFilter é declarado para especificar um conjunto de métodos de requisição permitidos para as ações CRUD.

```
use yii\filters\VerbFilter;

public function behaviors()
{
    return [
        'verbs' => [
            'class' => VerbFilter::class,
            'actions' => [
                'index' => ['get'],
                'view' => ['get'],
                'create' => ['get', 'post'],
            ]
        ]
    ];
}
```

¹⁸https://pt.wikipedia.org/wiki/Leaky_Bucket

```

        'update' => ['get', 'put', 'post'],
        'delete' => ['post', 'delete'],
    ],
],
];
}

```

Cors

O compartilhamento de recursos cross-origin CORS¹⁹ é um mecanismo que permite vários recursos (por exemplo, fontes, JavaScript, etc) na página Web sejam solicitados por outros domínios. Em particular, as chamadas AJAX do JavaScript podem usar o mecanismo XMLHttpRequest. Estas chamadas “cross-domain” são proibidas pelos navegadores Web, por desrespeitarem a política de segurança de origem. O CORS define um modo em que o navegador e o servidor possam interagir para determinar se deve ou não permitir as requisições cross-origin.

O filtro `Cors` deve ser definido antes dos filtros de Autenticação/Autorização para garantir que os cabeçalhos CORS sejam sempre enviados.

```

use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::class,
        ],
    ], parent::behaviors());
}

```

A filtragem da classe `Cors` pode ser ajustado pela propriedade `cors`.

- `cors['Origin']`: array usado para definir as origens permitidas. Pode ser ['*'] (qualquer um) ou ['https://www.myserver.net', 'https://www.myotherserver.com']. O padrão é ['*'].
- `cors['Access-Control-Request-Method']`: array com os métodos de requisição permitidos, tais como ['GET', 'OPTIONS', 'HEAD']. O padrão é ['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS'].
- `cors['Access-Control-Request-Headers']`: array com os cabeçalhos permitidos. Pode ser ['*'] para todos os cabeçalhos ou um específico como ['X-Request-With']. O padrão é ['*'].
- `cors['Access-Control-Allow-Credentials']`: define se a requisição atual pode ser feita usando credenciais. Pode ser `true`, `false` ou `null` (não definida). O padrão é `null`.

¹⁹<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/CORS>

- `cors['Access-Control-Max-Age']`: define o tempo de vida do pré-processamento (pre-flight) da requisição. O padrão é 86400.

Por exemplo, permitindo CORS para a origem: `https://www.myserver.net` com os métodos GET, HEAD e 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());
}
```

Você pode ajustar os cabeçalhos do CORS sobrescrevendo os parâmetros padrão para cada ação. Por exemplo, para adicionar o parâmetro `Access-Control-Allow-Credentials` somente na ação login, você poderia fazer conforme a seguir:

```
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());
}
```

3.10 Widgets

Os widgets são blocos de construção reutilizáveis usados nas [views](#) (visões) para criar e configurar complexos elementos de interface do usuário sob uma

modelagem orientada a objetos. Por exemplo, um widget datapicker pode gerar um calendário que permite aos usuários selecionarem uma data que desejam inserir em um formulário. Tudo o que você precisa fazer é apenas inserir um código na view (visão) conforme o seguinte:

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget(['name' => 'date']) ?>
```

Existe uma quantidade considerável de widgets empacotados no Yii, como o `active form`, o `menu`, o `jQuery UI widgets`, o `Twitter Bootstrap widgets`, etc. A seguir, iremos introduzir os conhecimentos básicos sobre os widgets. Por favor, consulte a documentação de classes da API se você quiser saber mais sobre o uso de um determinado widget.

3.10.1 Usando Widgets

Os widgets são usados principalmente nas [views \(visões\)](#). Você pode chamar o método `yii\base\Widget::widget()` para usar um widget em uma view (visão). O método possui um array de [configuração](#) para inicializar o widget e retornar o resultado da renderização do widget. Por exemplo, o código a seguir insere um widget datapicker configurado para usar o idioma Russo e manter a data selecionada no atributo `from_date` do `$model`.

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget([
    'model' => $model,
    'attribute' => 'from_date',
    'language' => 'ru',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]) ?>
```

Alguns widgets podem ter um bloco de conteúdo que deve ser colocado entre as chamadas dos métodos `yii\base\Widget::begin()` e `yii\base\Widget::end()`. Por exemplo, o código a seguir usa o widget `yii\widgets\ActiveForm` para gerar um formulário de login. O widget irá gerar as tags de abertura e de fechamento do `<form>` respectivamente nos lugares onde os métodos `begin()` e `end()` foram chamados. Qualquer conteúdo entre estes métodos serão renderizados entre as tags de abertura e de fechamento do `<form>`.

```
<?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(); ?>

```

Observe que ao contrário do `yii\base\Widget::widget()` que retorna a renderização de um widget, o método `yii\base\Widget::begin()` retorna uma instância do widget que pode ser usado para construir o seu conteúdo.

3.10.2 Criando Widgets

Para criar um widget, estenda a classe `yii\base\Widget` e sobrescreva os métodos `yii\base\Widget::init()` e/ou `yii\base\Widget::run()`. Normalmente, o método `init()` deve conter os códigos que normalizam as propriedades do widget, enquanto o método `run()` deve conter o código que gera o resultado da renderização do widget. O resultado da renderização pode ser feito diretamente dando “echo” ou pelo retorno de uma string no método `run()`.

No exemplo a seguir, o `HelloWidget` codifica o HTML e exibe o conteúdo atribuído à sua propriedade `message`. Se a propriedade não for definida, será exibido “Hello World” como padrão.

```

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);
    }
}

```

Para usar este widget, simplesmente insira o código a seguir em uma view (visão):

```

<?php
use app\components\HelloWidget;
?>
<?= HelloWidget::widget(['message' => 'Good morning']) ?>

```

O `HelloWidget` abaixo é uma variante que pega o conteúdo entre as chamadas de `begin()` e `end()`, codifica o HTML e em seguida os exibe.

```

namespace app\components;

use yii\base\Widget;
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);
    }
}

```

Como você pode ver, o buffer de saída do PHP é iniciado no método `init()` para que qualquer conteúdo entre as chamadas de `init()` e `run()` possam ser capturadas, processadas e retornadas em `run()`.

Informação: Ao chamar o `yii\base\Widget::begin()`, uma nova instância do widget será criada e o método `init()` será chamado logo ao final de seu construtor. Ao chamar o `yii\base\Widget::end()`, o método `run()` será chamado cujo o resultado da renderização será dado *echo* pelo `end()`.

O código a seguir mostra como você pode usar esta nova variante do `HelloWidget`:

```

<?php
use app\components\HelloWidget;
?>
<?php HelloWidget::begin(); ?>

```

um conteúdo qualquer...

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

Algumas vezes, um widget pode precisar renderizar um grande conteúdo. Enquanto você pode inserir todo este conteúdo no método `run()`, uma boa prática é colocá-lo em uma [view \(visão\)](#) e chamar o `yii\base\Widget::render()` para renderizá-lo. Por exemplo,

```
public function run()
{
    return $this->render('hello');
}
```

Por padrão, as views (visões) para um widget devem ser armazenadas em arquivos sob o diretório `WidgetPath/views`, onde o `WidgetPath` significa o diretório que contém os arquivos da classe do widget. Portanto, o exemplo anterior irá renderizar o arquivo de view (visão) `@app/components/views/hello.php`, assumindo que a classe widget está localizada sob o diretório `@app/components`. Você pode sobrescrever o método `yii\base\Widget::getViewPath()` para personalizar o diretório que conterá os arquivos de views (visões) do widget.

3.10.3 Boas Práticas

Os widgets são uma maneira orientada a objetos de reutilizar códigos de view (visão).

Ao criar os widgets, você ainda deve seguir o padrão MVC. Em geral, você deve manter a lógica nas classes widgets e manter as apresentações nas [views \(visões\)](#).

Os widgets devem ser projetados para serem autossuficientes. Isto é, ao utilizar um widget, você deverá ser capaz de removê-lo de uma view (visão) sem fazer qualquer outra coisa. Isto pode ser complicado se um widget requerer recursos externos, tais como CSS, JavaScript, imagens, etc. Felizmente, o Yii fornece o suporte para [asset bundles](#), que pode ser utilizado para resolver este problema.

Quando um widget contiver somente código de view (visão), será bem semelhante a uma [view \(visão\)](#). Na verdade, neste caso, a única diferença é que um widget é uma classe para ser redistribuída, enquanto uma view é apenas um simples script PHP que você preferirá manter em sua aplicação

3.11 Assets

Um asset no Yii é um arquivo que pode ser referenciado em uma página Web. Pode ser um arquivo CSS, JavaScript, imagem, vídeo, etc. Os assets

estão localizados em um diretório acessível pela Web e estão diretamente disponibilizados por servidores Web.

Muitas vezes, é preferível gerenciá-los programaticamente. Por exemplo, quando você usar o widget `yii\jui\DatePicker` em uma página, será incluído automaticamente os arquivos CSS e JavaScript requeridos, ao invés de pedir para você encontrar estes arquivos e incluí-los manualmente. E quando você atualizar o widget para uma nova versão, automaticamente usará a nova versão dos arquivos de assets. Neste tutorial, iremos descrever esta poderosa capacidade de gerência de assets fornecidas pelo Yii.

3.11.1 Asset Bundles

O Yii gerencia os assets na unidade de *asset bundle*. Um asset bundle é simplesmente uma coleção de assets localizados em um diretório. Quando você registrar um asset bundle em uma [view \(visão\)](#), serão incluídos os arquivos CSS e JavaScript do bundle na página Web renderizada.

3.11.2 Definindo os Asset Bundles

Os asset bundles são especificados como classes PHP que estendem de `yii\web\AssetBundle`. O nome de um bundle corresponde simplesmente a um nome de classe PHP totalmente qualificada (sem a primeira barra invertida). Uma classe de asset bundle deve ser [autoloadable](#). Geralmente é especificado onde os asset estão localizados, quais arquivos CSS e JavaScript possuem e como o bundle depende de outro bundles.

O código a seguir define o asset bundle principal que é usado pelo [template básico de projetos](#):

```
<?php

namespace app\assets;

use yii\web\AssetBundle;

class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.css',
    ];
    public $js = [
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

A classe `AppAsset` acima, especifica que os arquivos de assets estão localizadas sob o diretório `@webroot` que corresponde à URL `@web`; O bundle contém um único arquivo CSS `css/site.css` e nenhum arquivo JavaScript; O bundle depende de outros dois bundles: `yii\web\YiiAsset` e `yii\bootstrap\BootstrapAsset`. Mais detalhes sobre as propriedades do `yii\web\AssetBundle` serão encontradas a seguir:

- **sourcePath**: especifica o diretório que contém os arquivos de assets neste bundle. Esta propriedade deve ser definida se o diretório root não for acessível pela Web. Caso contrário, você deve definir as propriedades **basePath** e **baseUrl**. Os [alias de caminhos](#) podem ser usados nesta propriedade.
- **basePath**: especifica um diretório acessível pela Web que contém os arquivos de assets neste bundle. Quando você especificar a propriedade **sourcePath**, o gerenciador de asset publicará os assets deste bundle para um diretório acessível pela Web e sobrescreverá a propriedade **basePath** para ficar em conformidade. Você deve definir esta propriedade caso os seus arquivos de asset já estejam em um diretório acessível pela Web e não precisam ser publicados.
As [alias de caminhos](#) podem ser usados aqui.
- **baseUrl**: especifica a URL correspondente ao diretório **basePath**. Assim como a propriedade **basePath**, se você especificar a propriedade **sourcePath**, o gerenciador de asset publicará os assets e sobrescreverá esta propriedade para entrar em conformidade. Os [alias de caminhos](#) podem ser usados aqui.
- **js**: um array listando os arquivos JavaScript contidos neste bundle. Observe que apenas a barra “/” pode ser usada como separadores de diretórios. Cada arquivo JavaScript deve ser especificado em um dos dois seguintes formatos:
 - um caminho relativo representando um local do arquivo JavaScript (por exemplo, `js/main.js`). O caminho real do arquivo pode ser determinado pela precedência do `yii\web\AssetManager::$basePath` no caminho relativo e a URL real do arquivo pode ser determinado pela precedência do `yii\web\AssetManager::$baseUrl` no caminho relativo.
 - uma URL absoluta representando um arquivo JavaScript externo. Por exemplo, `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`.
- **css**: uma array listando os arquivos CSS contidos neste bundle. O formato deste array é igual ao que foi mencionado no **js**.
- **depends**: um array listando os nomes dos asset bundles que este bundle depende (será explicado em breve).
- **jsOptions**: especifica as opções que serão passadas para o método `yii\web\View::registerJsFile()` quando for chamado para regis-

trar *cada* arquivo JavaScript neste bundle.

- **cssOptions**: especifica as opções que serão passadas para o método `yii\web\View::registerCssFile()` quando for chamado para registrar *cada* arquivo CSS neste bundle.
- **publishOptions**: especifica as opções que serão passadas para o método `yii\web\AssetManager::publish()` quando for chamado para publicar os arquivos de asset para um diretório Web. Este é usado apenas se você especificar a propriedade **sourcePath**.

Localização dos Assets

Os assets, com base em sua localização, podem ser classificados como:

- **assets fonte**: os arquivos de asset estão localizados juntos ao código fonte PHP que não podem ser acessados diretamente na Web. Para utilizar os assets fonte em uma página, devem ser copiados para um diretório Web a fim de torna-los como os chamados assets publicados. Este processo é chamado de *publicação de asset* que será descrito em detalhes ainda nesta seção.
- **assets publicados**: os arquivos de asset estão localizados em um diretório Web e podendo, assim, serem acessados diretamente na Web.
- **assets externos**: os arquivos de asset estão localizados em um servidor Web diferente do que a aplicação está hospedada.

Ao definir uma classe de asset bundle e especificar a propriedade **sourcePath**, significará que quaisquer assets listados usando caminhos relativos serão considerados como assets fonte. Se você não especificar esta propriedade, significará que estes assets serão assets publicados (portanto, você deve especificar as propriedades **basePath** e **baseUrl** para deixar o Yii saber onde eles estão localizados).

É recomendado que você coloque os assets da aplicação em um diretório Web para evitar o processo de publicação de assets desnecessários. É por isso que o **AppAsset** do exemplo anterior especifica a propriedade **basePath** ao invés da propriedade **sourcePath**.

Para as **extensões**, por seus assets estarem localizados juntamente com seus códigos fonte em um diretório não acessível pela Web, você terá que especificar a propriedade **sourcePath** ao definir as classes de asset bundle.

Observação: Não use o `@webroot/assets` como o **caminho da fonte**. Este diretório é usado por padrão pelo **gerenciador de asset** para salvar os arquivos de asset publicados a partir de seu local de origem. Qualquer conteúdo deste diretório será considerado como temporário e podem estar sujeitos a serem deletados.

Dependências de Assets

Ao incluir vários arquivos CSS ou JavaScript em uma página Web, devem seguir uma determinada ordem para evitar problemas de sobrescritas. Por exemplo, se você estiver usando um widget JQuery UI em um página, você deve garantir que o arquivo JavaScript do JQuery esteja incluído antes que o arquivo JavaScript do JQuery UI. Chamamos esta tal ordenação de dependência entre os assets.

A dependência de assets são especificados principalmente através da propriedade `yii\web\AssetBundle::$depends`. No exemplo do `AppAsset`, o asset bundle depende de outros dois asset bundles: `yii\web\YiiAsset` e `yii\bootstrap\BootstrapAsset`, o que significa que os arquivos CSS e JavaScript do `AppAsset` serão incluídos *após* a inclusão dos arquivos dos dois bundles dependentes.

As dependências de assets são transitivas. Isto significa que se um asset bundle A depende de B e que o B depende de C, o A também dependerá de C.

Opções do Asset

Você pode especificar as propriedades `cssOptions` e `jsOptions` para personalizar o modo que os arquivos CSS e JavaScript serão incluídos em uma página. Os valores destas propriedades serão passadas respectivamente para os métodos `yii\web\View::registerCssFile()` e `yii\web\View::registerJsFile()`, quando forem chamados pela *view* (visão) para incluir os arquivos CSS e JavaScript.

Observação: As opções definidas em uma classe bundle aplicam-se para *todos* os arquivos CSS/JavaScript de um bundle. Se você quiser usar opções diferentes para arquivos diferentes, você deve criar asset bundles separados e usar um conjunto de opções para cada bundle.

Por exemplo, para incluir condicionalmente um arquivo CSS para navegadores IE9 ou mais antigo, você pode usar a seguinte opção:

```
public $cssOptions = ['condition' => 'lte IE9'];
```

Isto fara com que um arquivo CSS do bundle seja incluído usando as seguintes tags HTML:

```
<!--[if lte IE9]>
<link rel="stylesheet" href="path/to/foo.css">
<![endif]-->
```

Para envolver as tags links do CSS dentro do `<noscript>`, você poderá configurar o `cssOptions` da seguinte forma,

```
public $cssOptions = ['noscript' => true];
```

Para incluir um arquivo JavaScript na seção <head> de uma página (por padrão, os arquivos JavaScript são incluídos no final da seção <body>, use a seguinte opção:

```
public $jsOptions = ['position' => \yii\web\View::POS_HEAD];
```

Por padrão, quando um asset bundle está sendo publicado, todo o conteúdo do diretório especificado pela propriedade `yii\web\AssetBundle::$sourcePath` serão publicados. Para você personalizar este comportamento configurando a propriedade `publishOptions`. Por exemplo, para publicar apenas um ou alguns subdiretórios do `yii\web\AssetBundle::$sourcePath`, você pode fazer a seguinte classe de asset bundle.

```
<?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 function init()
    {
        parent::init();
        $this->publishOptions['beforeCopy'] = function ($from, $to) {
            $dirname = basename(dirname($from));
            return $dirname === 'fonts' || $dirname === 'css';
        };
    }
}
```

O exemplo anterior define um asset bundle para o pacote de “fontawesome”²⁰. Ao especificar a opção de publicação `beforeCopy`, apenas os subdiretórios `fonts` e `css` serão publicados.

Assets do Bower e NPM

A maioria dos pacotes JavaScript/CSS são gerenciados pelo Bower²¹ e/ou NPM²². Se sua aplicação ou extensão estiver usando um destes pacotes, é recomendado que você siga os passos a seguir para gerenciar os assets na biblioteca:

²⁰<https://fontawesome.com/>

²¹<https://bower.io/>

²²<https://www.npmjs.com/>

1. Modifique o arquivo de sua aplicação ou extensão e informe os pacotes na entrada `require`. Você deve usar `bower-asset/PackageName` (para pacotes Bower) ou `npm-asset/PackageName` (para pacotes NPM) para referenciar à biblioteca.
2. Crie uma classe asset bundle e informe os arquivos JavaScript/CSS que você pretende usar em sua aplicação ou extensão. Você deve especificar a propriedade `sourcePath` como `@bower/PackageName` ou `@npm/PackageName`. Isto porque o Composer irá instalar os pacotes Bower ou NPM no diretório correspondente a estas alias.

Observação: Alguns pacotes podem colocar todos os seus arquivos distribuídos em um subdiretório. Se este for o caso, você deve especificar o subdiretório como o valor da propriedade `sourcePath`. Por exemplo, o `yii\web\jQueryAsset` usa `@bower/jquery/dist` ao invés de `@bower/jquery`.

3.11.3 Usando Asset Bundles

Para usar um asset bundle, registre uma [view \(visão\)](#) chamando o método `yii\web\AssetBundle::register()`. Por exemplo, no template da view (visão) você pode registrar um asset bundle conforme o exemplo a seguir:

```
use app\assets\AppAsset;  
AppAsset::register($this); // $this representa o objeto da view (visão)
```

Informação: O método `yii\web\AssetBundle::register()` retorna um objeto asset bundle contendo informações sobre os assets publicados, tais como o `basePath` ou `baseUrl`.

Se você estiver registrando um asset bundle em outros lugares, você deve fornecer o objeto da view (visão) necessário. Por exemplo, para registrar um asset bundle em uma classe [widget](#), você pode obter o objeto da view (visão) pelo `$this->view`.

Quando um asset bundle for registrado em um view (visão), o Yii registrará todos os seus asset bundles dependentes. E, se um asset bundle estiver localizado em um diretório inacessível pela Web, será publicado em um diretório Web. Em seguida, quando a view (visão) renderizar uma página, será gerado as tags `<link>` e `<script>` para os arquivos CSS e JavaScript informados nos bundles registrados. A ordem destas tags são determinados pelas dependências dos bundles registrados e pela ordem dos assets informados nas propriedades `yii\web\AssetBundle::$css` e `yii\web\AssetBundle::$js`.

Personalizando os Asset Bundles

O Yii gerencia os asset bundles através do componente de aplicação chamado `assetManager` que é implementado pelo `yii\web\AssetManager`. Ao

configurar a propriedade `yii\web\AssetManager::$bundles`, é possível personalizar o comportamento de um asset bundle. Por exemplo, o asset bundle padrão `yii\web\JqueryAsset` usa o arquivo `jquery.js` do pacote JQuery instalado pelo Bower. Para melhorar a disponibilidade e o desempenho, você pode querer usar uma versão hospedada pelo Google. Isto pode ser feito configurando o `assetManager` na configuração da aplicação conforme o exemplo a seguir:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => [
                    'sourcePath' => null,    // do not publish the bundle
                    'js' => [
                        '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',
                    ],
                ],
            ],
        ],
    ],
];
```

Você pode configurar diversos asset bundles de forma semelhante através da propriedade `yii\web\AssetManager::$bundles`. As chaves do array devem ser os nomes das classes (sem a barra invertida) dos asset bundles e os valores do array devem corresponder aos [arrays de configuração](#).

Dica: Você pode, de forma condicional, escolher os assets que queira usar em um asset bundle. O exemplo a seguir mostra como usar o `jquery.js` no ambiente de desenvolvimento e o `jquery.min.js` em outra situação:

```
'yii\web\JqueryAsset' => [
    'js' => [
        YII_ENV_DEV ? 'jquery.js' : 'jquery.min.js'
    ],
],
```

Você pode desabilitar um ou vários asset bundles, associando `false` aos nomes dos asset bundles que queira ser desabilitado. Ao registrar um asset bundle desabilitado em um view (visão), nenhuma das suas dependências serão registradas e a view (visão) também não incluirá quaisquer assets do bundle na página renderizada. Por exemplo, para desabilitar o `yii\web\JqueryAsset`, você pode usando a seguinte configuração.

```
return [
    // ...
```

```

        'components' => [
            'assetManager' => [
                'bundles' => [
                    'yii\web\jQueryAsset' => false,
                ],
            ],
        ],
    ];

```

Você também pode desabilitar *todos* os asset bundles definindo o `yii\web\AssetManager::$bundles` como `false`.

Mapeando Asset

Às vezes, você pode querer “corrigir” os caminhos dos arquivos de asset incorretos/incompatíveis em vários asset bundles. Por exemplo, o bundle A usa o `jquery.min.js` com a versão 1.11.1 e o bundle B usa o `jquery.js` com a versão 2.1.1. Embora você possa corrigir o problema personalizando cada bundle, existe um modo mais simples usando o recurso de *mapeamento de asset* para mapear todos os assets incorretos para os assets desejados de uma vez. Para fazer isso, configure a propriedade `yii\web\AssetManager::$assetMap` conforme o exemplo a seguir:

```

return [
    // ...
    'components' => [
        'assetManager' => [
            'assetMap' => [
                'jquery.js' =>
                    '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',
            ],
        ],
    ],
];

```

As chaves do array `assetMap` são os nomes dos assets que você deseja corrigir e o valor são os caminhos dos assets desejados. Ao registrar um asset bundle em uma view (visão), cada arquivo de asset relativo aos arrays `css` e `js` serão examinados a partir deste mapeamento. Se qualquer uma das chaves forem encontradas para serem a última parte de um arquivo de asset (que é prefixado com o `yii\web\AssetBundle::$sourcePath` se disponível), o valor correspondente substituirá o asset a ser registrado na view (visão). Por exemplo, o arquivo de asset `my/path/to/jquery.js` corresponde a chave a chave `jquery.js`.

Observação: Apenas os assets especificados usando caminhos relativos estão

sujeitos ao mapeamento de assets. O caminho dos assets devem ser URLs absolutas ou caminhos relativos ao caminho da propriedade `yii\web\AssetManager::$basePath`.

Publicação de Asset

Como mencionado anteriormente, se um asset bundle for localizado em um diretório que não é acessível pela Web, os seus assets serão copiados para um diretório Web quando o bundle estiver sendo registrado na view (visão). Este processo é chamado de *publicação de asset* e é feito automaticamente pelo **gerenciador de asset**.

Por padrão, os assets são publicados para o diretório `@webroot/assets` que corresponde a URL `@web/assets`. Você pode personalizar este local configurando as propriedades `basePath` e `baseUrl`.

Ao invés de publicar os assets pela cópia de arquivos, você pode considerar o uso de links simbólicos, caso o seu sistema operacional e o servidor Web permita-os. Este recurso pode ser habilitado definindo o `linkAssets` como `true`.

```
return [  
    // ...  
    'components' => [  
        'assetManager' => [  
            'linkAssets' => true,  
        ],  
    ],  
];
```

Com a configuração acima, o gerenciador de asset irá criar um link simbólico para o caminho fonte de um asset bundle quando estiver sendo publicado. Isto é mais rápido que a cópia de arquivos e também pode garantir que os assets publicados estejam sempre atualizados.

Cache Busting

Para aplicações Web que estão rodando no modo produção, é uma prática comum habilitar o cache HTTP para assets e outros recursos estáticos. A desvantagem desta prática é que sempre que você modificar um asset e implantá-lo em produção, um cliente pode ainda estar usando a versão antiga, devido ao cache HTTP. Para superar esta desvantagem, você pode utilizar o recurso cache busting, que foi implementado na versão 2.0.3, configurando o `yii\web\AssetManager` como mostrado a seguir:

```
return [  
    // ...  
    'components' => [  
        'assetManager' => [  
            'appendTimestamp' => true,  
        ],  
    ],  
];
```

Ao fazer isto, a URL de cada asset publicado será anexada ao seu último horário de modificação. Por exemplo, a URL do `yii.js` pode parecer com `/assets/5515a87c/yii.js?v=1423448645`, onde o parâmetro `v` representa o último horário de modificação do arquivo `yii.js`. Agora se você modificar um asset, a sua URL será alterada, fazendo com que o cliente busque a versão mais recente do asset.

3.11.4 Asset Bundles de Uso Comum

O código nativo do Yii definiu vários asset bundles. Entre eles, os bundles a seguir são de uso comum e podem ser referenciados em sua aplicação ou no código de extensão.

- `yii\web\YiiAsset`: Inclui principalmente o arquivo `yii.js` que implementa um mecanismo para organizar os códigos JavaScript em módulos. Ele também fornece um suporte especial para os atributos `data-method` e `data-confirm` e outros recursos úteis.
- `yii\web\jQueryAsset`: Inclui o arquivo `jquery.js` do pacote jQuery do Bower.
- `yii\bootstrap\BootstrapAsset`: Inclui o arquivo CSS do framework Twitter Bootstrap.
- `yii\bootstrap\BootstrapPluginAsset`: Inclui o arquivo JavaScript do framework Twitter Bootstrap para dar suporte aos plug-ins JavaScript do Bootstrap.
- `yii\jui\JuiAsset`: Inclui os arquivos CSS e JavaScript do biblioteca jQuery UI.

Se o seu código depende do jQuery, jQuery UI ou Bootstrap, você deve usar estes asset bundles predefinidos ao invés de criar suas próprias versões. Se a definição padrão destes bundles não satisfizer o que precisa, você pode personaliza-los conforme descrito na subseção Personalizando os Asset Bundles.

3.11.5 Conversão de Assets

Ao invés de escrever diretamente códigos CSS e/ou JavaScript, os desenvolvedores geralmente os escrevem em alguma sintaxe estendida e usam ferramentas especiais para convertê-los em CSS/JavaScript. Por exemplo, para o código CSS você pode usar LESS²³ ou SCSS²⁴; e para o JavaScript você pode usar o TypeScript²⁵.

Você pode listar os arquivos de asset em sintaxe estendida nas propriedades `css` e `js` de um asset bundle. Por exemplo,

²³<https://lesscss.org/>

²⁴<https://sass-lang.com/>

²⁵<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',
    ];
}

```

Ao registrar um determinado asset bundle em uma view (visão), o **gerenciador de asset** automaticamente rodará as ferramentas de pré-processamento para converter os assets de sintaxe estendida em CSS/JavaScript. Quando a view (visão) finalmente renderizar uma página, será incluído os arquivos CSS/JavaScript convertidos ao invés dos arquivos de assets originais em sintaxe estendida.

O Yii usa as extensões dos nomes de arquivos para identificar se é um asset com sintaxe estendida. Por padrão, o Yii reconhecerá as seguintes sintaxes e extensões de arquivos:

- LESS²⁶: `.less`
- SCSS²⁷: `.scss`
- Stylus²⁸: `.styl`
- CoffeeScript²⁹: `.coffee`
- TypeScript³⁰: `.ts`

O Yii conta com ferramentas de pré-processamento instalados para converter os assets. Por exemplo, para usar o LESS³¹ você deve instalar o comando de pré-processamento `lessc`.

Você pode personalizar os comandos de pré-processamento e o da sintaxe estendida suportada configurando o `yii\web\AssetManager::$converter` conforme o exemplo a seguir:

```

return [
    'components' => [
        'assetManager' => [
            'converter' => [
                'class' => 'yii\web\AssetConverter',

```

²⁶<https://lesscss.org/>

²⁷<https://sass-lang.com/>

²⁸<https://stylus-lang.com/>

²⁹<https://coffeescript.org/>

³⁰<https://www.typescriptlang.org/>

³¹<https://lesscss.org/>

```

        'commands' => [
            'less' => ['css', 'lessc {from} {to} --no-color'],
            'ts' => ['js', 'tsc --out {to} {from}'],
        ],
    ],
],
];

```

No exemplo acima, especificamos a sintaxe estendida suportada pela propriedade `yii\web\AssetConverter::$commands`. As chaves do array correspondem a extensão dos arquivos (sem o ponto a esquerda) e o valor do array possui a extensão do arquivo de asset resultante e o comando para executar a conversão do asset. Os tokens `{from}` e `{to}` nos comandos serão substituídos pelo caminho do arquivo de asset fonte e pelo caminho do arquivo de asset de destino.

Informação: Existem outros modos de trabalhar com assets em sintaxe estendida, além do descrito acima. Por exemplo, você pode usar ferramentas de compilação tais como o `grunt`³² para monitorar e automatizar a conversão de assets em sintaxe estendidas. Neste caso, você deve listar os arquivos de CSS/JavaScript resultantes nos asset bundles ao invés dos arquivos originais.

3.11.6 Combinando e Comprimindo Assets

Uma página Web pode incluir muitos arquivos CSS e/ou JavaScript. Para reduzir o número de requisições HTTP e o tamanho total de downloads destes arquivos, uma prática comum é combinar e comprimir vários arquivos CSS/JavaScript em um ou em poucos arquivos e em seguida incluir estes arquivos comprimidos nas páginas Web ao invés dos originais.

Informação: A combinação e compressão de assets normalmente são necessárias quando uma aplicação está em modo de produção. No modo de desenvolvimento, usar os arquivos CSS/JavaScript originais muitas vezes são mais convenientes para depuração.

A seguir, apresentaremos uma abordagem para combinar e comprimir arquivos de assets sem precisar modificar o código da aplicação existente.

1. Localize todos os asset bundles em sua aplicação que você deseja combinar e comprimir.
2. Divida estes bundles em um ou alguns grupos. Observe que cada bundle pode apenas pertencer a um único grupo.

³²<https://gruntjs.com/>

3. Combinar/Comprimir os arquivos CSS de cada grupo em um único arquivo. Faça isto de forma semelhante para os arquivos JavaScript.
4. Defina um novo asset bundle para cada grupo:
 - Defina as propriedades `css` e `js` com os arquivos CSS e JavaScript combinados, respectivamente.
 - Personalize os asset bundles de cada grupo definindo as suas propriedades `css` e `js` como vazias e definindo a sua propriedade `depends` para ser o novo asset bundle criado para o grupo.

Usando esta abordagem, quando você registrar um asset bundle em uma view (visão), fará com que registre automaticamente o novo asset bundle do grupo que o bundle original pertence. E, como resultado, os arquivos de asset combinados/comprimidos serão incluídos na página, ao invés dos originais.

Um Exemplo

Vamos usar um exemplo para explicar melhor o exemplo acima:

Assuma que sua aplicação possua duas páginas, X e Y. A página X usa os asset bundles A, B e C, enquanto a página Y usa os asset bundles B, C e D.

Você tem duas maneiras de dividir estes asset bundles. Uma delas é a utilização de um único grupo para incluir todos os asset bundles e a outra é colocar o A no Grupo X, o D no Grupo Y e (B, C) no Grupo S. Qual deles é o melhor? Isto depende. A primeira maneira tem a vantagem de ambas as páginas compartilharem os mesmos arquivos CSS e JavaScript combinados, o que torna o cache HTTP mais eficaz. Por outro lado, pelo fato de um único grupo conter todos os bundles, o tamanho dos arquivos CSS e JavaScript combinados será maior e, assim, aumentará o tempo de carregamento inicial. Para simplificar este exemplo, vamos usar a primeira maneira, ou seja, usaremos um único grupo para conter todos os bundles.

Informação: Dividir os asset bundles em grupos não é uma tarefa trivial. Geralmente requer que análise sobre o real tráfego de dados de diversos assets em páginas diferentes. Para começar, você pode usar um único grupo para simplificar.

Use as ferramentas existentes (por exemplo, Closure Compiler³³, YUI Compressor³⁴) para combinar e comprimir os arquivos CSS e JavaScript em todos os bundles. Observe que os arquivos devem ser combinados na ordem que satisfaça as dependências entre os bundles. Por exemplo, se o bundle A depende do B e que dependa tanto do C quanto do D, você deve listar os arquivos de asset a partir do C e D, em seguida pelo B e finalmente pelo A.

³³<https://developers.google.com/closure/compiler/>

³⁴<https://github.com/yui/yuicompressor/>

Depois de combinar e comprimir, obteremos um arquivo CSS e um arquivo JavaScript. Suponha que os arquivos serão chamados de `all-xyz.css` e `all-xyz.js`, onde o `xyz` significa um timestamp ou um hash que é usado para criar um nome de arquivo único para evitar problemas de cache HTTP.

Nós estamos na última etapa agora. Configure o **gerenciador de asset** como o seguinte na configuração da aplicação:

```
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']],
            ],
        ],
    ],
];
```

Como foi explicado na subseção Personalizando os Asset Bundles, a configuração acima altera o comportamento padrão de cada bundle. Em particular, os bundles A, B, C e D não precisam mais de arquivos de asset. Agora todos dependem do bundle `all` que contém os arquivos `all-xyz.css` e `all-xyz.js` combinados. Consequentemente, para a página X, ao invés de incluir os arquivos fontes originais dos bundles A, B e C, apenas estes dois arquivos combinados serão incluídos; a mesma coisa acontece com a página Y.

Existe um truque final para fazer o trabalho da abordagem acima de forma mais simples. Ao invés de modificar diretamente o arquivo de configuração da aplicação, você pode colocar o array de personalização do bundle em um arquivo separado e condicionalmente incluir este arquivo na configuração da aplicação. Por exemplo,

```
return [
    'components' => [
        'assetManager' => [
            'bundles' => require __DIR__ . '/' . (YII_ENV_PROD ?
                'assets-prod.php' : 'assets-dev.php'),
        ],
    ],
];
```


Ou seja, o array de configuração do asset bundle será salvo no arquivo `assets-prod.php` quando estiver em modo de produção e o arquivo `assets-dev.php` quando não estiver em produção.

Usando o Comando

O Yii fornece um comando console chamado `asset` para automatizar a abordagem que acabamos de descrever.

Para usar este comando, você deve primeiro criar um arquivo de configuração para descrever quais asset bundles devem ser combinados e como devem ser agrupados. Você pode usar o subcomando `asset/template` para gerar um template para que possa modificá-lo para atender as suas necessidades.

```
yii asset/template assets.php
```

O comando gera um arquivo chamado `assets.php` no diretório onde foi executado. O conteúdo deste arquivo assemelha-se ao seguinte:

```
<?php
/**
 * Arquivo de configuração para o comando console "yii asset".
 * Observer que no ambiente de console, alguns caminhos de alias como
 * '@webroot' e o '@web' podem não existir.
 * Por favor, defina os caminhos de alias inexistentes.
 */
return [
    // Ajuste do comando/call-back para a compressão os arquivos
    // JavaScript:
    'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_file
    {to}',
    // Ajuste de comando/callback para a compressão dos arquivos CSS:
    'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o
    {to}',
    // A lista de asset bundles que serão comprimidos:
    'bundles' => [
        // 'yii\web\YiiAsset',
        // 'yii\web\jQueryAsset',
    ],
    // Asset bundle do resultado da compressão:
    'targets' => [
        'all' => [
            'class' => 'yii\web\AssetBundle',
            'basePath' => '@webroot/assets',
            'baseUrl' => '@web/assets',
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
        ],
    ],
    // Configuração do gerenciados de asset:
    'assetManager' => [
```

```
    ],  
  ],  
];
```

Você deve modificar este arquivo e especificar quais bundles você deseja combinar na opção `bundles`. Na opção `targets` você deve especificar como os bundles devem ser divididos em grupos. Você pode especificar um ou vários grupos, como mencionado anteriormente.

Observação: Como as alias `@webroot` e `@web` não estão disponíveis na aplicação console, você deve defini-los explicitamente na configuração.

Os arquivos JavaScript são combinados, comprimidos e escritos no arquivo `js/all-{hash}.js` onde `{hash}` será substituído pelo hash do arquivo resultante.

As opções `jsCompressor` e `cssCompressor` especificam os comando ou callbacks PHP para realizar a combinação/compressão do JavaScript e do CSS. Por padrão, o Yii usa o Closure Compiler³⁵ para combinar os arquivos JavaScript e o YUI Compressor³⁶ para combinar os arquivos CSS. Você deve instalar estas ferramentas manualmente ou ajustar estas opções para usar as suas ferramentas favoritas.

Com o arquivo de configuração, você pode executar o comando `asset` para combinar e comprimir os arquivos de asset e em seguida gerar um novo arquivo de configuração de asset bundle `assets-prod.php`:

```
yii asset assets.php config/assets-prod.php
```

O arquivo de configuração gerado pode ser incluído na configuração da aplicação, conforme descrito na última subseção.

Informação: O uso do comando `asset` não é a única opção para automatizar o processo de combinação e compressão de asset. Você pode usar a excelente ferramenta chamada `grunt`³⁷ para atingir o mesmo objetivo.

Agrupando Asset Bundles

Na última subseção, nós explicamos como combinar todos os asset bundles em um único bundle, a fim de minimizar as requisições HTTP de arquivos de asset referenciados em uma aplicação. Porém, isto nem sempre é desejável na prática. Por exemplo, imagine que sua aplicação possua um “front end”, bem como um “back end”, cada um possuindo um conjunto diferente de

³⁵<https://developers.google.com/closure/compiler/>

³⁶<https://github.com/yui/yuicompressor/>

³⁷<https://gruntjs.com/>

JavaScript e CSS. Neste caso, combinando todos os asset bundles de ambas as extremidades em um único bundle não faz sentido, porque os asset bundles para o “front end” não são utilizados pelo “back end” e seria um desperdício de uso de banda de rede para enviar os assets do “back end” quando uma página de “front end” for solicitada.

Para resolver este problema, você pode dividir asset bundles e, grupos e combinar asset bundles para cada grupo. A configuração a seguir mostra como pode agrupar os asset bundles:

```
return [
    ...
    // Especifique o bundle de saída com os grupos:
    'targets' => [
        'allShared' => [
            'js' => 'js/all-shared-{hash}.js',
            'css' => 'css/all-shared-{hash}.css',
            'depends' => [
                // Inclua todos os asset que serão compartilhados entre o
                // 'backend' e o 'frontend'
                'yii\web\YiiAsset',
                'app\assets\SharedAsset',
            ],
        ],
        'allBackEnd' => [
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
            'depends' => [
                // Inclua apenas os assets do 'backend':
                'app\assets\AdminAsset'
            ],
        ],
        'allFrontEnd' => [
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
            'depends' => [], // Inclua todos os asset restantes
        ],
    ],
    ...
];
```

Como você pode ver, os asset bundles são divididos em três grupos: `allShared`, `allBackEnd` e `allFrontEnd`. Cada um deles depende de um conjunto de asset bundles. Por exemplo, o `allBackEnd` depende de `app\assets\AdminAsset`. Ao executar o comando `asset` com essa configuração, será combinado os asset bundles de acordo com as especificações acima.

Informação: Você pode deixar a configuração `depends` em branco para um determinado bundle. Ao fazer isso, esse asset bundle dependerá de todos os asset bundles restantes que outros determinados bundles não dependam.

3.12 Extensões

As extensões são pacotes de software redistribuíveis especialmente projetadas para serem usadas em aplicações Yii e fornecem recursos prontos para o uso. Por exemplo, a extensão `yiisoft/yii2-debug` adiciona uma barra de ferramentas de depuração na parte inferior de todas as páginas em sua aplicação para ajudar a compreender mais facilmente como as páginas são geradas. Você pode usar as extensões para acelerar o processo de desenvolvimento. Você também pode empacotar seus códigos como extensões para compartilhar com outras pessoas o seu bom trabalho.

Informação: Usamos o termo “extensão” para referenciar os pacotes de software específicos do Yii. Para propósito geral, os pacotes de software que podem ser usados sem o Yii, referenciamos sob o termo de “pacote” ou “biblioteca”.

3.12.1 Usando Extensões

Para usar uma extensão, você precisa instalá-lo primeiro. A maioria das extensões são distribuídas como pacotes do Composer³⁸ que podem ser instaladas seguindo dois passos:

1. modifique o arquivo `composer.json` de sua aplicação e especifique quais extensões (pacotes do Composer) você deseja instalar.
2. execute `composer install` para instalar as extensões especificadas.

Observe que você pode precisa instalar o Composer³⁹ caso você não tenha feito isto antes.

Por padrão, o Composer instala pacotes registrados no Packagist⁴⁰ - o maior repositório open source de pacotes do Composer. Você também pode criar o seu próprio repositório⁴¹ e configurar o Composer para usá-lo. Isto é útil caso você desenvolva extensões privadas que você deseja compartilhar apenas em seus projetos.

As extensões instaladas pelo Composer são armazenadas no diretório `BasePath/vendor`, onde o `BasePath` refere-se ao `caminho base` da aplicação. Como o Composer é um gerenciador de dependências, quando ele instala um pacote, também instala todos os pacotes dependentes.

Por exemplo, para instalar a extensão `yiisoft/yii2-imagine`, modifique seu `composer.json` conforme o seguinte exemplo:

³⁸<https://getcomposer.org/>

³⁹<https://getcomposer.org/>

⁴⁰<https://packagist.org/>

⁴¹<https://getcomposer.org/doc/05-repositories.md#repository>

```
{
    // ...

    "require": {
        // ... other dependencies

        "yiisoft/yii2-imagine": "~2.0.0"
    }
}
```

Depois da instalação, você deve enxergar o diretório `yiisoft/yii2-imagine` sob o diretório `BasePath/vendor`. Você também deve enxergar outro diretório `imagine/imagine` que contém os pacotes dependentes instalados.

Informação: O `yiisoft/yii2-imagine` é uma extensão nativa desenvolvida e mantida pela equipe de desenvolvimento do Yii. Todas as extensões nativas estão hospedadas no Packagist⁴² e são nomeadas como `yiisoft/yii2-xyz`, onde `xyz` varia para cada extensão.

Agora, você pode usar as extensões instaladas como parte de sua aplicação. O exemplo a seguir mostra como você pode usar a classe `yii\image\Image` fornecido pela extensão `yiisoft/yii2-imagine`:

```
use Yii;
use yii\image\Image;

// gera uma imagem thumbnail
Image::thumbnail('@webroot/img/test-image.jpg', 120, 120)
->save(Yii::getAlias('@runtime/thumb-test-image.jpg'), ['quality' =>
50]);
```

Informação: As classes de extensão são carregadas automaticamente pela classe autoloader do Yii.

Instalando Extensões Manualmente

Em algumas raras ocasiões, você pode querer instalar algumas ou todas extensões manualmente, ao invés de depender do Composer. Para fazer isto, você deve:

1. fazer o download da extensão com os arquivos zipados e os dezipar no diretório `vendor`
2. instalar as classes autoloaders fornecidas pela extensão, se houver.
3. fazer o download e instalar todas as extensões dependentes que foi instruído.

⁴²<https://packagist.org/>

Se uma extensão não tiver uma classe autoloader seguindo a norma PSR-4⁴³, você pode usar a classe autoloader fornecida pelo Yii para carregar automaticamente as classes de extensão. Tudo o que você precisa fazer é declarar uma *alias root* para o diretório root da extensão. Por exemplo, assumindo que você instalou uma extensão no diretório `vendor/mycompany/myext` e que a classe da extensão está sob o namespace `myext`, você pode incluir o código a seguir na configuração de sua aplicação:

```
[
    'aliases' => [
        'myext' => '@vendor/mycompany/myext',
    ],
]
```

3.12.2 Criando Extensões

Você pode considerar criar uma extensão quando você sentir a necessidade de compartilhar o seu bom código para outras pessoas. Uma extensão pode conter qualquer código que você deseje, tais como uma classe helper, um widget, um módulo, etc.

É recomendado que você crie uma extensão através do pacote do Composer⁴⁴ de modo que possa ser mais facilmente instalado e usado por outros usuário, como descrito na última subseção.

Abaixo estão as básicas etapas que você pode seguir para criar uma extensão como um pacote do Composer.

1. Crie uma projeto para sua extensão e guarde-o em um repositório CVS, como o github.com⁴⁵. O trabalho de desenvolvimento e de manutenção deve ser feito neste repositório.
2. Sob o diretório root do projeto, crie um arquivo chamado `composer.json` como o requerido pelo Composer. Por favor, consulte a próxima subseção para mais detalhes.
3. Registre sua extensão no repositório do Composer, como o Packagist⁴⁶, de modo que outros usuário possam achar e instalar suas extensões usando o Composer.

Cada pacote do Composer deve ter um arquivo `composer.json` no diretório root. O arquivo contém os metadados a respeito do pacote. Você pode

⁴³<https://www.php-fig.org/psr/psr-4/>

⁴⁴<https://getcomposer.org/>

⁴⁵<https://github.com>

⁴⁶<https://packagist.org/>

achar a especificação completa sobre este arquivo no Manual do Composer⁴⁷. O exemplo a seguir mostra o arquivo `composer.json` para a extensão `yiisoft/yii2-imagine`:

```
{
    // nome do pacote
    "name": "yiisoft/yii2-imagine",

    // tipo de pacote
    "type": "yii2-extension",

    "description": "The Imagine integration for the Yii framework",
    "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"
        }
    ],

    // dependências do pacote
    "require": {
        "yiisoft/yii2": "~2.0.0",
        "imagine/imagine": "v0.5.0"
    },

    // especifica as classes autoloading
    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

Nome do Pacote Cada pacote do Composer deve ter um nome que identifica unicamente o pacote entre todos os outros. Os nomes dos pacotes devem seguir o formato `vendorName/projectName`. Por exemplo, no nome do pacote `yiisoft/yii2-imagine`, o nome do vendor e o nome do projeto são `yiisoft` e `yii2-imagine`, respectivamente.

NÃO utilize `yiisoft` como nome do seu vendor já que ele é usado pelo Yii para os códigos nativos.

⁴⁷<https://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup>

Recomendamos que você use o prefixo `yii2-` para o nome do projeto dos pacotes de extensões em Yii 2, por exemplo, `myname/yii2-mywidget`. Isto permitirá que os usuários encontrem mais facilmente uma extensão Yii 2.

Tipo de Pacote É importante que você especifique o tipo de pacote de sua extensão como `yii2-extension`, de modo que o pacote possa ser reconhecido como uma extensão do Yii quando for instalado.

Quando um usuário executar `composer install` para instalar uma extensão, o arquivo `vendor/yiisoft/extensions.php` será atualizada automaticamente para incluir informações referentes a nova extensão. A partir deste arquivo, as aplicações Yii podem saber quais extensões estão instaladas (a informação pode ser acessada através da propriedade `yii\base\Application::$extensions`).

Dependências Sua extensão depende do Yii (claro!). Sendo assim, você deve listar (`yiisoft/yii2`) na entrada `require` do `composer.json`. Se sua extensão também depender de outras extensões ou de bibliotecas de terceiros, você deve lista-los também. Certifique-se de listar as constantes de versões apropriadas (por exemplo, `1.*`, `@stable`) para cada pacote dependente. Utilize dependências estáveis quando sua extensão estiver em uma versão estável.

A maioria dos pacotes JavaScript/CSS são gerenciados pelo Bower⁴⁸ e/ou pelo NPM⁴⁹, ao invés do Composer. O Yii usa o plugin de asset do Composer⁵⁰ para habilitar a gerência destes tipos de pacotes através do Composer. Se sua extensão depender do pacote do Bower, você pode simplesmente listar a dependência no `composer.json` conforme o exemplo a seguir:

```
{
  // package dependencies
  "require": {
    "bower-asset/jquery": ">=1.11.*"
  }
}
```

O código anterior indica que a extensão depende do pacote `jquery` do Bower. Em geral, no `composer.json`, você pode usar o `bower-asset/PackageName` para referenciar um pacote do Bower no `composer.json`, e usar o `npm-asset/PackageName` para referenciar um pacote do NPM, por padrão o conteúdo do pacote será instalado sob os diretórios `@vendor/bower/PackageName` e `@vendor/npm/Packages`, respectivamente. Estes dois diretórios podem ser referenciados para usar alias mais curtas como `@bower/PackageName` e `@npm/PackageName`.

Para mais detalhes sobre o gerenciamento de asset, por favor, consulte a seção [Assets](#).

⁴⁸<https://bower.io/>

⁴⁹<https://www.npmjs.com/>

⁵⁰<https://github.com/fxpiao/composer-asset-plugin>

Classe Autoloading Para que suas classes sejam carregadas automaticamente pela classe autoloader do Yii ou da classe autoloader do Composer, você deve especificar a entrada `autoload` no arquivo `composer.json`, conforme mostrado a seguir:

```
{
    // ...

    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

Você pode listar um ou vários namespaces e seus caminhos de arquivos correspondentes.

Quando a extensão estiver instalada em uma aplicação, o Yii irá criar para cada namespace listada uma `alias` que se referenciará ao diretório correspondente ao namespace. Por exemplo, a declaração acima do `autoload` corresponderá a uma alias chamada `@yii/imagine`.

Práticas Recomendadas

Como as extensões são destinadas a serem usadas por outras pessoas, você precisará, por muitas vezes, fazer um esforço extra durante o desenvolvimento. A seguir, apresentaremos algumas práticas comuns e recomendadas na criação de extensões de alta qualidade.

Namespaces Para evitar conflitos de nomes e criar classes autocarregáveis em sua extensão, você deve usar namespaces e nomear as classes seguindo o padrão PSR-4⁵¹ ou o padrão PSR-0⁵².

Seus namespaces de classes devem iniciar com `vendorName\extensionName`, onde a `extensionName` é semelhante ao nome da extensão, exceto que ele não deve conter o prefixo `yii2-`. Por exemplo, para a extensão `yiisoft/yii2-imagine`, usamos o `yii\imagine` como namespace para suas classes.

Não use `yii`, `yii2` ou `yiisoft` como nome do seu vendor. Estes nomes são reservados para serem usados para o código nativo do Yii.

Inicialização das Classes As vezes, você pode querer que sua extensão execute algum código durante o `processo de inicialização` de uma aplicação. Por exemplo, a sua extensão pode querer responder ao evento `beginRequest` da aplicação para ajustar alguma configuração do ambiente. Embora você

⁵¹<https://www.php-fig.org/psr/psr-4/>

⁵²<https://www.php-fig.org/psr/psr-0/>

possa instruir os usuários que usam a extensão para associar explicitamente a sua função ao evento `beginRequest`, a melhor maneira é fazer isso automaticamente.

Para atingir este objetivo, você pode criar uma *classe de inicialização* implementando o `yii\base\BootstrapInterface`. Por exemplo,

```
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 () {
            // fazer alguma coisa aqui
        });
    }
}
```

Em seguida, liste esta classe no arquivo `composer.json` de sua extensão conforme o seguinte,

```
{
    // ...

    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

Quando a extensão for instalada em uma aplicação, o Yii instanciará automaticamente a classe de inicialização e chamará o método `bootstrap()` durante o processo de inicialização para cada requisição.

Trabalhando com Banco de Dados Sua extensão pode precisar acessar banco de dados. Não pressuponha que as aplicações que usam sua extensão SEMPRE usam o `Yii::$db` como a conexão do banco de dados. Em vez disso, você deve declarar a propriedade `db` para as classes que necessitam acessar o banco de dados. A propriedade permitirá que os usuários de sua extensão personalizem quaisquer conexão de banco de dados que gostariam de usar. Como exemplo, você pode consultar a classe `yii\caching\DbCache` e ver como declara e usa a propriedade `db`.

Se sua extensão precisar criar uma tabela específica no banco de dados ou fazer alterações no esquema do banco de dados, você deve:

- fornecer [migrations](#) para manipular o esquema do banco de dados, ao invés de usar arquivos simples de SQL;

- tentar criar migrations aplicáveis em diferentes SGDB;
- evitar o uso de [Active Record](#) nas migrations.

Usando Assets Se sua extensão usar um widget ou um módulo, pode ter grandes chances de requerer algum [assets](#) para funcionar. Por exemplo, um módulo pode exibir algumas páginas que contém imagens, JavaScript e CSS. Como os arquivos de uma extensão estão todos sob o diretório que não é acessível pela Web quando instalado em uma aplicação, você tem duas escolhas para tornar estes arquivos de asset diretamente acessíveis pela Web:

- informe aos usuários da extensão copiar manualmente os arquivos de asset para uma pasta determinada acessível pela Web;
- declare um [asset bundle](#) e conte com o mecanismo de publicação de asset para copiar automaticamente os arquivos listados no asset bundle para uma pasta acessível pela Web.

Recomendamos que você use a segunda abordagem de modo que sua extensão possa ser usada com mais facilidade pelos usuários. Por favor, consulte a seção [Assets](#) para mais detalhes sobre como trabalhar com assets em geral.

Internacionalização e Localização Sua extensão pode ser usada por aplicações que suportam diferentes idiomas! Portanto, se sua extensão exibir conteúdo para os usuários finais, você deve tentar usar [internacionalização e localização](#). Em particular,

- Se a extensão exibir mensagens aos usuários finais, as mensagens devem usadas por meio do método `Yii::t()` de modo que eles possam ser traduzidas. As mensagens voltadas para os desenvolvedores (como mensagens internas de exceções) não precisam ser traduzidas.
- Se a extensão exibir números, datas, etc., devem ser formatadas usando a classe `yii\i18n\Formatter` com as regras de formatação apropriadas.

Para mais detalhes, por favor, consulte a seção [Internacionalização](#).

Testes Você quer que sua extensão execute com perfeição sem trazer problemas para outras pessoas. Para alcançar este objetivo, você deve testar sua extensão antes de liberá-lo ao público.

É recomendado que você crie várias unidades de testes para realizar simulações no código de sua extensão ao invés de depender de testes manuais. Toda vez que liberar uma nova versão de sua extensão, você pode simplesmente rodar as unidades de teste para garantir que tudo esteja em boas condições. O Yii fornece suporte para testes, que podem ajuda-los a escrever mais facilmente testes unitários, testes de aceitação e testes funcionais. Para mais detalhes, por favor, consulte a seção [Testing](#).

Versionamento Você deve dar para cada liberação de sua extensão um número de versão (por exemplo, 1.0.1). Recomendamos que você siga a prática versionamento semântico⁵³ ao determinar qual número de versão será usado.

Liberando Versões Para que outras pessoas saibam sobre sua extensão, você deve liberá-lo ao público.

Se é a primeira vez que você está liberando uma extensão, você deve registrá-lo no repositório do Composer, como o Packagist⁵⁴. Depois disso, tudo o que você precisa fazer é simplesmente criar uma tag de liberação (por exemplo, v1.0.1) no repositório CVS de sua extensão e notificar o repositório do Composer sobre a nova liberação. As pessoas, então, serão capazes de encontrar a nova versão e instalá-lo ou atualizá-lo através do repositório do Composer.

As versões de sua extensão, além dos arquivos de códigos, você deve também considerar a inclusão de roteiros para ajudar as outras pessoas aprenderem a usar a sua extensão:

- Um arquivo `readme` no diretório root do pacote: descreve o que sua extensão faz e como faz para instalá-lo e usá-lo. Recomendamos que você escreva no formato Markdown⁵⁵ e o nome do arquivo como `readme.md`.
- Um arquivo `changelog` no diretório root do pacote: lista quais mudanças foram feitas em cada versão. O arquivo pode ser escrito no formato Markdown e nomeado como `changelog.md`.
- Um arquivo de atualização no diretório root do pacote: fornece as instruções de como atualizar a extensão a partir de versões antigas. O arquivo deve ser escrito no formato Markdown e nomeado como `upgrade.md`.
- Tutoriais, demos, screenshots, etc.: estes são necessários se sua extensão fornece muitos recursos que podem não ser totalmente cobertos no arquivo `readme`.
- Documentação da API: seu código deve ser bem documentado para permitir que outros usuários possam ler e entender mais facilmente. Você pode consultar o arquivo da classe `BaseObject`⁵⁶ para aprender como documentar o seu código.

Informação: Os seus comentários no código podem ser escritos no formato Markdown. A extensão `yiisoft/yii2-apidoc` fornece uma ferramenta para gerar uma documentação da API com base nos seus comentários.

⁵³<https://semver.org>

⁵⁴<https://packagist.org/>

⁵⁵<https://daringfireball.net/projects/markdown/>

⁵⁶<https://github.com/yiisoft/yii2/blob/master/framework/base/BaseObject.php>

Informação: Embora não seja um requisito, sugerimos que sua extensão se conforme a determinados estilos de codificação. Você pode consultar o estilo de codificação do framework⁵⁷.

3.12.3 Extensões Nativas

O Yii fornece as seguintes extensões que são desenvolvidas e mantidas pela equipe de desenvolvimento do Yii. Todos são registrados no Packagist⁵⁸ e podem ser facilmente instalados como descrito na subseção Usando Extensões.

- yiisoft/yii2-apidoc⁵⁹: fornece um gerador de API de documentação extensível e de alto desempenho. Também é usado para gerar a API de documentação do framework.
- yiisoft/yii2-authclient⁶⁰: fornece um conjunto comum de autenticadores de clientes, como Facebook OAuth2 client, GitHub OAuth2 client.
- yiisoft/yii2-bootstrap⁶¹: fornece um conjunto de widgets que encapsulam os componentes e plug-ins do Bootstrap⁶².
- yiisoft/yii2-debug⁶³: fornece suporte a depuração para aplicações Yii. Quando esta extensão é usada, uma barra de ferramenta de depuração aparecerá na parte inferior de cada página. A extensão também fornece um conjunto de páginas independentes para exibir mais detalhes das informações de depuração.
- yiisoft/yii2-elasticsearch⁶⁴: fornece suporte para o uso de Elasticsearch⁶⁵. Este inclui suporte a consultas/pesquisas básicas e também implementa o padrão **Active Record** que permite que você armazene os active records no Elasticsearch.
- yiisoft/yii2-faker⁶⁶: fornece suporte para o uso de Faker⁶⁷ para gerar dados falsos para você.
- yiisoft/yii2-gii⁶⁸: fornece um gerador de código baseado na Web que é altamente extensível e pode ser usado para gerar rapidamente models (modelos), formulários, módulos, CRUD, etc.
- yiisoft/yii2-httpclient⁶⁹: provides an HTTP client.

⁵⁷<https://github.com/yiisoft/yii2/blob/master/docs/internals/core-code-style.md>

⁵⁸<https://packagist.org/>

⁵⁹<https://github.com/yiisoft/yii2-apidoc>

⁶⁰<https://github.com/yiisoft/yii2-authclient>

⁶¹<https://github.com/yiisoft/yii2-bootstrap>

⁶²<https://getbootstrap.com/>

⁶³<https://github.com/yiisoft/yii2-debug>

⁶⁴<https://github.com/yiisoft/yii2-elasticsearch>

⁶⁵<https://www.elastic.co/>

⁶⁶<https://github.com/yiisoft/yii2-faker>

⁶⁷<https://github.com/fzaninotto/Faker>

⁶⁸<https://github.com/yiisoft/yii2-gii>

⁶⁹<https://github.com/yiisoft/yii2-httpclient>

- yiisoft/yii2-imagine⁷⁰: fornece funções de manipulação de imagens comumente utilizados com base no Imagine⁷¹.
- yiisoft/yii2-jui⁷²: fornece um conjunto de widgets que encapsulam as interações e widgets do JQuery UI⁷³.
- yiisoft/yii2-mongodb⁷⁴: fornece suporte para o uso do MongoDB⁷⁵. Este inclui recursos como consultas básicas, Active Record, migrations, cache, geração de códigos, etc.
- yiisoft/yii2-redis⁷⁶: fornece suporte para o uso do redis⁷⁷. Este inclui recursos como consultas básicas, Active Record, cache, etc.
- yiisoft/yii2-smarty⁷⁸: fornece um motor de template baseado no Smarty⁷⁹.
- yiisoft/yii2-sphinx⁸⁰: fornece suporte para o uso do Sphinx⁸¹. Este inclui recursos como consultas básicas, Active Record, geração de códigos, etc.
- yiisoft/yii2-swiftmailer⁸²: fornece recursos para envio de e-mails baseados no swiftmailer⁸³.
- yiisoft/yii2-twig⁸⁴: fornece um motor de template baseado no Twig⁸⁵.

⁷⁰<https://github.com/yiisoft/yii2-imagine>

⁷¹<https://imagine.readthedocs.org/>

⁷²<https://github.com/yiisoft/yii2-jui>

⁷³<https://jqueryui.com/>

⁷⁴<https://github.com/yiisoft/yii2-mongodb>

⁷⁵<https://www.mongodb.com/>

⁷⁶<https://github.com/yiisoft/yii2-redis>

⁷⁷<https://redis.io/>

⁷⁸<https://github.com/yiisoft/yii2-smarty>

⁷⁹<https://www.smarty.net/>

⁸⁰<https://github.com/yiisoft/yii2-sphinx>

⁸¹<https://sphinxsearch.com>

⁸²<https://github.com/yiisoft/yii2-swiftmailer>

⁸³<https://swiftmailer.org/>

⁸⁴<https://github.com/yiisoft/yii2-twig>

⁸⁵<https://twig.symfony.com/>

Capítulo 4

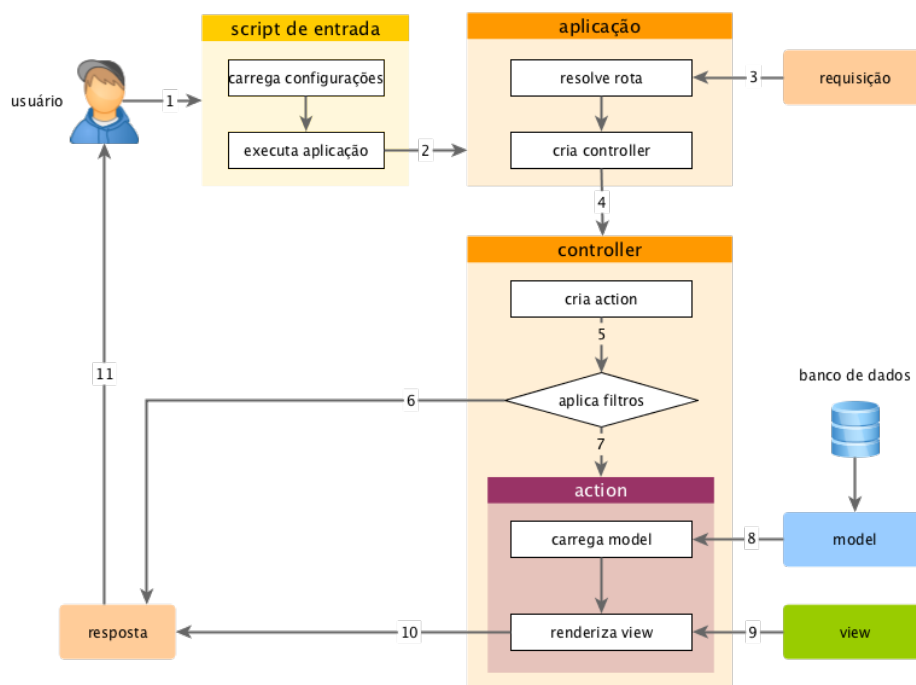
Tratando Requisições

4.1 Visão Geral

Cada vez que uma aplicação Yii processa uma requisição, ele passa por um fluxo de trabalho parecido como o seguinte:

1. Um usuário faz uma pedido para o [script de entrada](#) `web/index.php`.
2. O script de entrada carrega a [configuração](#) da aplicação e cria uma instância da [aplicação](#) para processar o pedido.
3. A aplicação resolve a [rota](#) solicitada com a ajuda do componente [request](#) da aplicação.
4. A aplicação cria uma instância do [controller \(controlador\)](#) para processar o pedido.
5. O controller (controlador) cria uma instância da [ação](#) e executar os filtros para a ação.
6. Se qualquer filtro falhar, a ação será cancelada.
7. Se todos os filtros passarem, a ação será executada.
8. A ação carrega os dados do model (modelo), possivelmente a partir de um banco de dados.
9. A ação renderiza uma view (visão), com os dados fornecidos pelo model (modelo).
10. O resultado da renderização é devolvida para o componente [response](#) da aplicação.
11. O componente response envia o resultado da renderização para o navegador do usuário.

O diagrama a seguir mostra como uma aplicação processa um pedido.



Nesta seção, descreveremos com mais detalhes como alguns destes passos trabalham.

4.2 Inicialização (Bootstrapping)

A inicialização refere-se ao processo de preparação do ambiente antes que uma aplicação comece a resolver e processar um pedido de requisição. A inicialização é feita em duas etapas: O **script de entrada** e a **aplicação**.

No **script de entrada**, a classe de autoloaders de diferentes bibliotecas são registradas. Inclui o autoloader do Composer através do seu arquivo `autoload.php` e o autoloader do Yii através do seu arquivo `yii`. O script de entrada, em seguida, carrega a **configuração** da aplicação e cria uma instância da **aplicação**.

No construtor da aplicação, as seguintes etapas de inicialização serão realizadas:

1. O método `preInit()` é chamado, na qual algumas propriedades da aplicação de alta prioridade serão configuradas, como o `basePath`.
2. Registra o **manipulador de erro**.
3. Inicializa as propriedades da aplicação a partir da configuração da aplicação.

4. O método `init()` é chamado, que por sua vez chamará o método `bootstrap()` para executar os componentes de inicialização.
 - Inclui o arquivo `vendor/yiisoft/extensions.php` de manifesto da extensão.
 - Cria e executa os [componentes de inicialização](#) declaradas pelas extensões.
 - Cria e executa os [componentes da aplicação](#) e/ou os [módulos](#) declarados na [propriedade bootstrap](#) da aplicação.

Como as etapas de inicialização tem que ser feitos antes da manipulação de *cada* requisição, é muito importante que mantenha este processo limpo e otimizado o máximo possível.

Tente não registrar muitos componentes de inicialização. Um componente de inicialização é necessário apenas se quiser participar de todo o ciclo de vida do processo da requisição. Por exemplo, se um módulo precisar registrar uma análise de regras de URL adicionais, deve ser listados na [propriedade bootstrap](#) de modo que as novas regras de URL possam ter efeito antes que sejam usados para resolver as requisições.

No modo de produção, habilite um cache de bytecode, como o PHP OPcache¹ ou APC², para minimizar o tempo necessário para a inclusão e análise os arquivos PHP.

Algumas aplicações de larga escala possuem [configurações](#) complexas, que são divididos em vários arquivos menores. Se este for o caso, considere guardar o cache de todo o array da configuração e carregue-o diretamente a partir deste cache antes da criação da instância da aplicação no script de entrada.

4.3 Roteamento e Criação de URL

Quando uma aplicação Yii começa a processar uma URL requerida, o primeiro passo necessário é obter a rota pela análise da URL. A rota é usada para instanciar o [controlador \(controller\) da ação](#) correspondente para manipular a requisição. Todo este processo é chamado de *roteamento*.

O processo inverso do roteamento é chamada de *criação de URL*, onde é criado uma URL a partir de uma determinada rota e seus parâmetros. Quando a URL criada for exigida em outro momento, o processo de roteamento pode resolve-la de volta para a rota original e seus parâmetros.

O ponto central responsável pelo roteamento e pela criação de URL é o [gerenciador de URL](#), na qual é registrado como o [componente da aplicação](#) `urlManager`. O [gerenciador de URL](#) fornece o método `parseRequest()` para analisar um requisição de entrada a fim de obter uma rota e seus parâmetros

¹https://www.php.net/manual/pt_BR/intro.opcache.php

²https://www.php.net/manual/pt_BR/book.apcu.php

associados; e o método `createUrl()` para criar uma URL a partir de uma rota e seus parâmetros associados.

Ao configurar o componente `urlManager` na configuração da aplicação, poderá fazer com que sua aplicação reconheça de forma arbitrária diversos formatos de URL sem modificar o código existente da aplicação. Por exemplo, você pode usar o código a seguir para criar uma URL a partir da ação `post/view`:

```
use yii\helpers\Url;

// Url::to() chama UrlManager::createUrl() para criar uma URL
$url = Url::to(['post/view', 'id' => 100]);
```

Dependendo da configuração da propriedade `urlManager`, a URL criada pode ser parecida com um dos formatos a seguir (ou até mesmo com outro formato). E se a URL criada for requerida, ainda será analisada a fim de obter a rota e os valores dos parâmetros originais.

```
/index.php?r=post/view&id=100
/index.php/post/100
/posts/100
```

4.3.1 Formatos de URL

O gerenciador de URL suporta dois formatos de URL: o formato de URL padrão e o formato de URL amigável (pretty URL).

O formato de URL padrão usa um parâmetro chamado `r` para representar a rota e os demais parâmetros representam os parâmetros associados a rota. Por exemplo, a URL `/index.php?r=post/view&id=100` representa a rota `post/view` e o parâmetro `id` com o valor 100. O formato de URL padrão não exige qualquer tipo de configuração no gerenciador de URL e trabalha em qualquer servidor Web.

O formato de URL amigável (pretty URL) usa um caminho adicional após o nome do script de entrada para representar a rota e seus parâmetros. Por exemplo, o caminho adicional na URL `/index.php/post/100` é `/post/100`, onde pode representar, em uma adequada regra de URL, a rota `post/view` e o parâmetro `id` com o valor 100. Para usar o formato de URL amigável (pretty URL), você precisará escrever um conjunto de regras de URLs de acordo com a necessidade sobre como as URLs devem parecer.

Você pode alterar entre os dois formatos de URLs, alternando a propriedade `enablePrettyUrl` do gerenciador de URL sem alterar qualquer código na aplicação.

4.3.2 Roteamento

O roteamento envolve duas etapas. Na primeira etapa, a requisição de entrada é transformada em uma rota e seus parâmetros. Na segunda etapa, a

ação do controller (controlador) correspondente a rota analisada será criada para processar a requisição.

Ao utilizar o formato de URL padrão, a análise de uma requisição para obter uma rota é tão simples como pegar o valor via `GET` do parâmetro `r`.

Ao utilizar o formato de URL amigável (pretty URL), o gerenciado de URL examinará as regras de URLs registradas para encontrar alguma correspondência que poderá resolver a requisição em uma rota. Se tal regra não for encontrada, uma exceção `yii\web\NotFoundHttpException` será lançada.

Uma vez que a requisição analisada apresentar uma rota, é hora de criar a ação do controller (controlador) identificado pela rota. A rota é dividida em várias partes pelo separador barra (“/”). Por exemplo, a rota `site/index` será dividida em duas partes: `site` e `index`. Cada parte é um ID que pode referenciar a um módulo, um controller (controlador) ou uma ação. A partir da primeira parte da rota, a aplicação executará as seguintes etapas para criar o módulo (se existir), o controller (controlador) e a ação:

1. Define a aplicação como o módulo atual.
2. Verifica se o mapa do controller (controlador) do módulo contém o ID atual. Caso exista, um objeto do controller (controlador) será criado de acordo com a configuração do controller (controlador) encontrado no mapa e a etapa 3 e 4 não serão executadas.
3. Verifica se o ID referencia a um módulo listado na propriedade `modules` do módulo atual. Caso exista, um módulo será criado de acordo com as configurações encontradas na lista e a etapa 2 será executada como etapa seguinte do processo, no âmbito de usar o contexto do módulo recém-criado.
4. Trata o ID como um ID do controller (controlador) e cria um objeto do controller (controlador). Siga para a próxima etapa, como parte restante do processo.
5. O controller (controlador) procura o ID atual em seu mapa de ações. Caso exista, será criada uma ação de acordo com a configuração encontrada no mapa. Caso contrário, o controller (controlador) tentará criar uma ação inline que é definida por um método da ação correspondente ao ID atual.

Nas etapas acima, se ocorrer qualquer erro, uma exceção `yii\web\NotFoundHttpException` será lançada, indicando a falha no processo de roteamento.

Rota Padrão

Quando uma requisição analisada apresentar uma rota vazia, a assim chamada *rota padrão* será usada em seu lugar. A rota padrão é `site/index`

é utilizada como padrão, que referencia a ação `index` do controller (controlador) `site`. Você pode personalizar esta configuração pela propriedade `defaultRoute` na configuração da aplicação como mostrado a seguir:

```
[
    // ...
    'defaultRoute' => 'main/index',
];
```

Rota

Às vezes você pode querer colocar sua aplicação Web em modo de manutenção temporariamente e exibir uma mesma página com informações para todas as requisições. Existem muitas maneiras de atingir este objetivo. Mas uma das maneiras mais simples é apenas configurar a propriedade `yii\web\Application::$catchAll` na configuração da aplicação como mostrado a seguir:

```
[
    // ...
    'catchAll' => ['site/offline'],
];
```

Na configuração acima, a ação `site/offline` será utilizado para lidar com todas as requisições recebidas.

A propriedade `catchAll` deve receber um array, o primeiro elemento especifica a rota e o restante dos elementos (pares de nomes e seus valores) especificam os parâmetros a serem associados a ação.

4.3.3 Criando URLs

O Yii fornece um método de ajuda `yii\helpers\Url::to()` para criar vários tipos de URLs a partir de determinadas rotas e seus parâmetros. Por exemplo,

```
use yii\helpers\Url;

// cria uma URL para uma rota: /index.php?r=post/index
echo Url::to(['post/index']);

// cria uma URL para uma rota com parâmetros: /index.php?r=post/view&id=100
echo Url::to(['post/view', 'id' => 100]);

// cria uma URL ancorada: /index.php?r=post/view&id=100#content
echo Url::to(['post/view', 'id' => 100, '#' => 'content']);

// cria uma URL absoluta: https://www.example.com/index.php?r=post/index
echo Url::to(['post/index'], true);
```

```
// cria uma URL absoluta usando https:
https://www.example.com/index.php?r=post/index
echo Url::to(['post/index'], 'https');
```

Observe que nos exemplos acima, assumimos o formato de URL padrão. Se o formato de URL amigável (pretty URL) estiver habilitado, as URLs criadas serão diferentes de acordo com as **regra de URL** em uso.

A rota passada para o método `yii\helpers\Url::to()` é sensível ao contexto. Ele pode ser tanto uma rota *relativa* quanto uma rota *absoluta* que será normalizada de acordo com as regras a seguir:

- Se a rota for uma string vazia, a requisição atual **route** será usada;
- Se a rota não contiver barras ("/"), ele será considerado um ID da ação do controller (controlador) atual e será antecedido pelo valor da propriedade **uniqueId** do controller (controlador) atual;
- Se a rota não contiver uma barra ("/") inicial, será considerado como uma rota relativa ao módulo atual e será antecedido pelo valor da propriedade **uniqueId** do módulo atual.

A partir da versão 2.0.2, você pode especificar uma rota usando **alias**. Se for este o caso, a alias será a primeira a ser convertida em uma rota real e em seguida será transformada em uma rota absoluta de acordo com as regras informadas anteriormente.

Por exemplo, assumindo que o módulo atual é **admin** e o controller (controlador) atual é **post**,

```
use yii\helpers\Url;

// rota atual requerida: /index.php?r=admin/post/index
echo Url::to(['']);

// uma rota relativa com apenas o ID da ação: /index.php?r=admin/post/index
echo Url::to(['index']);

// uma rota relativa: /index.php?r=admin/post/index
echo Url::to(['post/index']);

// uma rota absoluta: /index.php?r=post/index
echo Url::to(['/post/index']);

// /index.php?r=post/index      assumindo que a alias "@posts" foi definida
//                               como "/post/index"
echo Url::to(['@posts']);
```

O método `yii\helpers\Url::to()` é implementado através das chamadas dos métodos `createUrl()` e `createAbsoluteUrl()` do gerenciador de URL. Nas próximas subseções, iremos explicar como configurar o gerenciador de URL para personalizar os formatos das URLs criadas.

O método `yii\helpers\Url::to()` também suporta a criação de URLs NÃO relacionadas a uma rota em particular. Neste caso, ao invés de passar

um array como seu primeiro parâmetro, você pode passar uma string. Por exemplo,

```
use yii\helpers\Url;

// rota atual requerida: /index.php?r=admin/post/index
echo Url::to();

// uma alias da URL: https://example.com
Yii::setAlias('@example', 'https://example.com/');
echo Url::to('@example');

// uma URL absoluta: https://example.com/images/logo.gif
echo Url::to('/images/logo.gif', true);
```

Além do método `to()`, a classe auxiliar `yii\helpers\Url` também fornece uma série de métodos referentes a criação de URLs. Por exemplo,

```
use yii\helpers\Url;

// URL da página inicial: /index.php?r=site/index
echo Url::home();

// URL base, útil se a aplicação for implementada em uma subpasta da pasta
raiz do servidor Web
echo Url::base();

// A URL canônica da requisição atual
// Veja mais detalhes em
https://en.wikipedia.org/wiki/Canonical_link_element
echo Url::canonical();

// Obtém a URL da requisição anterior para reutilizá-la em requisições
futuras
Url::remember();
echo Url::previous();
```

4.3.4 Usando URLs Amigáveis (Pretty URLs)

Para utilizar as URLs amigáveis (pretty URLs), configure o componente `urlManager` na configuração da aplicação conforme o exemplo a seguir:

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => false,
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

A propriedade `enablePrettyUrl` é obrigatória para habilitar o formato de URL amigável (pretty URL). O restante das propriedades são opcionais. No entanto, a configuração utilizada acima é a mais utilizada.

- **showScriptName**: esta propriedade determina se o script de entrada deve ser incluído ou não nas URLs criadas. Por exemplo, ao invés de criar uma URL `/index.php/post/100`, definindo esta propriedade como `false`, a URL `/post/100` será gerada.
- **enableStrictParsing**: esta propriedade habilita uma análise rigorosa (strict parsing) da requisição. Caso a análise rigorosa estiver habilitada, a URL da requisição deve corresponder pelo menos a uma das regras definidas pela propriedade **rules** a fim de ser tratado como uma requisição válida, caso contrário a exceção `yii\web\NotFoundHttpException` será lançada. Caso a análise rigorosa estiver desabilitada, as regras definidas pela propriedade **rules** NÃO serão verificadas e as informações obtidas pela URL serão tratadas como a rota da requisição.
- **rules**: esta propriedade contém uma lista de regras especificando como serão analisadas e criadas as URLs. Esta é a principal propriedade que você deve trabalhar para criar URLs cujos formatos satisfaçam a sua exigência em particular.

Observação: A fim de esconder o nome do script de entrada nas URLs criadas, além de definir a propriedade **showScriptName** como `false`, você também pode precisar configurar o seu servidor Web para identificar corretamente o script PHP quando uma URL da requisição não especificar um explicitamente. Caso você estejam utilizando um servidor Web com Apache, você pode consultar a configuração recomendada, conforme descrito na seção [Installation](#).

Regras de URLs

Uma regra de URL é uma instância de `yii\web\UrlRule` ou de classes que as estendam. Cada regra de URL consiste de um padrão usado para combinar as partes do caminho das URLs, como uma rota e alguns parâmetros. Uma regra de URL pode ser usado para analisar uma URL da requisição somente se o padrão corresponder com a esta URL. Uma regra de URL pode ser usada para criar uma URL para corresponder a uma determinada rota e seus parâmetros.

Quando uma formato de URL amigável (pretty URL) estiver habilitada, o **gerenciador de URL** utilizará as regras de URLs declaradas na propriedade **rules** para analisar as requisições e para criar URLs. Em particular, para analisar uma requisição, o **gerenciador de URL** verificará as regras na ordem em que foram declaradas e só enxergará a *primeira* regra que corresponda a URL da requisição. A regra que foi correspondida é então utilizada

para obter a rota e seus parâmetros a partir de sua análise. A mesma coisa acontece na criação de URLs, o gerenciador de URL enxergará apenas a primeira regra que corresponda a rota e seus parâmetros para serem utilizados na criação de uma URL.

Você pode configurar a propriedade `yii\web\UrlManager::$rules` com um array composto de pares de chave-valor, onde a chave é o padrão da regra e o valor serão as rotas. Cada par de padrão-rota define uma regra de URL. Por exemplo, as regras a seguir configuram duas regras de URL. A primeira regra corresponde a uma URL chamada `posts` sendo mapeado para utilizar a rota `post/index`. A segunda regra corresponde a uma URL que combine com expressão regular `post/(\d+)` seguido de um parâmetro chamado `id` sendo mapeado para utilizar a rota `post/view`.

```
[
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

Informação: O padrão em uma regra é usado para identificar o caminho de uma URL. Por exemplo, o caminho da URL `/index.php/post/100?source=ad` é `post/100` (as barras (“/”) iniciais e finais serão ignoradas) combinando com o padrão `post/(\d+)`.

Além de declarar regras de URL como pares de padrão-rota, você também pode declarar como array. Cada array é utilizado para configurar um único objeto da regra de URL. Isto se faz necessário quando você deseja configurar outras propriedades de uma regra de URL. Por exemplo,

```
[
    // ...outras regras de URL...

    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

Por padrão, se você não especificar a opção `class` na configuração de uma regra, será utilizado a classe `yii\web\UrlRule`.

Parâmetros Nomeados

Uma regra de URL pode ser associado a alguns parâmetros nomeados que são especificados no padrão `<ParamName:RegExp>`, onde o `ParamName` especifica o nome do parâmetro e o `RegExp` especifica uma expressão regular opcional usada para corresponder ao valor do parâmetro. Se o `RegExp` não for especificado, significará que o valor do parâmetro será uma string sem barras (“/”).

Observação: Você apenas pode especificar expressões regulares para os parâmetros. As demais partes de um padrão serão considerados como um texto simples.

Quando esta regra for utilizada para analisar uma URL, os parâmetros associados serão preenchidos com os valores que foram correspondidos pela regra e estes parâmetros serão disponibilizados logo a seguir no `$_GET` pelo componente da aplicação `request`.

Vamos utilizar alguns exemplos para demonstrar como os parâmetros nomeados funcionam. Supondo que declaramos as três regras a seguir:

```
[
    'posts/<year:\d{4}>/<category>' => 'post/index',
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

Quando as regras forem utilizadas para analisar as URLs:

- `/index.php/posts` obterá a rota `post/index` usando a segunda regra;
- `/index.php/posts/2014/php` obterá a rota `post/index`, o parâmetro `year` cujo o valor é 2014 e o parâmetro `category` cujo valor é `php` usando a primeira regra;
- `/index.php/post/100` obterá a rota `post/view` e o parâmetro `id` cujo valor é 100 usando a terceira regra;
- `/index.php/posts/php` causará uma exceção `yii\web\NotFoundHttpException` quando a propriedade `yii\web\UrlManager::$enableStrictParsing` for `true`, por não ter correspondido a nenhum dos padrões. Se a propriedade `yii\web\UrlManager::$enableStrictParsing` for `false` (o valor padrão), o caminho `posts/php` será retornado como uma rota.

E quando as regras forem utilizadas para criar as URLs:

- `Url::to(['post/index'])` cria `/index.php/posts` usando a segunda regra;
- `Url::to(['post/index', 'year' => 2014, 'category' => 'php'])` cria `/index.php/posts/2014/php` usando a primeira regra;
- `Url::to(['post/view', 'id' => 100])` cria `/index.php/post/100` usando a terceira regra;
- `Url::to(['post/view', 'id' => 100, 'source' => 'ad'])` cria `/index.php/post/100?source=ad` usando a terceira regra. Pela razão do parâmetro `source` não foi especificado na regra, ele será acrescentado como uma query string na criação da URL.
- `Url::to(['post/index', 'category' => 'php'])` cria `/index.php/post/index?category=php` usando nenhuma das regras. Observe que, se nenhuma das regras forem aplicadas, a URL será criada simplesmente como a rota sendo o caminho e todos os parâmetros como query string.

Parametrizando Rotas

Você pode incorporar nomes de parâmetros na rota de uma regra de URL. Isto permite que uma regra de URL seja utilizada para combinar diversas rotas. Por exemplo, a regra a seguir incorpora os parâmetros `controller` e `action` nas rotas.

```
[
  '<controller:(post|comment)>/<id:\d+>/<action:(create|update|delete)>'
  => '<controller>/<action>',
  '<controller:(post|comment)>/<id:\d+>' => '<controller>/view',
  '<controller:(post|comment)>s' => '<controller>/index',
]
```

Para analisar uma URL `/index.php/comment/100/create`, a primeira regra será aplicada, na qual foi definida o parâmetro `controller` para ser `comment` e o parâmetro `action` para ser `create`. Sendo assim, a rota `<controller>/<action>` é resolvida como `comment/create`.

De forma similar, para criar uma URL com a rota `comment/index`, a terceira regra será aplicada, criando um URL `/index.php/comments`.

Informação: Pela parametrização de rotas, é possível reduzir significativamente o número de regras de URL, que também pode melhorar o desempenho do gerenciador de URL.

Por padrão, todos os parâmetros declarados nas regras são obrigatórios. Se uma URL da requisição não contiver um dos parâmetros em particular, ou se a URL está sendo criado sem um dos parâmetros em particular, a regra não será aplicada. Para fazer com que algum parâmetro em particular seja opcional, você pode configurar a propriedade `defaults` da regra. Os parâmetros listados nesta propriedade são opcionais e serão utilizados quando os mesmos não forem fornecidos.

A declaração da regra a seguir, ambos os parâmetros `page` e `tag` são opcionais e utilizarão o valor 1 e a string vazia, respectivamente, quando não forem fornecidos.

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

A regra anterior pode ser usado para analisar ou criar qualquer uma das seguintes URLs:

- `/index.php/posts`: `page` é 1, `tag` é ''.

- `/index.php/posts/2`: page é 2, tag is ''.
- `/index.php/posts/2/news`: page é 2, tag é 'news'.
- `/index.php/posts/news`: page é 1, tag é 'news'.

Sem o uso dos parâmetros opcionais, você deveria criar 4 regras para alcançar o mesmo resultado.

Regras com Domínios

É possível incluir domínios nos padrões das regras de URL. Isto é útil quando sua aplicação se comporta de forma diferente em diferentes domínios. Por exemplo, a regra a seguir obtém a rota `admin/user/login` pela análise da URL `https://admin.example.com/login` e a rota `site/login` pela análise da URL `https://www.example.com/login`.

```
[  
    'https://admin.example.com/login' => 'admin/user/login',  
    'https://www.example.com/login' => 'site/login',  
]
```

Você também pode incorporar parâmetros nos domínios para extrair informações dinamicamente a partir deles. Por exemplo, a regra a seguir obtém a rota `post/index` e o parâmetro `language=en` pela análise da URL `https://en.example.com/posts`

```
[  
    'http://<language:\w>.example.com/posts' => 'post/index',  
]
```

Observação: Regras com domínios NÃO devem ser incluídos com subpastas do script de entrada em seus padrões. Por exemplo, se a aplicação estiver sob `https://www.example.com/sandbox/blog`, você deve usar o padrão `https://www.example.com/posts` ao invés de `https://www.example.com/sandbox/blog/posts`. Isto permite que sua aplicação seja implantado sob qualquer diretório sem a necessidade de alterar o código da aplicação.

Sufixos da URL

Você pode querer adicionar sufixos nas URLs para diversos fins. Por exemplo, você pode adicionar o `.html` nas URLs para que se pareçam com páginas estáticas. Você também pode adicionar o `.json` nas URLs para indicar o tipo de conteúdo esperado na resposta. Você pode alcançar este objetivo configurando a propriedade `yii\web\UrlManager::$suffix` na configuração da aplicação conforme o exemplo a seguir:

```
[  
    'components' => [  

```

```

    'urlManager' => [
      'enablePrettyUrl' => true,
      'showScriptName' => false,
      'enableStrictParsing' => true,
      'suffix' => '.html',
      'rules' => [
        // ...
      ],
    ],
  ],
]

```

A configuração anterior permitirá que o **gerenciador de URL** reconheçam as URLs solicitadas e que também criem URLs com o prefixo `.html`.

Dica: Você pode definir `/` como o sufixo para que todas as URLs terminem com barra.

Observação: Ao configurar um sufixo da URL e a URL da requisição não conter um, será considerado como uma URL não válida. Isto é uma prática recomendada no SEO (otimização para mecanismos de pesquisa, do *inglês search engine optimization*).

Às vezes você pode querer utilizar diferentes sufixos para diferentes URLs. Isto pode ser alcançado pela configuração da propriedade `suffix` individualmente para cada regra de URL. Quando uma regra de URL possuir esta propriedade definida, sobrescreverá o sufixo que foi definido da camada do **gerenciador de URL**. Por exemplo, a configuração a seguir contém uma regra de URL personalizada que usa o `.json` como sufixo ao invés do sufixo `.html` definido globalmente.

```

[
  'components' => [
    'urlManager' => [
      'enablePrettyUrl' => true,
      'showScriptName' => false,
      'enableStrictParsing' => true,
      'suffix' => '.html',
      'rules' => [
        // ...
        [
          'pattern' => 'posts',
          'route' => 'post/index',
          'suffix' => '.json',
        ],
      ],
    ],
  ],
],
]

```

Métodos HTTP

Ao implementar RESTful API, é necessário que sejam obtidas rotas diferentes pela análise de uma mesma URL de acordo com o método HTTP utilizado. Isto pode ser alcançado facilmente adicionando o prefixo do método HTTP suportado, separando os nomes dos métodos por vírgulas. Por exemplo, a regra a seguir possui o mesmo padrão `post/<id:\d+>` com suporte a diferentes métodos HTTP. A análise de uma requisição `PUT post/100` obterá a rota `post/create`, enquanto a requisição `GET post/100` obterá a rota `post/view`.

```
[
    'PUT,POST post/<id:\d+>' => 'post/create',
    'DELETE post/<id:\d+>' => 'post/delete',
    'post/<id:\d+>' => 'post/view',
]
```

Observação: Se uma regra de URL contiver método(s) HTTP, esta regra só será utilizada para análises de URLs. A regra será ignorada quando o gerenciador de URL for chamado para criar URLs.

Dica: Para simplificar o roteamento do RESTful APIs, o Yii fornece uma classe especial `yii\rest\UrlRule` de regras que é muito diferente. Esta classe suporta muitos recursos como a pluralização automática de IDs do controller (controlador). Para mais detalhes, por favor, consulte a seção [Routing](#) sobre o desenvolvimento de RESTful APIs.

Regras Personalizadas

Nos exemplo anteriores, as regras de URL são declaradas principalmente no formato de pares de padrão-rota. Este é um formato de atalho bastante utilizado. Em alguns cenários, você pode querer personalizar uma regra de URL configurando outras propriedades, tais como o `yii\web\UrlRule::$suffix`. Isto pode ser feito utilizando um array de configuração para especificar uma regra. O exemplo a seguir foi retirado da subseção Sufixos da URL,

```
[
    // ...outras regras de URL...

    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

Informações: Por padrão, se você não especificar a opção `class` na configuração de uma regra, será usado como padrão a classe `yii\web\UrlRule`.

Adicionando Regras Dinamicamente

As regras de URL podem ser adicionadas dinamicamente ao gerenciador de URL. Esta técnica muitas vezes se faz necessária em módulos que são redistribuídos e que desejam gerenciar as suas próprias regras de URL. Para que estas regras sejam adicionadas dinamicamente e terem efeito durante o processo de roteamento, você pode adicioná-las durante a inicialização (*bootstrapping*). Para os módulos, significa que deve implementar a interface `yii\base\BootstrapInterface` e adicionar as regras no método `bootstrap()` conforme o exemplo a seguir:

```
public function bootstrap($app)
{
    $app->getUrlManager()->addRules([
        // declare as regras aqui
    ], false);
}
```

Observe que você também deve listar estes módulos no `yii\web\Application::bootstrap()` para que eles sejam usados no processo de inicialização (*bootstrapping*).

Criando Classes de Regras

Apesar do fato que a classe padrão `yii\web\UrlRule` é flexível o suficiente para a maior parte dos projetos, há situações em que você terá que criar a sua própria classe de regra. Por exemplo, em um site de venda de carros, você pode querer dar suporte a um formato de URL como `/Manufacturer/Model`, que tanto o `Manufacturer` quanto o `Model` devem coincidir com os dados armazenados em uma tabela do banco de dados. A classe de regra padrão não vai funcionar nesta situação pois vão se basear em padrões estaticamente declarados.

Podemos criar uma classe de regra de URL para resolver este formato.

```
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; // esta regra não se aplica
    }

    public function parseRequest($manager, $request)
    {
        $pathInfo = $request->getPathInfo();
        if (preg_match('%^(\\w+)/?(\\w+)?$%', $pathInfo, $matches)) {
            // checa o $matches[1] e $matches[3] para verificar
            // se coincidem com um *fabricante* e um *modelo* no banco de
            // dados.
            // Caso coincida, define o $params['manufacturer'] e/ou
            $params['model']
            // e retorna ['car/index', $params]
        }
        return false; // esta regra não se aplica
    }
}

```

E utilize esta nova classe de regra na configuração `yii\web\UrlManager::`
`$rules`:

```

[
    // ...outras regras...

    [
        'class' => 'app\components\CarUrlRule',
        // ...configurar outras propriedades...
    ],
]

```

4.3.5 Considerando Performance

Ao desenvolver uma aplicação Web complexa, é importante otimizar as regras de URL para que leve menos tempo na análise de requisições e criação de URLs.

Utilizando rotas parametrizadas, você reduz o número de regras de URL, na qual pode melhorar significativamente o desempenho.

Na análise e criação de URLs, o **gerenciador de URL** examina as regras de URL na ordem em que foram declaradas. Portanto, você pode considerar ajustar a ordem destas regras, fazendo com que as regras mais específicas e/ou mais comuns sejam colocadas antes que os menos.

Se algumas regras de URL compartilharem o mesmo prefixo em seus padrões ou rotas, você pode considerar utilizar o `yii\web\GroupUrlRule`

para que sejam examinados de forma mais eficiente pelo gerenciador de URL como um grupo. Normalmente é o caso de aplicações compostos por módulos, onde cada módulo possui o seu próprio conjunto de regras de URL utilizando o ID do módulo como prefixo comum.

4.4 Requisições

As requisições realizadas na aplicação são representadas pelo objeto `yii\web\Request` que fornece informações como os parâmetros da requisição, cabeçalhos HTTP, cookies e etc. Em uma determinada requisição, você pode acessar o objeto da requisição correspondente através do [componente da aplicação](#) `request`, que é uma instância de `yii\web\Request`, por padrão. Nesta seção, descreveremos como você pode usar este componente em sua aplicação.

4.4.1 Parâmetros da Requisição

Para obter os parâmetros da requisição, você pode chamar os métodos `get()` e `post()` do componente `request`. Estes métodos retornam os valores de `$_GET` e `$_POST`, respectivamente. Por exemplo,

```
$request = Yii::$app->request;

$get = $request->get();
// equivalente à: $get = $_GET;

$id = $request->get('id');
// equivalente à: $id = isset($_GET['id']) ? $_GET['id'] : null;

$id = $request->get('id', 1);
// equivalente à: $id = isset($_GET['id']) ? $_GET['id'] : 1;

$post = $request->post();
// equivalente à: $post = $_POST;

$name = $request->post('name');
// equivalente à: $name = isset($_POST['name']) ? $_POST['name'] : null;

$name = $request->post('name', '');
// equivalente à: $name = isset($_POST['name']) ? $_POST['name'] : '';
```

Informação: Ao invés de acessar diretamente o `$_GET` e o `$_POST` para recuperar os parâmetros da requisição, é recomendável que os utilizem através do componente `request`, como mostrado nos exemplos acima. Isto permite que você escreva testes de forma mais simples, utilizando um componente da requisição que retornem valores pré-determinados.

Ao implementar o **RESTful APIs**, muitas vezes você precisará recuperar os parâmetros que foram enviados pelos métodos de requisição PUT, PATCH ou outro. Você pode recuperá-los chamando o método `yii\web\Request::getBodyParam()`. Por exemplo,

```
$request = Yii::$app->request;

// retorna todos os parâmetros
$params = $request->bodyParams;

// retorna o parâmetro "id"
$params = $request->getBodyParam('id');
```

Informação: Tirando os parâmetros GET, os parâmetros POST, PUT, PATCH e etc são enviados no corpo da requisição. O componente `request` analisará estes parâmetros quando você acessá-los através dos métodos descritos acima. Você pode personalizar a forma como estes parâmetros são analisados pela configuração da propriedade `yii\web\Request::$parsers`.

4.4.2 Métodos da Requisição

Você pode obter o método HTTP usado pela requisição atual através da expressão `Yii::$app->request->method`. Um conjunto de propriedades booleanas também são fornecidos para que você consiga verificar se o método atual é o correto. Por exemplo,

```
$request = Yii::$app->request;

if ($request->isAjax) { /* a requisição é uma requisição Ajax */ }
if ($request->isGet) { /* o método da requisição é GET */ }
if ($request->isPost) { /* o método da requisição é POST */ }
if ($request->isPut) { /* o método da requisição é PUT */ }
```

4.4.3 URLs da Requisição

O componente `request` fornece muitas formas de inspecionar a atual URL da requisição. Assumindo que a URL da requisição seja `https://example.com/admin/index.php/product?id=100`, você pode obter várias partes desta URL através das propriedades explicadas a seguir:

- `url`: retorna `/admin/index.php/product?id=100`, que é a URL sem as informações de protocolo e de domínio.
- `absoluteUrl`: retorna `https://example.com/admin/index.php/product?id=100`, que é a URL completa, incluindo as informações de protocolo e de domínio.
- `hostInfo`: retorna `https://example.com`, que são as informações de protocolo e de domínio da URL.

- **pathInfo**: retorna `/product`, que é a informação depois do script de entrada e antes da interrogação (da query string).
- **queryString**: retorna `id=100`, que é a informação depois da interrogação.
- **baseUrl**: retorna `/admin`, que é a informação depois do domínio e antes do script de entrada.
- **scriptUrl**: retorna `/admin/index.php`, que é a informação depois do domínio até o script de entrada, inclusive.
- **serverName**: retorna `example.com`, que é o domínio da URL.
- **serverPort**: retorna `80`, que é a porta usada pelo servidor Web.

4.4.4 Cabeçalho HTTP

Você pode obter as informações do cabeçalho HTTP através da coleção de cabeçalho retornado pela propriedade `yii\web\Request::$headers`. Por exemplo,

```
// $headers é um objeto de yii\web\HeaderCollection
$headers = Yii::$app->request->headers;

// retorna o valor do cabeçalho Accept
$accept = $headers->get('Accept');

if ($headers->has('User-Agent')) { /* existe o cabeçalho User-Agent */ }
```

O componente `request` também fornece suporte para fácil acesso de alguns cabeçalhos mais utilizados, incluindo:

- **userAgent**: retorna o valor do cabeçalho `User-Agent`.
- **contentType**: retorna o valor do cabeçalho `Content-Type` que indica o tipo MIME dos dados do corpo da requisição.
- **acceptableContentTypes**: retorna os tipos MIME acessíveis pelos usuários. Os tipos retornados são ordenados pela sua pontuação de qualidade. Tipos com mais pontuação aparecerão nas primeiras posições.
- **acceptableLanguages**: retorna os idiomas acessíveis pelos usuários. Os idiomas retornados são ordenados pelo nível de preferência. O primeiro elemento representa o idioma de maior preferência.

Se a sua aplicação suportar diversos idiomas e quiser exibir páginas no idioma de maior preferência do usuário, você pode usar o método de negociação `yii\web\Request::getPreferredLanguage()`. Este método pega uma lista de idiomas suportadas pela sua aplicação e compara com `acceptableLanguages`, para retornar o idioma mais adequado.

Dica: Você também pode utilizar o filtro `ContentNegotiator` para determinar dinamicamente qual tipo de conteúdo e idioma que deve ser utilizado na resposta. O filtro implementa negociação de conteúdo em cima das propriedades e métodos descritos acima.

4.4.5 Informações do Cliente

Você pode obter o nome do domínio ou endereço IP da máquina do cliente através das propriedades `userHost` e `userIP`, respectivamente. Por exemplo,

```
$userHost = Yii::$app->request->userHost;  
$userIP = Yii::$app->request->userIP;
```

Error: not existing file: runtime-responses.md

4.5 Sessões e Cookies

Sessões e cookies permitem que dados sejam persistentes entre várias requisições de usuários. No PHP puro você pode acessá-los através das variáveis globais `$_SESSION` e `$_COOKIE`, respectivamente. Yii encapsula sessões e cookies como objetos e portanto, permite que você possa acessá-los de um modo orientado à objetos com melhorias adicionais úteis.

4.5.1 Sessões

Assim como [requisições](#) e [respostas](#), você pode ter acesso a sessões através do [componente de aplicação](#) `session` que é uma instância de `yii\web\Session`, por padrão.

Abrindo e Fechando Sessões

Para abrir e fechar uma sessão, você pode fazer o seguinte:

```
$session = Yii::$app->session;

// verifica se a sessão está pronta para abrir
if ($session->isActive) ...

// abre uma sessão
$session->open();

// fecha uma sessão
$session->close();

// destrói todos os dados registrados em uma sessão.
$session->destroy();
```

Você pode chamar `open()` and `close()` várias vezes, sem causar erros; internamente os métodos irão verificar primeiro se a sessão já está aberta.

Acessando Dados da Sessão

Para acessar os dados armazenados em sessão, você pode fazer o seguinte:

```
$session = Yii::$app->session;

// obter uma variável de sessão. Os exemplos abaixo são equivalentes:
$language = $session->get('language');
$language = $session['language'];
$language = isset($_SESSION['language']) ? $_SESSION['language'] : null;

// definir uma variável de sessão. Os exemplos abaixo são equivalentes:
$session->set('language', 'en-US');
$session['language'] = 'en-US';
$_SESSION['language'] = 'en-US';
```

```
// remover uma variável de sessão. Os exemplos abaixo são equivalentes:
$session->remove('language');
unset($session['language']);
unset($_SESSION['language']);

// verifica se a variável de sessão existe. Os exemplos abaixo são
equivalentes:
if ($session->has('language')) ...
if (isset($session['language'])) ...
if (isset($_SESSION['language'])) ...

// percorrer todas as variáveis de sessão. Os exemplos abaixo são
equivalentes:
foreach ($session as $name => $value) ...
foreach ($_SESSION as $name => $value) ...
```

Observação: Quando você acessa os dados da sessão através do componente `session`, uma sessão será automaticamente aberta caso não tenha sido feito antes. Isso é diferente de acessar dados da sessão através de `$_SESSION`, o que requer uma chamada explícita de `session_start()`.

Ao trabalhar com dados de sessão que são arrays, o componente `session` tem uma limitação que o impede de modificar diretamente um elemento do array. Por exemplo,

```
$session = Yii::$app->session;

// o seguinte código não funciona
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// o seguinte código funciona:
$session['captcha'] = [
    'number' => 5,
    'lifetime' => 3600,
];

// o seguinte código também funciona:
echo $session['captcha']['lifetime'];
```

Você pode usar uma das seguintes soluções para resolver este problema:

```
$session = Yii::$app->session;

// use diretamente $_SESSION (certifique-se que Yii::$app->session->open()
tenha sido chamado)
$_SESSION['captcha']['number'] = 5;
$_SESSION['captcha']['lifetime'] = 3600;

// obter todo o array primeiro, modificá-lo e depois salvá-lo
```

```
$captcha = $session['captcha'];
$captcha['number'] = 5;
$captcha['lifetime'] = 3600;
$session['captcha'] = $captcha;

// use ArrayObject em vez de array
$session['captcha'] = new \ArrayObject;
...
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// armazenar dados de array utilizando chaves com um prefixo comum
$session['captcha.number'] = 5;
$session['captcha.lifetime'] = 3600;
```

Para um melhor desempenho e legibilidade do código, recomendamos a última solução alternativa. Isto é, em vez de armazenar um array como uma variável de sessão única, você armazena cada elemento do array como uma variável de sessão que compartilha o mesmo prefixo de chave com outros elementos do array.

Armazenamento de Sessão Personalizado

A classe padrão `yii\web\Session` armazena dados da sessão como arquivos no servidor. Yii Também fornece as seguintes classes de sessão implementando diferentes formas de armazenamento:

- `yii\web\DbSession`: armazena dados de sessão em uma tabela no banco de dados.
- `yii\web\CacheSession`: armazena dados de sessão em um cache com a ajuda de um [cache component](#) configurado.
- `yii\redis\Session`: armazena dados de sessão utilizando [redis](#)³ como meio de armazenamento.
- `yii\mongodb\Session`: armazena dados de sessão em um [MongoDB](#)⁴.

Todas essas classes de sessão suportam o mesmo conjunto de métodos da API. Como resultado, você pode mudar para uma classe de armazenamento de sessão diferente, sem a necessidade de modificar o código da aplicação que usa sessões.

Observação: Se você deseja acessar dados de sessão via `$_SESSION` enquanto estiver usando armazenamento de sessão personalizado, você deve certificar-se de que a sessão já foi iniciada por `yii\web\Session::open()`. Isso ocorre porque os manipuladores de armazenamento de sessão personalizada são registrados dentro deste método.

³<https://redis.io/>

⁴<https://www.mongodb.com/>

Para saber como configurar e usar essas classes de componentes, por favor consulte a sua documentação da API. Abaixo está um exemplo que mostra como configurar `yii\web\DbSession` na configuração da aplicação para usar uma tabela do banco de dados para armazenamento de sessão:

```
return [
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',
            // 'db' => 'mydb', // ID do componente de aplicação da conexão
            // DB. O padrão é 'db'.
            // 'sessionTable' => 'my_session', // nome da tabela de sessão.
            // O padrão é 'session'.
        ],
    ],
];
```

Você também precisa criar a tabela de banco de dados a seguir para armazenar dados de sessão:

```
CREATE TABLE session
(
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

onde ‘BLOB’ refere-se ao tipo BLOB do seu DBMS preferido. Estes são os tipos de BLOB que podem ser usados para alguns SGBD populares:

- MySQL: LONGBLOB
- PostgreSQL: BYTEA
- MSSQL: BLOB

Observação: De acordo com a configuração `session.hash_function` no `php.ini`, pode ser necessário ajustar o tamanho da coluna `id`. Por exemplo, se a configuração for `session.hash_function=sha256`, você deve usar um tamanho de 64 em vez de 40.

Dados Flash

Dados flash é um tipo especial de dados de sessão que, uma vez posto em uma requisição, só estarão disponíveis durante a próxima requisição e serão automaticamente excluídos depois. Dados flash são mais comumente usados para implementar mensagens que só devem ser exibidas para os usuários finais, uma vez, tal como uma mensagem de confirmação exibida após um usuário submeter um formulário com sucesso.

Você pode definir e acessar dados de flash através do componente da aplicação `session`. Por exemplo,


```

$session = Yii::$app->session;

// Request #1
// defini uma mensagem flash chamada "postDeleted"
$session->setFlash('postDeleted', 'You have successfully deleted your
post.');
```

```

// Request #2
// exibe uma mensagem flash chamada "postDeleted"
echo $session->getFlash('postDeleted');
```

```

// Request #3
// $result será falso uma vez que a mensagem flash foi automaticamente
excluída
$result = $session->hasFlash('postDeleted');
```

Assim como os dados da sessão regular, você pode armazenar dados arbitrários como os dados flash.

Quando você chama `yii\web\Session::setFlash()`, ele substituirá todos os dados flash existente que tenha o mesmo nome. Para acrescentar novos dados flash para uma mensagem existente com o mesmo nome, você pode chamá `yii\web\Session::addFlash()`. Por exemplo:

```

$session = Yii::$app->session;

// Request #1
// adicionar algumas mensagens flash com o nome de "alerts"
$session->addFlash('alerts', 'You have successfully deleted your post.');
```

```

$session->addFlash('alerts', 'You have successfully added a new friend.');
```

```

$session->addFlash('alerts', 'You are promoted.');
```

```

// Request #2
// $alerts é um array de mensagens flash com o nome de "alerts"
$alerts = $session->getFlash('alerts');
```

Observação: Tente não usar `yii\web\Session::setFlash()` junto com `yii\web\Session::addFlash()` para dados flash com o mesmo nome. Isto porque o último método coloca os dados flash automaticamente em um array, de modo que ele pode acrescentar novos dados flash com o mesmo nome. Como resultado, quando você chamar `yii\web\Session::getFlash()`, você pode às vezes achar que está recebendo um array, quando na verdade você está recebendo uma string, dependendo da ordem da invocação destes dois métodos.

Dica: Para mostrar mensagens flash você pode usar o widget `yii\bootstrap\Alert` da seguinte forma:

```

echo Alert::widget([
    'options' => ['class' => 'alert-info'],
    'body' => Yii::$app->session->getFlash('postDeleted'),
]);
```

4.5.2 Cookies

Yii representa cada cookie como um objeto de `yii\web\Cookie`. Ambos, `yii\web\Request` e `yii\web\Response` mantêm uma coleção de cookies através da propriedade chamada `cookies`. A coleção de cookie no primeiro representa os cookies submetidos na requisição, enquanto a coleção de cookie no último representa os cookies que são para serem enviados ao usuário.

Lendo Cookies

Você pode obter os cookies na requisição corrente usando o seguinte código:

```
// pega a coleção de cookie (yii\web\CookieCollection) do componente
"request"
$cookies = Yii::$app->request->cookies;

// pega o valor do cookie "language". se o cookie não existir, retorna "en"
como o valor padrão.
$language = $cookies->getValue('language', 'en');

// um caminho alternativo para pegar o valor do cookie "language"
if (($cookie = $cookies->get('language')) !== null) {
    $language = $cookie->value;
}

// você também pode usar $cookies como um array
if (isset($cookies['language'])) {
    $language = $cookies['language']->value;
}

// verifica se existe um cookie "language"
if ($cookies->has('language')) ...
if (isset($cookies['language'])) ...
```

Enviando Cookies

Você pode enviar cookies para o usuário final utilizando o seguinte código:

```
// pega a coleção de cookie (yii\web\CookieCollection) do componente
"response"
$cookies = Yii::$app->response->cookies;

// adicionar um novo cookie a resposta que será enviado
$cookies->add(new \yii\web\Cookie([
    'name' => 'language',
    'value' => 'zh-CN',
]));

// remove um cookie
$cookies->remove('language');
// outra alternativa para remover um cookie
unset($cookies['language']);
```

Além das propriedades `name`, `value` mostradas nos exemplos acima, a classe `yii\web\Cookie` também define outras propriedades para representar plenamente todas as informações de cookie disponíveis, tal como `domain`, `expire`. Você pode configurar essas propriedades conforme necessário para preparar um cookie e, em seguida, adicioná-lo à coleção de cookie da resposta.

Observação: Para melhor segurança, o valor padrão de `yii\web\Cookie::$httpOnly` é definido para `true`. Isso ajuda a reduzir o risco de um script do lado do cliente acessar o cookie protegido (se o browser suporta-lo). Você pode ler o `httpOnly` wiki article⁵ para mais detalhes.

Validação de Cookie

Quando você está lendo e enviando cookies através dos componentes `request` e `response` como mostrado nas duas últimas subseções, você aproveita a segurança adicional de validação de cookie que protege que os cookies sejam modificados no lado do cliente. Isto é conseguido através da assinatura de cada cookie com um hash string, que permite a aplicação dizer se o cookie foi modificado no lado do cliente. Se assim for, o cookie não será acessível através do `cookie collection` do componente `request`.

Observação: Validação de cookie apenas protege os valores dos cookies de serem modificados. Se um cookie não for validado, você ainda pode acessá-lo através de `$_COOKIE`. Isso ocorre porque as bibliotecas de terceiros podem manipular os cookies de forma independente que não envolve a validação de cookie.

Validação de cookie é habilitada por padrão. Você pode desabilitá-la definindo a propriedade `yii\web\Request::$enableCookieValidation` para `false`, entretanto recomendamos fortemente que você não o faça.

Observação: Cookies que são enviados/recebidos diretamente através de `$_COOKIE` e `setcookie()` NÃO serão validados.

Ao usar a validação de cookie, você deve especificar uma `yii\web\Request::$cookieValidationKey` que será usada para gerar o supracitado hash strings. Você pode fazê-lo através da configuração do componente `request` na configuração da aplicação:

```
return [  
    'components' => [  
        'request' => [  
            'cookieValidationKey' => 'fill in a secret key here',  
        ],  
    ],  
];
```

⁵<https://owasp.org/www-community/HttpOnly>

Observação: `cookieValidationKey` é fundamental para a segurança da sua aplicação. Ela só deve ser conhecida por pessoas de sua confiança. Não armazená-la no sistema de controle de versão.

4.6 Tratamento de Erros

O Yii inclui um próprio **tratamento de erro** que o torna uma experiência muito mais agradável do que antes. Em particular, o manipulador de erro do Yii faz o seguinte para melhorar o tratamento de erros:

- Todos os erros não-fatais do PHP (ex. advertências, avisos) são convertidas em exceções capturáveis.
- Exceções e erros fatais do PHP são exibidos com detalhes de informação em uma pilha de chamadas (call stack) e linhas de código-fonte no modo de depuração.
- Suporta o uso de uma **ação do controller** dedicado para exibir erros.
- Suporta diferentes formatos de resposta de erro.

O **manipulador de erro** é habilitado por padrão. Você pode desabilitá-lo definindo a constante `YII_ENABLE_ERROR_HANDLER` como `false` no **script de entrada** da aplicação.

4.6.1 Usando Manipulador de Erro

O manipulador de erro é registrado como um **componente da aplicação** chamado `errorHandler`. Você pode configurá-lo na configuração da aplicação da seguinte forma:

```
return [  
    'components' => [  
        'errorHandler' => [  
            'maxSourceLines' => 20,  
        ],  
    ],  
];
```

Com a configuração acima, o número de linhas de código fonte para ser exibido em páginas de exceção será de até 20.

Como já informado, o manipulador de erro transforma todos os erros não fatais do PHP em exceções capturáveis. Isto significa que você pode usar o seguinte código para lidar com erros do PHP:

```
use Yii;  
use yii\base\ErrorException;  
  
try {  
    10/0;  
} catch (ErrorException $e) {
```

```
Yii::warning("Division by zero.");  
}  
  
// Continua a execução...
```

Se você deseja mostrar uma página de erro dizendo ao usuário que a sua requisição é inválida ou inesperada, você pode simplesmente lançar uma exceção HTTP, tal como `yii\web\NotFoundException`. O manipulador de erro irá definir corretamente o código de status HTTP da resposta e usar uma exibição de erro apropriada para exibir a mensagem de erro.

```
use yii\web\NotFoundException;  
  
throw new NotFoundException();
```

4.6.2 Personalizando a Exibição de Erro

O manipulador de erro ajusta a exibição de erro de acordo com o valor da constante `YII_DEBUG`. Quando `YII_DEBUG` for `True` (significa modo de debug), o manipulador de erro irá exibir exceções com informações detalhadas da pilha de chamadas e linhas do código fonte para ajudar na depuração do erro. E quando `YII_DEBUG` for `false`, apenas a mensagem de erro será exibida para evitar revelar informações relevantes sobre a aplicação.

Observação: Se uma exceção descende de `yii\base\UserException`, nenhuma pilha de chamadas será exibido independentemente do valor do `YII_DEBUG`. Isso porque tais exceções são consideradas erros causados pelo usuário não havendo nada a ser corrigido por parte dos programadores.

Por padrão, o manipulador de erro mostra os erros utilizando duas [views](#):

- `@yii/views/errorHandler/error.php`: utilizada quando os erros devem ser exibidos sem informações pilha de chamadas. Quando `YII_DEBUG` for `false`, esta é a única exibição de erro a ser exibida.
- `@yii/views/errorHandler/exception.php`: utilizada quando os erros devem ser exibidos com informações pilha de chamadas. Você pode configurar as propriedades `errorView` e `exceptionView` do manipulador de erros para usar suas próprias views personalizando a exibição de erro.

Usando Ações de Erros

A melhor maneira de personalizar a exibição de erro é usar uma [ação](#) dedicada para este fim. Para fazê-lo, primeiro configure a propriedade `errorAction` do componente `errorHandler` como a seguir:

```
return [  
    'components' => [  
        'errorAction' => 'site/error',  
    ],  
];
```

```

        'errorHandler' => [
            'errorAction' => 'site/error',
        ],
    ],
];

```

A propriedade `errorAction` define uma [rota](#) para uma ação. A configuração acima afirma que quando um erro precisa ser exibido sem informações da pilha de chamadas, a ação `site/error` deve ser executada.

Você pode criar a ação `site/error` da seguinte forma,

```

namespace app\controllers;

use Yii;
use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'error' => [
                'class' => 'yii\web\ErrorAction',
            ],
        ];
    }
}

```

O código acima define a ação `error` usando a classe `yii\web\ErrorAction` que renderiza um erro usando a view `error`.

Além de usar `yii\web\ErrorAction`, você também pode definir a ação `error` usando um método da ação como o seguinte,

```

public function actionError()
{
    $exception = Yii::$app->errorHandler->exception;
    if ($exception !== null) {
        return $this->render('error', ['exception' => $exception]);
    }
}

```

Agora você deve criar um arquivo de exibição localizado na `views/site/error.php`. Neste arquivo de exibição, você pode acessar as seguintes variáveis se a ação de erro for definida como `yii\web\ErrorAction`:

- `name`: o nome do erro;
- `message`: a mensagem de erro;
- `exception`: o objeto de exceção através do qual você pode recuperar mais informações úteis, como o código de status HTTP, o código de erro, pilha de chamadas de erro, etc.

Observação: Se você está utilizando o [template básico](#) ou o [template avançado](#)⁶, a ação e a view de erro já estão definidas para você.

Customizando o Formato da Resposta de Erro

O manipulador de erro exibe erros de acordo com a definição do formato da [resposta](#). Se o [formato da resposta](#) for `html`, ele usará a view de erro ou exceção para exibir os erros, como descrito na última subseção. Para outros formatos de resposta, o manipulador de erro irá atribuir o array de representação da exceção para a propriedade `yii\web\Response::$data` que será então convertida para diferentes formatos de acordo com o que foi configurado. Por exemplo, se o formato de resposta for `json`, você pode visualizar a seguinte resposta:

```
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
}
```

Você pode personalizar o formato de resposta de erro, respondendo ao evento `beforeSend` do componente `response` na configuração da aplicação:

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
    ],
];
```

⁶<https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-pt-BR/README.md>

O código acima irá reformatar a resposta de erro como a seguir:

```
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 Log

O Yii fornece um poderoso framework de log que é altamente personalizável e extensível. Utilizando este framework, você pode facilmente gerar logs de vários tipos de mensagens, filtrá-las, e salva-las em diferentes meios, tais como arquivos, banco de dados, e-mails.

Usar o Yii framework Log envolve os seguintes passos:

- Registrar mensagens de log de vários lugares do seu código;
- Configurar o destino de log na configuração da aplicação para filtrar e exportar mensagens de log;
- Examinar as mensagens de erro exportadas (ex.: Yii debugger).

Nesta seção, vamos descrever principalmente os dois primeiros passos.

4.7.1 Gravar Mensagens

Gravar mensagens de log é tão simples como chamar um dos seguintes métodos de registro:

- `Yii::debug()`: gravar uma mensagem para rastrear como um determinado trecho de código é executado. Isso é principalmente para o uso de desenvolvimento.
- `Yii::info()`: gravar uma mensagem que transmite algumas informações úteis.
- `Yii::warning()`: gravar uma mensagem de aviso que indica que algo inesperado aconteceu.
- `Yii::error()`: gravar um erro fatal que deve ser investigado o mais rápido possível.

Estes métodos gravam mensagens de log em vários *níveis* e *categorias*. Eles compartilham a mesma assinatura de função `function ($message, $category`

= 'application'), onde `$message` significa a mensagem de log a ser gravada, enquanto `$category` é a categoria da mensagem de log. O código no exemplo a seguir registra uma mensagem de rastreamento sob a categoria padrão `application`:

```
Yii::debug('start calculating average revenue');
```

Observação: Mensagens de log podem ser strings, bem como dados complexos, tais como arrays ou objetos. É da responsabilidade dos destinos de log lidar adequadamente com as mensagens de log. Por padrão, se uma mensagem de log não for uma string, ela será exportada como uma string chamando `yii\helpers\VarDumper::export()`.

Para melhor organizar e filtrar as mensagens de log, é recomendável que você especifique uma categoria apropriada para cada mensagem de log. Você pode escolher um esquema de nomenclatura hierárquica para as categorias, o que tornará mais fácil para os destinos de log filtrar mensagens com base em suas categorias. Um esquema de nomes simples, mas eficaz é usar a constante mágica PHP `__METHOD__` para os nomes das categorias. Esta é também a abordagem utilizada no código central do framework Yii. Por exemplo,

```
Yii::debug('start calculating average revenue', __METHOD__);
```

A constante `__METHOD__` corresponde ao nome do método (prefixado com o caminho completo do nome da classe) onde a constante aparece. Por exemplo, é igual a string `'app\controllers\RevenueController::calculate'` se o código acima for chamado dentro deste método.

Observação: Os métodos de registro descritos acima são na verdade atalhos para o método `log()` do `logger object` que é um singleton acessível através da expressão `Yii::getLogger()`. Quando um determinado número de mensagens são logadas ou quando a aplicação termina, o objeto logger irá chamar um `message dispatcher` para enviar mensagens de log gravadas destinos de log.

4.7.2 Destinos de Log

Um destino de log é uma instância da classe `yii\log\Target` ou uma classe filha. Ele filtra as mensagens de log por seus níveis e categorias e, em seguida, às exportam para algum meio. Por exemplo, um **banco de dados de destino** exporta as mensagens de log para uma tabela no banco de dados, enquanto um **e-mail de destino** exporta as mensagens de log para algum e-mail especificado.

Você pode registrar vários destinos de log em uma aplicação configurando-os através do **componente da aplicação log** na configuração da aplicação, como a seguir:

```
return [
    // o componente "log" deve ser carregado durante o tempo de
    // inicialização
    'bootstrap' => ['log'],

    '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',
                    ],
                ],
            ],
        ],
    ],
];
```

Observação: O componente `log` deve ser carregado durante a **inicialização** para que ele possa enviar mensagens de log para alvos prontamente. É por isso que ele está listado no array `bootstrap` como mostrado acima.

No código acima, dois destinos de log são registrados na propriedade `yii\log\Dispatcher::$targets`:

- o primeiro seleciona mensagens de erro e de advertência e os salva em uma tabela de banco de dados;
- o segundo seleciona mensagens de erro sob as categorias cujos nomes começam com `yii\db\`, e as envia para os e-mails `admin@example.com` e `developer@example.com`.

Yii vem com os seguintes destinos de log preparados. Por favor consulte a documentação da API sobre essas classes para aprender como configurar e usá-los.

- `yii\log\DbTarget`: armazena mensagens de log em uma tabela de banco de dados.
- `yii\log\EmailTarget`: envia mensagens de log para um endereço de e-mail pré-definido.

- `yii\log\FileTarget`: salva mensagens de log em arquivos.
- `yii\log\SyslogTarget`: salva mensagens de log para o syslog chamando a função PHP `syslog()`.

A seguir, vamos descrever as características comuns a todos os destinos de log.

Filtragem de Mensagem

Para cada destino de log, você pode configurar suas propriedades `levels` e `categories` para especificar que os níveis e categorias das mensagens o destino de log deve processar.

A propriedade `levels` é um array que consiste em um ou vários dos seguintes valores:

- `error`: corresponde a mensagens logadas por `Yii::error()`.
- `warning`: corresponde a mensagens logadas por `Yii::warning()`.
- `info`: corresponde a mensagens logadas por `Yii::info()`.
- `trace`: corresponde a mensagens logadas por `Yii::debug()`.
- `profile`: corresponde a mensagens logadas por `Yii::beginProfile()` e `Yii::endProfile()`, que será explicado em mais detalhes na subseção Perfil de Desempenho.

Se você não especificar a propriedade `levels`, significa que o alvo de log processará mensagens de *qualquer* nível.

A propriedade `categories` é um array que consiste em categorias de mensagens ou padrões. Um destino de log irá processar apenas mensagens cuja categoria possa ser encontrada ou corresponder a um dos padrões do array. Um padrão de categoria é um prefixo de nome de categoria com um asterisco `*` na sua extremidade. Um nome de categoria corresponde a um padrão de categoria se ela iniciar com o mesmo prefixo do padrão. Por exemplo, `yii\db\Command::execute` e `yii\db\Command::query` são usados como nome de categoria para as mensagens de log gravadas na classe `yii\db\Command`. Ambos correspondem ao padrão `yii\db*`. Se você não especificar a propriedade `categories`, significa que o destino de log processará mensagens de *qualquer* categoria.

Além de criar uma whitelist de categorias através da propriedade `categories`, você também pode criar uma blacklist de categorias através da propriedade `except`. Se a categoria da mensagem for encontrada ou corresponder a um dos padrões desta propriedade, ela não será processada pelo destino de log.

A próxima configuração de destino de log especifica que o destino deve processar somente mensagens de erro e alertas das categorias cujos nomes correspondam a `yii\db*` ou `yii\web\HttpException:*`, mas não correspondam a `yii\web\HttpException:404`.

```
[
    'class' => 'yii\log\FileTarget',
    'levels' => ['error', 'warning'],
```

```

'categories' => [
    'yii\db\*',
    'yii\web\HttpException:*',
],
'except' => [
    'yii\web\HttpException:404',
],
]

```

Observação: Quando uma exceção HTTP é capturada pelo `error handler`, uma mensagem de erro será logada com o nome da categoria no formato de `yii\web\HttpException:ErrorCode`. Por exemplo, o `yii\web\NotFoundHttpException` causará uma mensagem de erro da categoria `yii\web\HttpException:404`.

Formatando Mensagem

Destinos de log exportam as mensagens de logs filtradas em um determinado formato. Por exemplo, se você instalar um destino de log da classe `yii\log\FileTarget`, você pode encontrar uma mensagem de log semelhante à seguinte no `runtime/log/app.log` file:

```

2014-10-04 18:10:15 [::1] [] [-] [trace] [yii\base\Module::getModule] Loading
module: debug

```

Por padrão, mensagens de log serão formatadas do seguinte modo pelo `yii\log\Target::formatMessage()`:

```

Timestamp [IP address] [User ID] [Session ID] [Severity Level] [Category]
Message Text

```

Você pode personalizar este formato configurando a propriedade `yii\log\Target::$prefix` que recebe um PHP callable retornando um prefixo de mensagem personalizado. Por exemplo, o código a seguir configura um destino de log para prefixar cada mensagem de log com o ID do usuário corrente (Endereço IP e ID da sessão são removidos por razões de privacidade).

```

[
    '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]";
    }
]

```

Além de prefixos de mensagens, destinos de mensagens também anexa algumas informações de contexto para cada lote de mensagens de log. Por padrão, os valores destas variáveis globais PHP são incluídas: `$_GET`, `$_POST`,

`$_FILES`, `$_COOKIE`, `$_SESSION` e `$_SERVER`. Você pode ajustar este comportamento configurando a propriedade `yii\log\Target::$logVars` com os nomes das variáveis globais que você deseja incluir para o destino de log. Por exemplo, a seguinte configuração de destino de log especifica que somente valores da variável `$_SERVER` seriam anexadas as mensagens de log.

```
[
    'class' => 'yii\log\FileTarget',
    'logVars' => ['_SERVER'],
]
```

Você pode configurar `logVars` para ser um array vazio para desativar totalmente a inclusão de informações de contexto. Ou se você quiser implementar sua própria maneira de fornecer informações de contexto, você pode sobrecrever o método `yii\log\Target::getContextMessage()`.

Nível de Rastreio de Mensagem

Durante o desenvolvimento, é desejável definir de onde cada mensagem de log virá. Isto pode ser conseguido por meio da configuração da propriedade `traceLevel` do componente `log` como a seguir:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [...],
        ],
    ],
];
```

A configuração da aplicação acima define o `traceLevel` para ser 3 se `YII_DEBUG` estiver ligado e 0 se `YII_DEBUG` estiver desligado. Isso significa, se `YII_DEBUG` estiver ligado, cada mensagem de log será anexada com no máximo 3 níveis de call stack (pilhas de chamadas) em que a mensagem de log é registrada; e se `YII_DEBUG` estiver desligado, nenhuma informação do call stack será incluída.

Observação: Obter informação do call stack não é trivial. Portanto, você deverá usar somente este recurso durante o desenvolvimento ou durante o debug da aplicação.

Libertação e Exportação de Mensagens

Como já mencionado, mensagens de log são mantidas em um array através do objeto `logger`. Para limitar o consumo de memória por este array, o objeto `logger` irá liberar as mensagens gravadas para os destinos de log cada vez que o array acumula um certo número de mensagens de log. Você

pode personalizar este número configurando a propriedade `flushInterval` do componente `log`:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 100,    // o padrão é 1000
            'targets' => [...],
        ],
    ],
];
```

Observação: Liberação de mensagens também acontece quando a aplicação termina, o que garante que alvos de log possam receber as informações completas de mensagens de log.

Quando o `logger object` libera mensagens de log para os alvos de log, elas não são exportadas imediatamente. Em vez disso, a exportação de mensagem só ocorre quando o alvo de log acumula certo número de mensagens filtradas. Você pode personalizar este número configurando a propriedade `exportInterval` de cada alvo de log, como a seguir,

```
[
    'class' => 'yii\log\FileTarget',
    'exportInterval' => 100,    // default is 1000
]
```

Devido a configuração de nível, liberação e exportação, por padrão quando você chama `Yii::debug()` ou qualquer outro método de log, você NÃO verá a mensagem de log imediatamente no destino. Isto poderia ser um problema para algumas aplicações console de longa execução. Para fazer cada mensagem de log aparecer imediatamente no destino, você deve configurar ambos `flushInterval` e `exportInterval` para 1, como mostrado a seguir:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 1,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'exportInterval' => 1,
                ],
            ],
        ],
    ],
];
```

Observação: A frequente liberação e exportação de mensagens irá degradar o desempenho da sua aplicação.

Alternando Destinos de Log

Você pode habilitar ou desabilitar um destino de log configurando a propriedade `enabled`. Você pode fazê-lo através da configuração do destino de log ou pela seguinte declaração em seu código PHP:

```
Yii::$app->log->targets['file']->enabled = false;
```

O código acima requer que você nomeie um destino como `file`, como mostrado acima usando chaves de string no array `targets`:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'targets' => [
                'file' => [
                    'class' => 'yii\log\FileTarget',
                ],
                'db' => [
                    'class' => 'yii\log\DbTarget',
                ],
            ],
        ],
    ],
];
```

Criando Novos Destinos

Criar uma nova classe de destino de log é muito simples. Você primeiramente precisa implementar o método `yii\log\Target::export()` enviando o conteúdo do array `yii\log\Target::$messages` para o meio designado. Você pode chamar o método `yii\log\Target::formatMessage()` para formatar cada mensagem. Para mais detalhes, você pode consultar qualquer uma das classes de destino de log incluído na versão Yii.

4.7.3 Perfil de Desempenho

Perfil de desempenho é um tipo especial de log de mensagem que é usado para medir o tempo que certos blocos de código demora e para descobrir quais são os gargalos de desempenho. Por exemplo, a classe `yii\db\Command` utiliza perfil de desempenho para descobrir o tempo que cada db query leva.

Para usar perfil de desempenho, primeiro identifique o bloco de código que precisa ser analisado. Então, encapsula cada bloco de código como o seguinte:

```
\Yii::beginProfile('myBenchmark');

...code block being profiled...

\Yii::endProfile('myBenchmark');
```

onde `myBenchmark` representa um token único de identificação de um bloco de código. Mais tarde quando você for examinar o resultado, você usará este token para localizar o tempo gasto pelo determinado bloco de código.

É importante certificar-se de que os pares de `beginProfile` e `endProfile` estão corretamente aninhadas. Por exemplo,

```
\Yii::beginProfile('block1');  
  
    // algum código a ser analisado  
  
    \Yii::beginProfile('block2');  
        // algum outro código a ser analisado  
    \Yii::endProfile('block2');  
  
\Yii::endProfile('block1');
```

Se você esquecer `\Yii::endProfile('block1')` ou trocar a ordem de `\Yii::endProfile('block1')` e `\Yii::endProfile('block2')`, o perfil de desempenho não funcionará.

Para cada bloco de código iniciado com `beginProfile`, uma mensagem de log com o nível `profile` é registrada. Você pode configurar um destino de log para coletar tais mensagens e exportá-las. O Yii debugger implementa um painel de perfil de desempenho mostrando os seus resultados.

Capítulo 5

Conceitos Chave

5.1 Componentes

Componente é a parte principal na construção de aplicações Yii. Componentes são instâncias de `yii\base\Component`, ou uma classe estendida. As três características principais que os componentes fornecem a outras classes são:

- [Propriedades](#)
- [Eventos](#)
- [Behaviors \(Comportamentos\)](#)

Separadamente e combinadas, essas características fazem com que as classes no Yii sejam muito mais customizáveis e fáceis de usar. Por exemplo, o `yii\jui\DatePicker`, um componente de interface do usuário, pode ser usado na view (visão) para gerar um `date picker` interativo:

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'ru',
    'name' => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

Os widgets são facilmente escritos porque a classe estende de `yii\base\Component`.

Enquanto os componentes são muito poderosos, eles são um pouco mais pesados do que um objeto normal, devido ao fato de que é preciso memória e CPU extra para dar suporte às funcionalidades de [evento](#) e de [behavior](#) em particular. Se o seu componente não precisa dessas duas características, você pode considerar estender a sua classe de componente de `yii\base\BaseObject` em vez de `yii\base\Component`. Se o fizer, fará com que seus

componentes sejam tão eficientes como objetos normais do PHP, mas com um suporte adicional para [propriedades](#).

Ao estender a classe de `yii\base\Component` ou `yii\base\BaseObject`, é recomendado que você siga as seguintes convenções:

- Se você sobrescrever o construtor, declare um parâmetro `$config` como último parâmetro do construtor, e em seguida passe este parâmetro para o construtor pai.
- Sempre chame o construtor pai *no final* do seu construtor reescrito.
- Se você sobrescrever o método `yii\base\BaseObject::init()`, certifique-se de chamar a implementação pai do `init` *no início* do seu método `init`.

Por Exemplo:

```
<?php

namespace yii\components\MyClass;

use yii\base\BaseObject;

class MyClass extends BaseObject
{
    public $prop1;
    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... initialization before configuration is applied

        parent::__construct($config);
    }

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

        // ... initialization after configuration is applied
    }
}
```

Seguindo essas orientações fará com que seus componentes sejam [configuráveis](#) quando forem criados. Por Exemplo:

```
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
// alternatively
$component = \Yii::createObject([
    'class' => MyClass::class,
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

Informação: Embora a forma de chamar `Yii::createObject()` pareça ser mais complicada, ela é mais poderosa porque ela é aplicada no topo de um [container de injeção de dependência](#).

A classe `yii\base\BaseObject` impõe o seguinte ciclo de vida do objeto:

1. Pré-inicialização dentro do construtor. Você pode definir valores de propriedade padrão aqui.
2. Configuração de objeto via `$config`. A configuração pode sobrescrever o valor padrão configurado dentro do construtor.
3. Pós-inicialização dentro do `init()`. Você pode sobrescrever este método para executar checagens e normalização das propriedades.
4. Chamadas de método de objeto.

Os três primeiros passos acontecem dentro do construtor do objeto. Isto significa que uma vez que você instanciar a classe (isto é, um objeto), esse objeto será inicializado adequadamente.

5.2 Propriedades

No PHP, atributos da classe são sempre chamadas de *propriedades*. Esses atributos fazem parte da definição da classe e são usadas para representar o estado de uma instância da classe (para diferenciar uma instância da classe de outra). Na prática, muitas vezes você pode querer lidar com a leitura ou a escrita de propriedades de maneiras especiais. Por exemplo, você pode querer trinar uma string sempre que for atribuído um valor para a propriedade `label`. Você *poderia* usar o código a seguir para realizar esta tarefa:

```
$object->label = trim($label);
```

A desvantagem do código acima é que você teria que chamar `trim()` em todos os lugares onde você definir a propriedade `label` no seu código. Se, no futuro, a propriedade `label` receber um novo requisito, tais como a primeira letra deve ser capitalizado, você teria que modificar novamente todos os pedaços de código que atribui um valor para a propriedade `label`. A repetição de código leva a erros, e é uma prática que você deve evitar sempre que possível.

Para resolver este problema, o Yii introduz uma classe base chamada `yii\base\BaseObject` que suporta definições de propriedades baseadas nos métodos *getter* e *setter* da classe. Se uma classe precisar dessa funcionalidade, ela deve estender a classe `yii\base\BaseObject`, ou de uma classe-filha.

Informação: Quase todas as classes nativas (core) do framework Yii estendem de `yii\base\BaseObject` ou de uma classe-filha. Isto significa que sempre que você vê um método *getter* ou *setter* em uma classe nativa (core), você pode usá-lo como uma propriedade.

Um método *getter* é um método cujo nome inicia com a palavra `get`; um método *setter* inicia com `set`. O nome depois do prefixo `get` ou `set` define o nome da propriedade. Por exemplo, um *getter* `getLabel()` e/ou um *setter* `setLabel()` define a propriedade chamada `label`, como mostrado no código a seguir:

```
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);
    }
}
```

(Para ser claro, os métodos *getter* e *setter* criam a propriedade `label`, que neste caso internamente refere-se ao atributo privado chamado `_label`.)

Propriedades definidas por *getters* e *setters* podem ser usados como atributos da classe. A principal diferença é que quando tal propriedade é iniciada para leitura, o método *getter* correspondente será chamado; quando a propriedade é iniciada atribuindo um valor, o método *setter* correspondente será chamado. Por exemplo:

```
// equivalent to $label = $object->getLabel();
$label = $object->label;

// equivalent to $object->setLabel('abc');
$object->label = 'abc';
```

A propriedade definida por um método *getter* sem um método *setter* é *somente de leitura*. Tentando atribuir um valor a tal propriedade causará uma exceção `InvalidCallException`. Semelhantemente, uma propriedade definida por um método *setter* sem um método *getter* é *somente de gravação*, e

tentar ler tal propriedade também causará uma exceção. Não é comum ter propriedade *somente de gravação*.

Existem várias regras especiais para, e limitações sobre, as propriedades definidas via getters e setters:

- Os nomes dessas propriedades são *case-insensitive*. Por exemplo, `$object->label` e `$object->Label` são a mesma coisa. Isso ocorre porque nomes de métodos no PHP são case-insensitive.
- Se o nome de uma tal propriedade é o mesmo que um atributo da classe, esta última terá precedência. Por exemplo, se a classe `Foo` descrita acima tiver um atributo `label`, então a atribuição `$object->label = 'abc'` afetará o atributo `label`; esta linha não executaria `setLabel()` método setter.
- Essas propriedades não suportam visibilidade. Não faz nenhuma diferença para a definição dos métodos getter ou setter se a propriedade é pública, protegida ou privada.
- As propriedades somente podem ser definidas por getters e/ou setters *não estáticos*. Os métodos estáticos não serão tratados da mesma maneira.

Voltando para o problema descrito no início deste guia, em vez de chamar `trim()` em todos os lugares que um valor for atribuído a `label`, agora o `trim()` só precisa ser invocado dentro do setter `setLabel()`. E se uma nova exigência faz com que seja necessário que o `label` seja inicializado capitalizado, o método `setLabel()` pode rapidamente ser modificado sem tocar em nenhum outro código. Esta única mudança afetará de forma global cada atribuição à propriedade `label`.

5.3 Eventos

Eventos permitem que você injete código personalizado dentro de outro código existente em determinados pontos de execução. Você pode anexar o código personalizado a um evento de modo que ao acionar o evento, o código é executado automaticamente. Por exemplo, um objeto de e-mail pode disparar um evento `messageSent` quando o envio da mensagem for bem sucedida. Se você quiser acompanhar as mensagens que são enviadas com sucesso, você poderia então simplesmente anexar o código de acompanhamento ao evento `messageSent`. O Yii disponibiliza uma classe base chamada `yii\base\Component` para dar suporte aos eventos. Se sua classe precisar disparar eventos, ela deverá estender de `yii\base\Component`, ou de uma classe-filha.

5.3.1 Manipuladores de Evento

Um manipulador de evento é uma função [Callback do PHP] (https://www.php.net/manual/pt_BR/language.types.callable.php) que é executada

quando o evento é disparado. Você pode usar qualquer um dos seguintes callbacks:

- uma função global do PHP especificada como uma string (sem parênteses), por exemplo, 'trim';
- Um método do objeto especificado como um array, informando o objeto e um nome do método como uma string (sem parênteses), por exemplo [*\$object*, 'methodName'];
- Um método estático da classe especificado como um array informando o nome da classe e nome do método como string (sem parênteses), por exemplo, ['ClassName', 'methodName'];
- Uma função anônima, por exemplo, `function ($event) { ... }`.

A assinatura de um manipulador de eventos é a seguinte:

```
function ($event) {  
    // $event is an object of yii\base\Event or a child class  
}
```

Através do parâmetro `$event`, um manipulador de evento pode receber as seguintes informações sobre o evento que ocorreu:

- nome do evento
- objeto chamador: o objeto cujo método `trigger()` foi chamado
- dados personalizados: os dados que são fornecidos ao anexar o manipulador de eventos (a ser explicado a seguir)

5.3.2 Anexando manipuladores de eventos

Você pode anexar um manipulador para um evento, chamando o método `yii\base\Component::on()`. Por exemplo:

```
$foo = new Foo;  
  
// esse manipulador é uma função global  
$foo->on(Foo::EVENT_HELLO, 'function_name');  
  
// esse manipulador é um método de objeto  
$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);  
  
// esse manipulador é um método estático da classe  
$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);  
  
// esse manipulador é uma função anônima  
$foo->on(Foo::EVENT_HELLO, function ($event) {  
    // Código ...  
});
```

Você também pode anexar manipuladores de eventos por meio de [configurações](#). Para mais detalhes, consulte a seção [Configurações](#).

Ao anexar um manipulador de eventos, você pode fornecer dados adicionais no terceiro parâmetro do método `yii\base\Component::on()`. Os

dados serão disponibilizados para o manipulador quando o evento for disparado e o manipulador chamado. Por exemplo:

```
// O código a seguir mostrará "abc" quando o evento for disparado
// porque $event->data contém os dados passados no terceiro parâmetro do
"on"
$foo->on(Foo::EVENT_HELLO, 'function_name', 'abc');

function function_name($event) {
    echo $event->data;
}
```

5.3.3 Ordem dos Manipuladores de Eventos

Você pode anexar um ou mais manipuladores para um único evento. Quando o evento é disparado, os manipuladores anexados serão chamados na ordem em que eles foram anexados ao evento. Se o manipulador precisar interromper os eventos subsequentes, pode definir a propriedade `yii\base\Event::$handled` do parâmetro `$event` para `true`:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    $event->handled = true;
});
```

Por padrão, um novo manipulador é anexado a fila de manipuladores existente para o evento.

Como resultado, o manipulador será chamado por último quando o evento for disparado.

Para inserir um novo manipulador de evento no início da fila de modo a ser chamado primeiro, você pode chamar o método `yii\base\Component::on()`, passando `false` para o quarto parâmetro `$append`:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // ...
}, $data, false);
```

5.3.4 Disparando Eventos

Os eventos são disparados chamando o método `yii\base\Component::trigger()`. Este método requer um *nome de evento*, e, opcionalmente, um objeto de evento que descreve os parâmetros a serem passados para os manipuladores de eventos. Por exemplo:

```
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);
    }
}

```

Com o código acima, todas as chamadas para `bar()` irão disparar um evento chamado `hello`.

Dica: Recomenda-se usar constantes de classe para representar nomes de eventos. No exemplo acima, a constante `EVENT_HELLO` representa o evento `hello`. Esta abordagem tem três benefícios. Primeiro, previne erros de digitação. Segundo, pode fazer o evento se tornar reconhecível para recursos de *auto-complete* de IDEs. Terceiro, você pode especificar quais eventos são suportados em uma classe, basta verificar suas declarações de constantes.

Às vezes, quando um evento é disparado você pode querer passar junto informações adicionais para os manipuladores de eventos. Por exemplo, um objeto de e-mail pode querer passar uma informação para o manipulador do evento `messageSent` de modo que os manipuladores podem conhecer os detalhes das mensagens enviadas. Para fazer isso, você pode fornecer um objeto de evento como o segundo parâmetro para o método `yii\base\Component::trigger()`. Este objeto precisa ser uma instância da classe `yii\base\Event` ou de uma classe filha. Por exemplo:

```

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);
    }
}

```



```
}  
}
```

Quando o método `yii\base\Component::trigger()` é chamado, ele chamará todos os manipuladores ligados ao evento passado.

5.3.5 Desvinculando manipuladores de eventos

Para retirar um manipulador de um evento, chame o método `yii\base\Component::off()`. Por Exemplo:

```
// o manipulador é uma função global  
$foo->off(Foo::EVENT_HELLO, 'function_name');
```

```
// o manipulador é um método de objeto  
$foo->off(Foo::EVENT_HELLO, [$object, 'methodName']);
```

```
// o manipulador é um método de estático da Classe  
$foo->off(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);
```

```
// o manipulador é uma função anônima  
$foo->off(Foo::EVENT_HELLO, $anonymousFunction);
```

Note que, em geral, você não deve tentar desvincular uma função anônima, a menos que você guarde em algum lugar quando ela for ligada ao evento. No exemplo acima, é assumido que a função anônima é armazenada em uma variável `$anonymousFunction`.

Para desvincular todos os manipuladores de um evento, simplesmente chame `yii\base\Component::off()` sem o segundo parâmetro:

```
$foo->off(Foo::EVENT_HELLO);
```

5.3.6 Manipuladores de Eventos de Classe

As subseções acima descreveram como anexar um manipulador para um evento a *nível de instância* (objeto). Às vezes, você pode querer responder a um evento acionado por *todas* as instâncias da classe em vez de apenas uma instância específica. Em vez de anexar um manipulador de evento em todas as instâncias, você pode anexar o manipulador a *nível da classe* chamando o método estático `yii\base\Event::on()`.

Por exemplo, um objeto [Active Record](#) irá disparar um evento `EVENT_AFTER_INSERT` sempre que inserir um novo registro no banco de dados. A fim de acompanhar as inserções feitas por *cada* objeto [Active Record](#), você pode usar o seguinte código:

```
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');
});
```

O manipulador de evento será invocado sempre que uma instância de `ActiveRecord`, ou uma de suas classes filhas, disparar o evento `EVENT_AFTER_INSERT`. No manipulador, você pode obter o objeto que disparou o evento através de `$event->sender`.

Quando um objecto dispara um evento, ele irá primeiro chamar manipuladores de nível de instância, seguido pelos manipuladores de nível de classe.

Você pode disparar um evento de *nível de classe* chamando o método estático `yii\base\Event::trigger()`. Um evento de nível de classe não está associado com um objeto particular. Como resultado, ele fará a chamada dos manipuladores de eventos apenas a nível da classe. Por exemplo:

```
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);
```

Note que, neste caso, `$event->sender` refere-se ao nome da classe acionando o evento em vez de uma instância do objeto.

Observação: Já que um manipulador de nível de classe vai responder a um evento acionado por qualquer instância dessa classe, ou qualquer classe filha, você deve usá-lo com cuidado, especialmente se a classe é uma classe base de baixo nível, tal como `yii\base\BaseObject`.

Para desvincular um manipulador de evento de nível de classe, chame `yii\base\Event::off()`. Por exemplo:

```
// desvincula $handler
Event::off(Foo::class, Foo::EVENT_HELLO, $handler);

// Desvincula todos os manipuladores de Foo::EVENT_HELLO
Event::off(Foo::class, Foo::EVENT_HELLO);
```

5.3.7 Eventos Globais

O Yii suporta o assim chamado *evento global*, que na verdade é um truque com base no mecanismo de eventos descrito acima. O evento global requer um *singleton* acessível globalmente, tal como a própria instância da aplicação.

Para criar o evento global, um evento *remetente* chama o método `singleton trigger()` para disparar o evento, em vez de chamar o método `trigger()` do *remetente*. Da mesma forma, os manipuladores de eventos são anexados ao evento no *singleton*. Por exemplo:

```
use Yii;
use yii\base\Event;
use app\components\Foo;

Yii::$app->on('bar', function ($event) {
    echo get_class($event->sender); // Mostra na tela "app\components\Foo"
});

Yii::$app->trigger('bar', new Event(['sender' => new Foo]));
```

A vantagem de usar eventos globais é que você não precisa de um objeto ao anexar um manipulador para o evento que será acionado pelo objeto. Em vez disso, a inclusão do manipulador e o evento acionado são ambos feitos através do *singleton*. (Por exemplo, uma instância da aplicação). Contudo, já que o namespace dos eventos globais é compartilhado com todos, você deve nomear os eventos globais sabiamente, tais como a introdução de algum tipo de namespace (por exemplo. “frontend.mail.sent”, “backend.mail.sent”).

5.4 Behaviors (Comportamentos)

Behaviors são instâncias de `yii\base\Behavior`, ou de uma classe-filha, também conhecido como mixins¹, permite melhorar a funcionalidade de uma classe **componente** existente sem a necessidade de mudar a herança dela. Anexar um behavior a um componente “introduz” os métodos e propriedades do behavior dentro do componente, tornando esses métodos e propriedades acessíveis como se estes fossem definidos na própria classe do componente. Além disso, um behavior pode responder a um **evento** disparado pelo componente, o que permite a customização do código normal.

5.4.1 Definindo Behaviors

Para definir um behavior, crie uma classe estendendo `yii\base\Behavior`, ou de uma classe-filha. Por exemplo:

```
namespace app\components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;
```

¹<https://en.wikipedia.org/wiki/Mixin>

```

private $_prop2;

public function getProp2()
{
    return $this->_prop2;
}

public function setProp2($value)
{
    $this->_prop2 = $value;
}

public function foo()
{
    // ...
}
}

```

O código acima define a classe behavior `app\components\MyBehavior`, com duas propriedades `_prop1` e `prop2` e um método `foo()`. Note que a propriedade `prop2` é definida através do método getter `getProp2()` e setter `setProp2()`. Isto é possível porque `yii\base\Behavior` estende de `yii\base\BaseObject` e portanto suporta definição de propriedades através de propriedades getters e setters.

Como essa classe é um behavior, quando ela está anexada a um componente, então este componente terá as propriedades `prop1` e `prop2` e o método `foo()`.

Dica: Em um behavior, você pode acessar o componente que o behavior está anexado através da propriedade `yii\base\Behavior::$owner`.

5.4.2 Manuseando Eventos de Componente

Se um behavior precisar responder a eventos disparados pelo componente ao qual está ligado, este deve sobrescrever o método `yii\base\Behavior::events()`. Por exemplo:

```

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)
    {
        // ...
    }
}

```

O método `events()` deve retornar uma lista de eventos e seus manipuladores correspondentes. O exemplo acima declara o evento `EVENT_BEFORE_VALIDATE` existente e define seu manipulador, `beforeValidate()`. Ao especificar um manipulador de evento, você pode utilizar um dos seguintes formatos:

- uma string que refere-se ao nome do método da classe behavior, como o exemplo acima
- um array com o nome do objeto ou classe, e um nome de método como string (sem parênteses), por exemplo, [`$object`, 'methodName'];
- uma função anônima

A assinatura de um manipulador de eventos deve ser como o exemplo abaixo, onde `$event` refere-se ao parâmetro do evento. Por favor, consulte a seção [Eventos](#) para mais detalhes sobre eventos.

```

function ($event) {
}

```

5.4.3 Anexando Behaviors (Comportamentos)

Você pode anexar um behavior a um componente de forma estática ou dinâmica. Na prática a forma estática é a mais comum.

Para anexar um behavior de forma estática, sobrescreva o método `behaviors()` da classe componente para o behavior que está sendo anexado. O método `behaviors()` deve retornar uma lista de [configurações](#) de behaviors. Cada configuração de behavior pode ser tanto um nome da classe behavior ou um array de configuração:

```

namespace app\models;

use yii\db\ActiveRecord;
use app\components\MyBehavior;

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // behavior anônimo, somente o nome da classe
            MyBehavior::class,
        ];
    }
}

```

```

// behavior nomeado, somente o nome da classe
'myBehavior2' => MyBehavior::class,

// behavior anônimo, array de configuração
[
    'class' => MyBehavior::class,
    'prop1' => 'value1',
    'prop2' => 'value2',
],

// behavior nomeado, array de configuração
'myBehavior4' => [
    'class' => MyBehavior::class,
    'prop1' => 'value1',
    'prop2' => 'value2',
]
];
}
}

```

Você pode associar um nome com um behavior especificando a chave do array correspondente à configuração do behavior. Neste caso o behavior é chamado *behavior nomeado*. No exemplo acima existem dois behaviors nomeados: `myBehavior2` e `myBehavior4`. Se um behavior não está associado a um nome, ele é chamado de *behavior anônimo*.

Para anexar um behavior dinamicamente, execute o método `yii\base\Component::attachBehavior()` do componente para o behavior que está sendo anexado:

```

use app\components\MyBehavior;

// anexando um objeto behavior
$component->attachBehavior('myBehavior1', new MyBehavior);

// anexando uma classe behavior
$component->attachBehavior('myBehavior2', MyBehavior::class);

// anexando através de um array de configuração
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::class,
    'prop1' => 'value1',
    'prop2' => 'value2',
]);

```

Você pode anexar vários behaviors de uma só vez usando o método `yii\base\Component::attachBehaviors()`:

```

$component->attachBehaviors([
    'myBehavior1' => new MyBehavior, // um behavior nomeado
    MyBehavior::class, // um behavior anônimo
]);

```

Você também pode anexar behaviors através de [configurações](#) conforme o exemplo a seguir:

```
[
  'as myBehavior2' => MyBehavior::class,

  'as myBehavior3' => [
    'class' => MyBehavior::class,
    'prop1' => 'value1',
    'prop2' => 'value2',
  ],
]
```

Para mais detalhes, por favor, consulte a seção [Configurações](#).

5.4.4 Usando Behaviors

Para usar um behavior, primeiro este deve ser anexado à um **componente** conforme as instruções mencionadas anteriormente. Uma vez que o behavior está anexado ao componente, seu uso é simples.

Você pode acessar uma variável *pública* ou uma [propriedade](#) definida por um getter e/ou um setter do behavior através do componente ao qual ele está anexado:

```
// "prop1" é uma propriedade definida na classe behavior
echo $component->prop1;
$component->prop1 = $value;
```

Você também pode executar um método *público* do behavior de forma parecida:

```
// foo() é um método público definido na classe behavior
$component->foo();
```

Como você pode ver, embora `$component` não defina `prop1` e nem `foo()`, eles podem ser utilizados como se eles fizessem parte da definição do componente, isto se deve ao behavior anexado.

Se dois behaviors definem a mesma propriedade ou método e ambos são anexados ao mesmo componente, o behavior que for anexado primeiramente ao componente terá precedência quando a propriedade ou método for acessada.

Um behavior pode estar associado a um nome quando ele for anexado a um componente. Sendo esse o caso, você pode acessar o objeto behavior usando o seu nome:

```
$behavior = $component->getBehavior('myBehavior');
```

Você também pode pegar todos os behaviors anexados a um componente:

```
$behaviors = $component->getBehaviors();
```

5.4.5 Desvinculando Behaviors (Comportamentos)

Para desvincular um behavior, execute `yii\base\Component::detachBehavior()` com o nome associado ao behavior:

```
$component->detachBehavior('myBehavior1');
```

Você também pode desvincular *todos* os behaviors:

```
$component->detachBehaviors();
```

5.4.6 Usando

Para encerrar, vamos dar uma olhada no `yii\behaviors\TimestampBehavior`. Este behavior suporta atualização automática dos atributos timestamp de um **Active Record** toda vez que o model (modelo) for salvo (por exemplo, na inserção ou na alteração).

Primeiro, anexe este behavior na classe **Active Record** que você planeja usar:

```
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'],
                ],
            ],
        ];
    }
}
```

A configuração do behavior acima especifica que o registro ao ser:

- inserido, o behavior deve atribuir o timestamp atual para os atributos `created_at` e `updated_at`
- atualizado, o behavior deve atribuir o timestamp atual para o atributo `updated_at`

Com esse código no lugar, se você tem um objeto `User` e tenta salvá-lo, você encontrará seus `created_at` e `updated_at` automaticamente preenchidos com a data e hora atual:

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
echo $user->created_at; // mostra a data atual
```

O `TimestampBehavior` também oferece um método útil `touch()`, que irá atribuir a data e hora atual para um atributo específico e o salva no banco de dados:

```
$user->touch('login_time');
```

5.4.7 Comparando Behaviors com Traits

Apesar de behaviors serem semelhantes a traits² em que ambos “injetam” suas propriedades e métodos para a classe principal, eles diferem em muitos aspectos. Tal como explicado abaixo, ambos têm prós e contras. Eles funcionam mais como complemento um do outro.

Razões para usar Behaviors

Classes Behavior, como classes normais, suportam herança. Traits, por outro lado, pode ser só suporta a programação “copia e cola”. Eles não suportam herança.

Behaviors podem ser anexados e desvinculados a um componente dinamicamente sem necessidade de modificação da classe componente. Para usar um trait, você deve modificar o código da classe.

Behaviors são configuráveis enquanto traits não são.

Behaviors podem customizar a execução do código do componente respondendo aos seus eventos. Quando houver nomes conflitantes entre diferentes behaviors anexados ao mesmo componente, o conflito é automaticamente resolvido priorizando o behavior anexado primeiramente ao componente. Nomes conflitantes causados por diferentes traits requer resolução manual renomeando as propriedades ou métodos afetados.

Razões para usar Traits

Traits são muito mais eficientes do que behaviors, estes são objetos e requerem mais tempo e memória.

IDEs são mais amigáveis com traits por serem nativos do PHP.

²https://www.php.net/manual/pt_BR/language.oop5.traits.php

5.5 Configurações

As configurações são amplamente utilizadas em Yii na criação de novos objetos ou inicializando objetos existentes. Configurações geralmente incluem o nome da classe do objeto que está sendo criado, e uma lista de valores iniciais que devem ser atribuídos as [propriedades](#) do objeto. Configurações também podem incluir uma lista de manipuladores que devem ser anexados aos [eventos](#) do objeto e/ou uma lista de [behaviors](#) que também deve ser ligado ao objeto.

A seguir, uma configuração é usada para criar e inicializar uma conexão com o banco:

```
$config = [  
    'class' => 'yii\db\Connection',  
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',  
    'username' => 'root',  
    'password' => '',  
    'charset' => 'utf8',  
];  
  
$db = Yii::createObject($config);
```

O método `Yii::createObject()` recebe um array de configuração como argumento, e cria um objeto instanciando a classe informada na configuração. Quando o objeto é instanciado, o resto da configuração será usada para inicializar as propriedades, manipuladores de eventos, e behaviors do objeto.

Se você já tem um objeto, você pode utilizar `Yii::configure()` para inicializar as propriedades do objeto com o array de configuração:

```
Yii::configure($object, $config);
```

Note que, neste caso, o array de configuração não deve conter o elemento `class`.

5.5.1 Formato da Configuração

O formato de uma configuração pode ser descrita formalmente como:

```
[  
    'class' => 'ClassName',  
    'propertyName' => 'propertyValue',  
    'on eventName' => $eventHandler,  
    'as behaviorName' => $behaviorConfig,  
]
```

Onde:

- O elemento `class` determina um nome de classe totalmente qualificado para o objeto que está sendo criado.

- O elemento `propertyName` determina os valores iniciais para a propriedade nomeada. As chaves são os nomes das propriedades e os valores são os valores iniciais correspondentes. Apenas variáveis públicas e [propriedades](#) definidas por getters/setters podem ser configuradas.
- O elemento `on eventName` determina quais manipuladores devem ser anexados aos [eventos](#) do objeto. Observe que as chaves do array são formadas prefixando a palavra `on` ao nome do evento. Por favor, consulte a seção [Eventos](#) para formatos de manipulador de eventos suportados.
- O elemento `as behaviorName` determina quais [behaviors](#) devem ser anexados ao objeto. Observe que as chaves do array são formadas prefixando a palavra `as` ao nome do behavior; O valor, `$behaviorConfig`, representa a configuração para a criação do behavior, como uma configuração normal descrita aqui.

Abaixo está um exemplo mostrando uma configuração com valores iniciais de propriedades, manipulador de evento e behaviors:

```
[
    'class' => 'app\components\SearchEngine',
    'apiKey' => 'xxxxxxx',
    'on search' => function ($event) {
        Yii::info("Keyword searched: " . $event->keyword);
    },
    'as indexer' => [
        'class' => 'app\components\IndexerBehavior',
        // ... property init values ...
    ],
]
```

5.5.2 Usando Configurações

Configurações são utilizadas em vários lugares no Yii. No início desta seção, mostramos como criar um objeto utilizando configuração `Yii::createObject()`. Nesta subseção, nós descreveremos a configuração de aplicação e configuração de widget - dois principais usos de configurações.

Configurações da Aplicação

A configuração de uma [aplicação](#) é provavelmente um dos mais complexos arrays no Yii. Isto porque a classe `application` tem muitas propriedades e eventos configuráveis. Mais importante, suas propriedades `components` podem receber um array de configuração para a criação de componentes que são registrados através da aplicação. O exemplo abaixo é um resumo do arquivo de configuração da aplicação para o [Template Básico de Projetos](#).

```
$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',
            ],
        ],
    ];

```

A configuração não tem uma chave `class`. Isto porque ele é utilizado como um [script de entrada](#), onde o nome da classe já está informado,

```
(new yii\web\Application($config))->run();
```

Mais detalhes sobre a configuração das propriedades `componentes` de uma aplicação podem ser encontrados na seção [Aplicações](#) e na seção [Service Locator](#).

Configurações de Widget

Ao utilizar os [widgets](#), muitas vezes você precisa usar configurações para customizar as propriedades do widget. Ambos os métodos `yii\base\Widget::widget()` e `yii\base\Widget::begin()` podem ser utilizados para criar um widget. Eles precisam de um array de configuração, como o exemplo abaixo,

```

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],
    ],
]);

```

```
    ],
  );
```

O código acima cria um widget `Menu` e inicializa suas propriedades `activateItems` com `false`. A propriedade `items` também é configurada com os itens do menu para serem exibidos.

Observe que, como o nome da classe já está dado, o array de configuração não precisa da chave `class`.

5.5.3 Arquivos de Configuração

Quando uma configuração é muito complexa, uma prática comum é armazená-la em um ou mais arquivos PHP, conhecidos como *arquivos de configuração*. Um arquivo de configuração retorna um array PHP representando a configuração. Por exemplo, você pode guardar uma configuração da aplicação em um arquivo chamado `web.php`, como a seguir,

```
return [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'components' => require __DIR__ . '/components.php',
];
```

Como a configuração de `components` é muito complexa, você o guarda em um arquivo separado chamado `components.php` e faz um “require” deste arquivo no `web.php` como mostrado acima. o conteúdo de `components.php` é como abaixo,

```
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',
    ],
];
```

Para pegar a configuração armazenada em um arquivo de configuração, simplesmente faça um “require” deste arquivo, como o exemplo abaixo:

```
$config = require 'path/to/web.php';  
(new yii\web\Application($config))->run();
```

5.5.4 Configurações Padrões

O método `Yii::createObject()` é implementado com base em um *container de injeção de dependência*. Ele permite que você especifique um conjunto do chamado *configurações padrões* que será aplicado a todas as instâncias das classes especificadas quando elas forem criadas usando `Yii::createObject()`. As configurações padrões podem ser especificadas executando `Yii::$container->set()` na *inicialização (bootstrapping)* do código.

Por exemplo, se você quiser personalizar `yii\widgets\LinkPager` de modo que todas as páginas mostrarão no máximo 5 botões (o valor padrão é 10), você pode utilizar o código abaixo para atingir esse objetivo,

```
\Yii::$container->set('yii\widgets\LinkPager', [  
    'maxButtonCount' => 5,  
]);
```

Sem usar as configurações padrão, você teria que configurar `maxButtonCount` em todos os lugares que utilizassem este recurso.

5.5.5 Constantes de Ambiente

Configurações frequentemente variam de acordo com o ambiente no qual a aplicação é executada. Por exemplo, no ambiente de desenvolvimento, você pode querer usar um banco de dados chamado `mydb_dev`, enquanto no servidor de produção você pode querer usar o banco de dados `mydb_prod`. Para facilitar a troca de ambientes, o Yii fornece uma constante chamada `YII_ENV` que você pode definir no *script de entrada* da sua aplicação. Por exemplo,

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

você pode definir `YII_ENV` como um dos seguintes valores:

- **prod**: ambiente de produção. A constante `YII_ENV_PROD` será avaliada como verdadeira. Este é o valor padrão da constante `YII_ENV` caso você não a defina.
- **dev**: ambiente de desenvolvimento. A constante `YII_ENV_DEV` será avaliada como verdadeira.
- **test**: ambiente de teste. A constante `YII_ENV_TEST` será avaliada como verdadeira.

Com estas constantes de ambientes, você pode especificar suas configurações de acordo com o ambiente atual. Por exemplo, sua configuração da aplicação pode conter o seguinte código para habilitar a barra de ferramentas de depuração e depurador no ambiente de desenvolvimento.

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

if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
}

return $config;
```

5.6 Aliases (Apelidos)

Aliases são usados para representar caminhos de arquivos ou URLs de forma que você não precise acoplar o código usando caminhos absolutos ou URLs em seu projeto. Um alias deve começar com o caractere @ para se diferenciar de um caminho de arquivo normal ou URL. O Yii já possui vários aliases predefinidos disponíveis. Por exemplo, o alias @yii representa o local em que o framework Yii foi instalado; @web representa a URL base para a aplicação que está sendo executada no momento.

5.6.1 Definindo Aliases

Você pode definir um alias para um caminho de arquivo ou URL chamando `Yii::setAlias()`:

```
// um alias de um caminho de arquivo
Yii::setAlias('@foo', '/caminho/para/foo');

// um alias de uma URL
Yii::setAlias('@bar', 'https://www.exemplo.com.br');
```

Observação: O caminho do arquivo ou URL sendo *apelidado* (aliased) *não* necessariamente refere-se a um arquivo ou a recursos existentes.

Dado um alias definido, você pode derivar um novo alias (sem a necessidade de chamar `Yii::setAlias()`) apenas acrescentando uma barra / seguido de um ou mais segmentos de caminhos de arquivos. Os aliases definidos através de `Yii::setAlias()` tornam-se o *alias raiz* (root alias), enquanto que aliases derivados dele, tornam-se *aliases derivados*. Por exemplo, @foo é um *alias raiz* (root alias), enquanto @foo/bar/arquivo.php é um alias derivado.

Você pode definir um alias usando outro alias (tanto raiz quanto derivado):

```
Yii::setAlias('@foobar', '@foo/bar');
```

Aliases raiz são normalmente definidos durante o estágio de *inicialização*. Por exemplo, você pode chamar `Yii::setAlias()` no *script de entrada*. Por conveniência, as *aplicações* definem uma propriedade `aliases` que você pode configurar na *configuração* da aplicação:

```
return [  
    // ...  
    'aliases' => [  
        '@foo' => '/caminho/para/foo',  
        '@bar' => 'https://www.exemplo.com.br',  
    ],  
];
```

5.6.2 Resolvendo Aliases

Você pode chamar `Yii::getAlias()` em um alias raiz para resolver o caminho de arquivo ou URL que ele representa. O mesmo método pode resolver também um alias derivado em seu caminho de arquivo ou URL correspondente.

```
echo Yii::getAlias('@foo');           // exibe: /caminho/para/foo  
echo Yii::getAlias('@bar');           // exibe: https://www.example.com  
echo Yii::getAlias('@foo/bar/arquivo.php'); // exibe:  
/caminho/para/foo/bar/arquivo.php
```

O caminho/URL representado por um alias derivado é determinado substituindo a parte do alias raiz com o seu caminho/URL correspondente.

Observação: O método `Yii::getAlias()` não checa se o caminho/URL resultante refere-se a um arquivo ou recursos existentes.

Um alias raiz pode também conter caracteres de barra /. O método `Yii::getAlias()` é inteligente o suficiente para descobrir que parte de um alias é um alias raiz e assim determina o caminho de arquivo ou URL correspondente:

```
Yii::setAlias('@foo', '/caminho/para/foo');  
Yii::setAlias('@foo/bar', '/caminho2/bar');  
Yii::getAlias('@foo/test/arquivo.php'); // exibe:  
/caminho/para/foo/test/arquivo.php  
Yii::getAlias('@foo/bar/arquivo.php');  // exibe:  
/caminho2/bar/arquivo.php
```

Se `@foo/bar` não estivesse definido como um alias raiz, a última chamada exibiria `/caminho/para/foo/bar/arquivo.php`.

5.6.3 Usando Aliases

Aliases são reconhecidos em muitos lugares no Yii sem a necessidade de chamar `Yii::getAlias()` para convertê-los em caminhos e URLs. Por exemplo, `yii\caching\FileCache::$cachePath` pode aceitar tanto um caminho de arquivo quanto um alias representando o caminho do arquivo, graças ao prefixo `@` que nos permite diferenciar um caminho de arquivo de um alias.

```
use yii\caching\FileCache;

$cache = new FileCache([
    'cachePath' => '@runtime/cache',
]);
```

Por favor, consulte a documentação da API para saber se o parâmetro de uma propriedade ou método suporta aliases.

5.6.4 Aliases Predefinidos

O Yii já predefine uma gama de aliases para referenciar facilmente caminhos de arquivos e URLs comumente usados:

- `@yii`, o diretório onde o arquivo `BaseYii.php` está localizado (também chamado de diretório do framework).
- `@app`, o caminho base da aplicação sendo executada no momento.
- `@runtime`, o caminho runtime da aplicação sendo executada no momento.
- `@webroot`, o diretório webroot da aplicação sendo executada no momento. Este é determinado baseado no diretório contendo o [script de entrada](#).
- `@web`, a URL base da aplicação sendo executada no momento. Esta tem o mesmo valor de `yii\web\Request::$baseUrl`.
- `@vendor`, o caminho da pasta vendor do Composer. Seu padrão é `@app/vendor`.
- `@bower`, o caminho raiz que contém os pacotes bower³. Seu padrão é `@vendor/bower`.
- `@npm`, o caminho raiz que contém pacotes npm⁴. Seu padrão é `@vendor/npm`.

O alias `@yii` é definido quando você inclui o arquivo `Yii.php` em seu [script de entrada](#). O resto dos aliases são definidos no construtor da aplicação ao aplicar a [configuração](#) da aplicação.

5.6.5 Aliases para Extensões

Um alias é automaticamente definido para cada [extensão](#) que for instalada através do Composer. Cada alias é nomeado a partir do namespace raiz da

³<https://bower.io/>

⁴<https://www.npmjs.com/>

extensão como declarada em seu arquivo `composer.json`, e cada alias representa o diretório raiz de seu pacote. Por exemplo, se você instalar a extensão `yiisoft/yii2-jui`, você terá automaticamente o alias `@yii/jui` definido durante o estágio de *inicialização*, equivalente a:

```
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

5.7 Autoloading de Classes

O Yii baseia-se no mecanismo de autoloading de classe⁵ para localizar e incluir todos os arquivos de classe necessários. Ele fornece um autoloader de alto desempenho que é compatível com o PSR-4 standard⁶. O autoloader é instalado quando o arquivo `Yii.php` é incluído. > Observação: Para simplificar a descrição, nesta seção, nós falaremos apenas sobre autoloading de classe. No entanto, tenha em mente que o conteúdo que estamos descrevendo aqui se aplica a autoloading de interfaces e traits também.

5.7.1 Usando o Autoloader do Yii

Para fazer uso da autoloader de classe do Yii, você deve seguir duas regras simples ao criar e nomear suas classes:

- Cada classe deve estar debaixo de um namespace⁷ (exemplo. `foo\bar\MyClass`)
- Cada classe deve ser salvo em um arquivo individual cujo caminho é determinado pelo seguinte algoritmo:

```
// $className é um nome de classe totalmente qualificado sem o primeiro  
barra invertida  
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) .  
' .php');
```

Por exemplo, se um nome de classe e seu namespace for `foo\bar\MyClass`, o alias correspondente ao caminho do arquivo da classe seria `@foo/bar/MyClass.php`. Para que este alias seja resolvido em um caminho de arquivo, de outra forma `@foo` ou `@foo/bar` deve ser um *alias raiz*.

Quando se utiliza o *Template Básico de Projetos*, você pode colocar suas classes sob o namespace de nível superior `app` de modo que eles podem ser carregados automaticamente pelo Yii sem a necessidade de definir um novo alias. Isto é porque `@app` é um *alias predefinido*, e um nome de classe como `app\components\MyClass` pode ser resolvido no arquivo de classe `AppBasePath/components/MyClass.php`, de acordo com o algoritmo já descrito.

⁵https://www.php.net/manual/pt_BR/language.oop5.autoload.php

⁶<https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md>

⁷https://www.php.net/manual/pt_BR/language.namespaces.php

No Template Avançado de Projetos⁸, cada camada tem sua própria alias raiz. Por exemplo, a camada front-end tem um alias raiz `@frontend`, enquanto a camada back-end o alias raiz é `@backend`. Como resultado, você pode colocar as classes do front-end debaixo do namespace `frontend` enquanto as classes do back-end estão debaixo do namespace `backend`. Isto permitirá que estas classes sejam carregadas automaticamente pelo Yii autoloader.

5.7.2 Mapeamento de Classes

O autoloader de classe do Yii suporta o recurso de *mapeamento de classe*, que mapeia nomes de classe para os caminhos de arquivo das classes correspondentes. Quando o autoloader está carregando uma classe, ele irá primeiro verificar se a classe se encontra no mapa. Se assim for, o caminho do arquivo correspondente será incluído diretamente, sem mais verificações. Isso faz com que classe seja carregada super rápido. De fato, todas as classes principais do Yii são carregadas automaticamente dessa maneira. Você pode adicionar uma classe no mapa de classe, armazenado em `Yii::$classMap`, usando:

```
Yii::$classMap['foo\bar\MyClass'] = 'path/to/MyClass.php';
```

Os *aliases* podem ser usados para especificar caminhos de arquivo de classe. Você deve definir o mapa de classe no processo de *inicialização (bootstrap)* de modo que o mapa está pronto antes de suas classes serem usadas.

5.7.3 Usando outros Autoloaders

Uma vez que o Yii utiliza o Composer como seu gerenciador de dependência de pacotes, é recomendado que você sempre instale o autoloader do Composer. Se você está utilizando bibliotecas de terceiros que tem seus próprios autoloaders, você também deve instalá-los. Ao usar o Yii autoloader junto com outros autoloaders, você deve incluir o arquivo `Yii.php` *depois* de todos os outros autoloaders serem instalados. Isso fará com que o Yii autoloader seja o primeiro a responder a qualquer solicitação de autoloading de classe. Por exemplo, o código a seguir foi extraído do *script de entrada* do *Template Básico de Projeto*. A primeira linha instala o Composer autoloader, enquanto a segunda linha instala o Yii autoloader:

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

Você pode usar o autoloader do Composer sozinho sem o autoloader do Yii. No entanto, ao fazê-lo, o desempenho do carregamento automático das classes pode ser degradada, e você deve seguir as regras estabelecidas pelo Composer para que suas classes sejam auto carregáveis.

⁸<https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-pt-BR/README.md>

Informação: Se você não quiser utilizar o autoloader do Yii, você deve criar sua própria versão do arquivo `Yii.php` e incluí-lo no seu [script de entrada](#).

5.7.4 Autoloading de Classes de Extensões

O autoloader do Yii é capaz de realizar autoloading de classes de [extensões](#). O único requisito é que a extensão especifique a seção `autoload` corretamente no seu arquivo `composer.json`. Por favor, consulte a documentação do Composer⁹ para mais detalhes sobre especificação de `autoload`.

No caso de você não usar o autoloader do Yii, o autoloader do Composer ainda pode realizar o `autoload` das classes de extensão para você.

5.8 Service Locator

Um service locator é um objeto que sabe como fornecer todos os tipos de serviços (ou componentes) que uma aplicação pode precisar. Num service locator, existe uma única instância de cada componente, exclusivamente identificados por um ID. Você usa o ID para recuperar um componente do service locator.

No Yii, um service locator é simplesmente uma instância da classe `yii\di\ServiceLocator` ou de classes que as estendam.

O service locator mais comumente utilizado no Yii é o objeto *application*, que pode ser acessado através de `\Yii::$app`. Os serviços que ele fornece são chamados de *componentes de aplicação*, tais como os componentes `request`, `response`, e `urlManager`. Você pode configurar esses componentes, ou mesmo substituí-los com suas próprias implementações, facilmente através de funcionalidades fornecidas pelo service locator.

Além do objeto *application*, cada objeto *module* também é um service locator. Para usar um service locator, o primeiro passo é registrar os componentes nele. Um componente pode ser registrado com `yii\di\ServiceLocator::set()`. O código abaixo mostra os diferentes modos de registrar um componente:

```
use yii\di\ServiceLocator;
use yii\caching\FileCache;

$locator = new ServiceLocator;

// registra "cache" utilizando um nome de classe que pode ser usado para
// criar um componente
$locator->set('cache', 'yii\caching\ApcCache');

// Registra "db" utilizando um array de configuração que pode ser usado para
// criar um componente
```

⁹<https://getcomposer.org/doc/04-schema.md#autoload>

```

$locator->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=demo',
    'username' => 'root',
    'password' => '',
]);

// registra "search" utilizando uma função anônima que cria um componente
$locator->set('search', function () {
    return new app\components\SolrService;
});

// registra "pageCache" utilizando um componente
$locator->set('pageCache', new FileCache);

```

Uma vez que um componente tenha sido registrado, você pode acessá-lo utilizando seu ID, em uma das duas maneiras abaixo:

```

$cache = $locator->get('cache');
// ou alternativamente
$cache = $locator->cache;

```

Como mostrado acima, `yii\di\ServiceLocator` permite-lhe acessar um componente como uma propriedade usando o ID do componente. Quando você acessa um componente pela primeira vez, `yii\di\ServiceLocator` usará as informações de registro do componente para criar uma nova instância do componente e retorná-lo. Mais tarde, se o componente for acessado novamente, o service locator irá retornar a mesma instância.

Você pode utilizar `yii\di\ServiceLocator::has()` para checar se um ID de componente já está registrado. Se você executar `yii\di\ServiceLocator::get()` com um ID inválido, uma exceção será lançada.

Uma vez que service locators geralmente são criados com [configurações](#), uma propriedade chamada `components` é fornecida. Isso permite que você possa configurar e registrar vários componentes de uma só vez. O código a seguir mostra um array de configuração que pode ser utilizado para configurar um service locator (por exemplo. uma [aplicação](#)) com o “db”, “cache” e “search” components:

```

return [
    // ...
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],
        'cache' => 'yii\caching\ApcCache',
        'search' => function () {
            $solr = new app\components\SolrService('127.0.0.1');

```

```

        // ... outras inicializações ...
        return $solr;
    },
],
];

```

No código acima, existe um caminho alternativo para configurar o componente “search”. Em vez de escrever diretamente um PHP callback que cria uma instância de `SolrService`, você pode usar um método estático de classe para retornar semelhante a um callback, como mostrado abaixo:

```

class SolrServiceBuilder
{
    public static function build($ip)
    {
        return function () use ($ip) {
            $solr = new app\components\SolrService($ip);
            // ... outras inicializações ...
            return $solr;
        };
    }
}

return [
    // ...
    'components' => [
        // ...
        'search' => SolrServiceBuilder::build('127.0.0.1'),
    ],
];

```

Esta abordagem alternativa é mais preferível quando você disponibiliza um componente Yii que encapsula alguma biblioteca de terceiros. Você usa o método estático como mostrado acima para representar a lógica complexa da construção do objeto de terceiros e o usuário do seu componente só precisa chamar o método estático para configurar o componente.

5.9 Container de Injeção de Dependência

Um container de injeção de dependência (DI) é um objeto que sabe como instanciar e configurar objetos e todas as suas dependências. O artigo do Martin¹⁰ explica bem porque o container de DI é útil. Aqui vamos explicar principalmente a utilização do container de DI fornecido pelo Yii.

5.9.1 Injeção de Dependência

O Yii fornece o recurso container de DI através da classe `yii\di\Container`. Ela suporta os seguintes tipos de injeção de dependência:

¹⁰<https://martinfowler.com/articles/injection.html>

- Injeção de Construtor;
- Injeção de setter e propriedade;
- Injeção de PHP callable.

Injeção de Construtor

O container de DI suporta injeção de construtor com o auxílio dos *type hints* identificados nos parâmetros dos construtores. Os type hints informam ao container quais classes ou interfaces são dependentes no momento da criação de um novo objeto. O container tentará pegar as instâncias das classes dependentes ou interfaces e depois injetá-las dentro do novo objeto através do construtor. Por exemplo:

```
class Foo
{
    public function __construct(Bar $bar)
    {
    }
}
$foo = $container->get('Foo');
// que equivale a:
$bar = new Bar;
$foo = new Foo($bar);
```

Injeção de Setter e Propriedade

A injeção de setter e propriedade é suportado através de *configurações*. Ao registrar uma dependência ou ao criar um novo objeto, você pode fornecer uma configuração que será utilizada pelo container para injetar as dependências através dos setters ou propriedades correspondentes. Por exemplo:

```
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'),
]);
```

Informação: O método `yii\di\Container::get()` recebe em seu terceiro parâmetro um array de configuração que deve ser aplicado ao objecto a ser criado. Se a classe implementa a interface `yii\base\Configurable` (por exemplo, `yii\base\BaseObject`), o array de configuração será passado como o último parâmetro para o construtor da classe; caso contrário, a configuração será aplicada *depois* que o objeto for criado.

Injeção de PHP Callable

Neste caso, o container usará um PHP callable registrado para criar novas instâncias da classe. Cada vez que `yii\di\Container::get()` for chamado, o callable correspondente será invocado. O callable é responsável por resolver as dependências e injetá-las de forma adequada para os objetos recém-criados. Por exemplo:

```
$container->set('Foo', function ($container, $params, $config) {
    $foo = new Foo(new Bar);
    // ... Outras inicializações...
    return $foo;
});

$foo = $container->get('Foo');
```

Para ocultar a lógica complexa da construção de um novo objeto você pode usar um método estático de classe para retornar o PHP callable. Por exemplo:

```
class FooBuilder
{
    public static function build($container, $params, $config)
    {
        return function () {
            $foo = new Foo(new Bar);
            // ... Outras inicializações...
            return $foo;
        };
    }
}

$container->set('Foo', FooBuilder::build());

$foo = $container->get('Foo');
```


Como você pode ver, o PHP callable é retornado pelo método `FooBuilder::build()`. Ao fazê-lo, quem precisar configurar a classe `Foo` não precisará saber como ele é construído.

5.9.2 Registrando Dependências

Você pode usar `yii\di\Container::set()` para registrar dependências. O registro requer um nome de dependência, bem como uma definição de dependência. Um nome de dependência pode ser um nome de classe, um nome de interface, ou um alias; e a definição de dependência pode ser um nome de classe, um array de configuração ou um PHP callable.

```
$container = new \yii\di\Container;

// registrar um nome de classe. Isso pode ser ignorado.
$container->set('yii\db\Connection');

// registrar uma interface
// Quando uma classe depende da interface, a classe correspondente
// será instanciada como o objeto dependente
$container->set('yii\mail\MailInterface', 'yii\swiftmailer\Mailer');

// registrar um alias. Você pode utilizar $container->get('foo')
// para criar uma instância de Connection
$container->set('foo', 'yii\db\Connection');

// registrar uma classe com configuração. A configuração
// será aplicada quando quando a classe for instanciada pelo get()
$container->set('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// registrar um alias com a configuração de classe
// neste caso, um elemento "class" é requerido para especificar a classe
$container->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// registrar um PHP callable
// O callable será executado sempre quando $container->get('db') for
// chamado
$container->set('db', function ($container, $params, $config) {
    return new \yii\db\Connection($config);
});
```

```
// registrar uma instância de componente
// $container->get('pageCache') retornará a mesma instância toda vez que for
// chamada
$container->set('pageCache', new FileCache);
```

Dica: Se um nome de dependência é o mesmo que a definição de dependência correspondente, você não precisa registrá-lo no container de DI.

Um registro de dependência através de `set()` irá gerar uma instância a cada vez que a dependência for necessária. Você pode usar `yii\di\Container::setSingleton()` para registrar a dependência de forma a gerar apenas uma única instância:

```
$container->setSingleton('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

5.9.3 Resolvendo Dependências

Depois de registrar as dependências, você pode usar o container de DI para criar novos objetos e o container resolverá automaticamente as dependências instanciando e as injetando dentro do novo objeto criado. A resolução de dependência é recursiva, isso significa que se uma dependência tem outras dependências, essas dependências também serão resolvidas automaticamente.

Você pode usar `yii\di\Container::get()` para criar novos objetos. O método recebe um nome de dependência, que pode ser um nome de classe, um nome de interface ou um alias. O nome da dependência pode ou não ser registrado através de `set()` ou `setSingleton()`. Você pode, opcionalmente, fornecer uma lista de parâmetros de construtor de classe e uma [configuração](#) para configurara o novo objeto criado. Por exemplo:

```
// "db" é um alias registrado previamente
$db = $container->get('db');

// equivale a: $engine = new \app\components\SearchEngine($apiKey, ['type'
=> 1]);
$engine = $container->get('app\components\SearchEngine', [$apiKey], ['type'
=> 1]);
```

Nos bastidores, o container de DI faz muito mais do que apenas a criação de um novo objeto. O container irá inspecionar primeiramente o construtor da classe para descobrir classes ou interfaces dependentes e automaticamente resolver estas dependências recursivamente. O código abaixo mostra um exemplo mais sofisticado. A classe `UserLister` depende de um objeto que

implementa a interface `UserFinderInterface`; A Classe `UserFinder` implementa esta interface e depende do objeto `Connection`. Todas estas dependências são declaradas através de type hint dos parâmetros do construtor da classe. Com o registro de dependência de propriedade, o container de DI é capaz de resolver estas dependências automaticamente e cria uma nova instância de `UserLister` simplesmente com `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');

// que é equivalente a:

$db = new \yii\db\Connection(['dsn' => '...']);
$finder = new UserFinder($db);
$lister = new UserLister($finder);
```

5.9.4 Uso Prático

O Yii cria um container de DI quando você inclui o arquivo `Yii.php` no **script de entrada** de sua aplicação. O container de DI é acessível através do `Yii::$container`. Quando você executa o método `Yii::createObject()`, na verdade o que será realmente executado é o método `get()` do container para criar um novo objeto. Conforme já informado acima, o container de DI resolverá automaticamente as dependências (se existir) e as injeta dentro do novo objeto criado. Como o Yii utiliza `Yii::createObject()` na maior parte do seu código principal para criar novos objetos, isso significa que você pode personalizar os objetos globalmente lidando com `Yii::$container`.

Por exemplo, você pode customizar globalmente o número padrão de botões de paginação do `yii\widgets\LinkPager`:

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

Agora, se você usar o widget na view (visão) com o seguinte código, a propriedade `maxButtonCount` será inicializado como 5 em lugar do valor padrão 10 como definido na class.

```
echo \yii\widgets\LinkPager::widget();
```

Todavia, você ainda pode substituir o valor definido através container de DI:

```
echo \yii\widgets\LinkPager::widget(['maxButtonCount' => 20]);
```

Outro exemplo é se beneficiar da injeção automática de construtor do container de DI. Assumindo que a sua classe controller (controlador) depende de alguns outros objetos, tais como um serviço de reserva de um hotel.

Você pode declarar a dependência através de um parâmetro de construtor e deixar o container DI resolver isto para você.

```
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);
}
}
```

Se você acessar este controller (controlador) a partir de um navegador, você vai ver um erro informando que `BookingInterface` não pode ser instanciado. Isso ocorre porque você precisa dizer ao container de DI como lidar com esta dependência:

```
\Yii::$container->set('app\components\BookingInterface',
'app\components\BookingService');
```

Agora se você acessar o controller (controlador) novamente, uma instância de `app\components\BookingService` será criada e injetada como o terceiro parâmetro do construtor do controller (controlador).

5.9.5 Quando Registrar Dependência

Em função de existirem dependências na criação de novos objetos, o seu registro deve ser feito o mais cedo possível. Seguem abaixo algumas práticas recomendadas:

- Se você é o desenvolvedor de uma aplicação, você pode registrar dependências no [script de entrada] (`structure-entry-scripts.md`) da sua aplicação ou em um script incluído no script de entrada.
- Se você é um desenvolvedor de [extensão](#), você pode registrar as dependências no bootstrapping (inicialização) da classe da sua extensão.

5.9.6 Resumo

Ambas as injeção de dependência e [service locator](#) são padrões de projetos conhecidos que permitem a construção de software com alta coesão e baixo acoplamento. É altamente recomendável que você leia o Artigo do Martin¹¹ para obter uma compreensão mais profunda da injeção de dependência e service locator.

O Yii implementa o [service locator](#) no topo da injeção dependência container (DI). Quando um service locator tenta criar uma nova instância de objeto, ele irá encaminhar a chamada para o container de DI. Este último vai resolver as dependências automaticamente tal como descrito acima.

¹¹<https://martinfowler.com/articles/injection.html>

Capítulo 6

Trabalhando com Banco de Dados

Error: not existing file: db-dao.md

6.1 Query Builder (Construtor de Consulta)

Desenvolvido à partir do [Database Access Objects](#), o query builder permite que você construa uma instrução SQL em um programático e independente de banco de dados. Comparado a escrever instruções SQL à mão, usar query builder lhe ajudará a escrever um código SQL relacional mais legível e gerar declarações SQL mais seguras.

Usar query builder geralmente envolve dois passos:

1. Criar um objeto `yii\db\Query` para representar diferentes partes de uma instrução SQL (ex. `SELECT`, `FROM`).
2. Executar um método (ex. `all()`) do objeto `yii\db\Query` para recuperar dados do banco de dados.

O código a seguir mostra uma forma habitual de utilizar query builder:

```
$rows = (new \yii\db\Query())
->select(['id', 'email'])
->from('user')
->where(['last_name' => 'Smith'])
->limit(10)
->all();
```

O código acima gera e executa a seguinte instrução SQL, onde o parâmetro `:last_name` está ligado a string `'Smith'`.

```
SELECT `id`, `email`
FROM `user`
WHERE `last_name` = :last_name
LIMIT 10
```

Observação: Geralmente, você trabalhará mais com o `yii\db\Query` do que com o `yii\db\QueryBuilder`. Este último é chamado pelo primeiro implicitamente quando você chama um dos métodos da query. O `yii\db\QueryBuilder` é a classe responsável por gerar instruções SGDBs dependentes (ex. colocar aspas em nomes de tabela/coluna) a partir de objetos de query independentemente do SGDB.

6.1.1 Construindo Queries

Para construir um objeto `yii\db\Query`, você pode chamar diferentes métodos de construção de query para especificar diferentes partes de uma instrução SQL. Os nomes destes métodos assemelha-se as palavras-chave SQL utilizados nas partes correspondentes da instrução SQL. Por exemplo, para especificar a parte da instrução SQL `FROM`, você deve chamar o método `from()`. Todos os métodos de construção de query retornam o próprio objeto query, que permite você encadear várias chamadas em conjunto. A seguir, descreveremos o uso de cada método de construção de query.

`select()`

O método `select()` especifica o fragmento de uma instrução SQL `SELECT`. Você pode especificar colunas para ser selecionado em um array ou uma string, como mostrado abaixo. Os nomes das colunas que estão sendo selecionadas serão automaticamente envolvidas entre aspas quando a instrução SQL está sendo gerada a partir do objeto `query`.

```
$query->select(['id', 'email']);
```

// equivalente a:

```
$query->select('id, email');
```

Os nomes das colunas que estão sendo selecionadas podem incluir prefixos de tabela e/ou aliases de colunas, como você faz ao escrever instruções SQL manualmente. Por exemplo:

```
$query->select(['user.id AS user_id', 'email']);
```

// é equivalente a:

```
$query->select('user.id AS user_id, email');
```

Se você estiver usando um array para especificar as colunas, você também pode usar as chaves do array para especificar os aliases das colunas. Por exemplo, o código acima pode ser reescrito da seguinte forma,

```
$query->select(['user_id' => 'user.id', 'email']);
```

Se você não chamar o método `select()` na criação da query, o `*` será selecionado, o que significa selecionar *todas* as colunas.

Além dos nomes de colunas, você também pode selecionar expressões DB. Você deve usar o formato array quando utilizar uma expressão DB que contenha vírgula para evitar que sejam gerados nomes de colunas de forma equivocada. Por exemplo:

```
$query->select(["CONCAT(first_name, ' ', last_name) AS full_name",  
'email']);
```

A partir da versão 2.0.1, você também pode selecionar sub-queries. Você deve especificar cada sub-query na forma de um objeto `yii\db\Query`. Por exemplo:

```
$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');
```

Para utilizar a cláusula `distinct`, você pode chamar `distinct()`, como a seguir:

```
// SELECT DISTINCT `user_id` ...  
$query->select('user_id')->distinct();
```

Você pode chamar `addSelect()` para selecionar colunas adicionais. Por exemplo:

```
$query->select(['id', 'username'])  
->addSelect(['email']);
```

`from()`

O método `from()` especifica o fragmento de uma instrução SQL `FROM`. Por exemplo:

```
// SELECT * FROM `user`  
$query->from('user');
```

Você pode especificar todas tabelas a serem selecionadas a partir de uma string ou um array. O nome da tabela pode conter prefixos de esquema e/ou aliases de tabela, da mesma forma quando você escreve instruções SQL manualmente. Por exemplo:

```
$query->from(['public.user u', 'public.post p']);
```

// é equivalente a:

```
$query->from('public.user u, public.post p');
```

Se você estiver usando o formato array, você também pode usar as chaves do array para especificar os aliases de tabelas, como mostrado a seguir:

```
$query->from(['u' => 'public.user', 'p' => 'public.post']);
```

Além de nome de tabelas, você também pode selecionar a partir de sub-queries especificando-o um objeto `yii\db\Query`. Por exemplo:

```
$subQuery = (new Query())->select('id')->from('user')->where('status=1');
```

```
// SELECT * FROM (SELECT `id` FROM `user` WHERE status=1) u  
$query->from(['u' => $subQuery]);
```

`where()`

O método `where()` especifica o fragmento de uma instrução SQL `WHERE`. Você pode usar um dos três formatos para especificar uma condição `WHERE`:

- formato string, ex., `'status=1'`
- formato hash, ex. `['status' => 1, 'type' => 2]`
- formato operador, ex. `['like', 'name', 'test']`

Formato String Formato de string é mais usado para especificar condições WHERE muito simples. Esta forma é muito semelhante a condições WHERE escritas manualmente. Por exemplo:

```
$query->where('status=1');

// ou usar parâmetro para vincular os valores dinamicamente
$query->where('status=:status', [':status' => $status]);
```

NÃO incorporar variáveis diretamente na condição como exemplificado abaixo, especialmente se os valores das variáveis vêm de entradas de dados dos usuários finais, porque isso vai fazer a sua aplicação ficar sujeita a ataques de injeção de SQL.

```
// Perigoso! NÃO faça isto a menos que você esteja muito certo que o $status
// deve ser um número inteiro.
$query->where("status=$status");
```

Ao usar parâmetro, você pode chamar `params()` ou `addParams()` para especificar os parâmetros separadamente.

```
$query->where('status=:status')
->addParams([':status' => $status]);
```

Formato Hash Formato HASH é mais usado para especificar múltiplos AND - sub-condições concatenadas, sendo cada uma afirmação simples de igualdade. É escrito como um array cujas chaves são nomes de coluna e os valores correspondem ao conteúdo destas colunas. Por exemplo:

```
// ...WHERE (`status` = 10) AND (`type` IS NULL) AND (`id` IN (4, 8, 15))
$query->where([
    'status' => 10,
    'type' => null,
    'id' => [4, 8, 15],
]);
```

Como você pode ver, o query builder é inteligente o suficiente para lidar corretamente com valores que são nulos ou arrays. Você também pode usar sub-queries com o formato hash conforme mostrado abaixo:

```
$userQuery = (new Query())->select('id')->from('user');

// ...WHERE `id` IN (SELECT `id` FROM `user`)
$query->where(['id' => $userQuery]);
```

Formato Operador Formato operador lhe permite especificar arbitrariamente condições de uma forma programática. Ele tem o seguinte formato:

```
[operator, operand1, operand2, ...]
```

onde cada um dos operandos pode ser especificado no formato string, formato hash ou formato operador recursivamente, enquanto o operador pode ser um dos seguintes procedimentos:

- **and**: os operandos devem ser concatenados juntos usando **AND**. Por exemplo, `['and', 'id=1', 'id=2']` irá gerar `id=1 AND id=2`. Se um operando for um array, ele será convertido para string usando as regras descritas aqui. Por exemplo, `['and', 'type=1', ['or', 'id=1', 'id=2']]` irá gerar `type=1 AND (id=1 OR id=2)`. O método **NÃO** vai fazer qualquer tratamento de escapar caracteres ou colocar aspas.
- **or**: similar ao operador **and** exceto pelo fato de que os operandos são concatenados usando **OR**.
- **between**: o operando 1 deve ser um nome de coluna, e os operandos 2 e 3 devem ser os valores de início e fim. Por exemplo, `['between', 'id', 1, 10]` irá gerar `id BETWEEN 1 AND 10`.
- **not between**: similar ao **between** exceto pelo fato de que **BETWEEN** é substituído por **NOT BETWEEN** na geração da condição.
- **in**: o operando 1 deve ser um nome de coluna ou uma expressão DB. O operando 2 pode ser tanto um array ou um objeto `Query`. Será gerado uma condição **IN**. Se o operando 2 for um array, representará o intervalo dos valores que a coluna ou expressão DB devem ser; se o operando 2 for um objeto `Query`, uma sub-query será gerada e usada como intervalo da coluna ou expressão DB. Por exemplo, `['in', 'id', [1, 2, 3]]` irá gerar `id IN (1, 2, 3)`. O método fará o tratamento apropriado de aspas e escape de valores para o intervalo. O operador **in** também suporta colunas compostas. Neste caso, o operando 1 deve ser um array de colunas, enquanto o operando 2 deve ser um array de arrays ou um objeto `Query` representando o intervalo das colunas.
- **not in**: similar ao operador **in** exceto pelo fato de que o **IN** é substituído por **NOT IN** na geração da condição.
- **like**: o operando 1 deve ser uma coluna ou uma expressão DB, e o operando 2 deve ser uma string ou um array representando o valor que a coluna ou expressão DB devem atender. Por exemplo, `['like', 'name', 'tester']` irá gerar `name LIKE '%tester%'`. Quando a faixa de valor é dado como um array, múltiplos predicados **LIKE** serão gerados e concatenadas utilizando **AND**. Por exemplo, `['like', 'name', ['test', 'sample']]` irá gerar `name LIKE '%test%' AND name LIKE '%sample%'`. Você também pode fornecer um terceiro operando opcional para especificar como escapar caracteres especiais nos valores. O operando deve ser um array de mapeamentos de caracteres especiais. Se este operando

não for fornecido, um mapeamento de escape padrão será usado. Você pode usar `false` ou um array vazio para indicar que os valores já estão escapados e nenhum escape deve ser aplicado. Note-se que ao usar um mapeamento de escape (ou o terceiro operando não é fornecido), os valores serão automaticamente fechado dentro de um par de caracteres percentuais.

> Observação: Ao utilizar o SGDB PostgreSQL você também pode usar `ilike`¹ > em vez de `like` para diferenciar maiúsculas de minúsculas.

- `or like`: similar ao operador `like` exceto pelo fato de que `OR` é usado para concatenar os predicados `LIKE` quando o operando 2 é um array.
- `not like`: similar ao operador `like` exceto pelo fato de que `LIKE` é substituído por `NOT LIKE`.
- `or not like`: similar ao operador `not like` exceto pelo fato de que `OR` é usado para concatenar os predicados `NOT LIKE`.
- `exists`: requer um operador que deve ser uma instância de `yii\db\Query` representando a sub-query. Isto criará uma expressão `EXISTS (sub-query)`.
- `not exists`: similar ao operador `exists` e cria uma expressão `NOT EXISTS (sub-query)`.
- `>`, `<=`, ou qualquer outro operador válido que leva dois operandos: o primeiro operando deve ser um nome de coluna enquanto o segundo um valor. Ex., `['>', 'age', 10]` vai gerar `age>10`.

Acrescentando Condições Você pode usar `andWhere()` ou `orWhere()` para acrescentar condições adicionais a uma condição já existente. Você pode chamá-los várias vezes para acrescentar várias condições separadamente. Por exemplo:

```
$status = 10;
$search = 'yii';

$query->where(['status' => $status]);

if (!empty($search)) {
    $query->andWhere(['like', 'title', $search]);
}
```

Se o `$search` não estiver vazio, a seguinte instrução SQL será gerada:

```
... WHERE (`status` = 10) AND (`title` LIKE '%yii%')
```

Filtrar Condições Ao construir condições `WHERE` a partir de entradas de usuários finais, você geralmente deseja ignorar os valores vazios. Por

¹<https://www.postgresql.org/docs/8.3/functions-matching.html#FUNCTIONS-LIKE>

exemplo, em um formulário de busca que lhe permite pesquisar por nome ou e-mail, você poderia ignorar as condições nome/e-mail se não houver entradas destes valores. Para atingir este objetivo utilize o método `filterWhere()`:

```
// $username and $email são inputs dos usuário finais
$query->filterWhere([
    'username' => $username,
    'email' => $email,
]);
```

A única diferença entre `filterWhere()` e `where()` é que o primeiro irá ignorar valores vazios fornecidos na condição no formato hash. Então se `$email` for vazio e `$username` não, o código acima resultará um SQL como: `...WHERE username=:username.`

Observação: Um valor é considerado vazio se ele for `null`, um array vazio, uma string vazia ou uma string que consiste em apenas espaços em branco. Assim como `andWhere()` e `orWhere()`, você pode usar `andFilterWhere()` e `orFilterWhere()` para inserir condições de filtro adicionais.

`orderBy()`

O método `orderBy()` especifica o fragmento de uma instrução SQL `ORDER BY`. Por exemplo:

```
// ... ORDER BY `id` ASC, `name` DESC
$query->orderBy([
    'id' => SORT_ASC,
    'name' => SORT_DESC,
]);
```

No código acima, as chaves do array são nomes de colunas e os valores são a direção da ordenação. A constante PHP `SORT_ASC` indica ordem crescente e `SORT_DESC` ordem decrescente. Se `ORDER BY` envolver apenas nomes simples de colunas, você pode especificá-lo usando string, da mesma forma como faria escrevendo SQL manualmente. Por exemplo:

```
$query->orderBy('id ASC, name DESC');
```

Observação: Você deve usar o formato array se `ORDER BY` envolver alguma expressão DB.

Você pode chamar `addOrderBy()` para incluir colunas adicionais para o fragmento `ORDER BY`. Por exemplo:

```
$query->orderBy('id ASC')
->addOrderBy('name DESC');
```

groupBy()

O método `groupBy()` especifica o fragmento de uma instrução SQL `GROUP BY`. Por exemplo:

```
// ... GROUP BY `id`, `status`  
$query->groupBy(['id', 'status']);
```

Se o `GROUP BY` envolver apenas nomes de colunas simples, você pode especificá-lo usando uma string, da mesma forma como faria escrevendo SQL manualmente. Por exemplo:

```
$query->groupBy('id, status');
```

Observação: Você deve usar o formato array se `GROUP BY` envolver alguma expressão DB.

Você pode chamar `addGroupBy()` para incluir colunas adicionais ao fragmento `GROUP BY`. Por exemplo:

```
$query->groupBy(['id', 'status'])  
->addGroupBy('age');
```

having()

O método `having()` especifica o fragmento de uma instrução SQL `HAVING`. Este método recebe uma condição que pode ser especificada da mesma forma como é feito para o `where()`. Por exemplo:

```
// ... HAVING `status` = 1  
$query->having(['status' => 1]);
```

Por favor, consulte a documentação do `where()` para mais detalhes de como especificar uma condição.

Você pode chamar `andHaving()` ou `orHaving()` para incluir uma condição adicional para o fragmento `HAVING`. Por exemplo:

```
// ... HAVING (`status` = 1) AND (`age` > 30)  
$query->having(['status' => 1])  
->andHaving(['>', 'age', 30]);
```

limit() e offset()

Os métodos `limit()` e `offset()` especificam os fragmentos de uma instrução SQL `LIMIT` e `OFFSET`. Por exemplo:

```
// ... LIMIT 10 OFFSET 20  
$query->limit(10)->offset(20);
```


Se você especificar um `limit` ou `offset` inválido (Ex. um valor negativo), ele será ignorado.

Observação: Para SGDBs que não suportam `LIMIT` e `OFFSET` (ex. MSSQL), query builder irá gerar uma instrução SQL que emula o comportamento `LIMIT/OFFSET`.

`join()`

O método `join()` especifica o fragmento de uma instrução SQL `JOIN`. Por exemplo:

```
// ... LEFT JOIN `post` ON `post`.`user_id` = `user`.`id`  
$query->join('LEFT JOIN', 'post', 'post.user_id = user.id');
```

O método `join()` recebe quatro parâmetros:

- `$type`: tipo do join, ex., 'INNER JOIN', 'LEFT JOIN'.
- `$table`: o nome da tabela a ser unida.
- `$on`: opcional, a condição do join, isto é, o fragmento `ON`. Por favor, consulte `where()` para detalhes sobre como especificar uma condição.
- `$params`: opcional, os parâmetros a serem vinculados à condição do join.

Você pode usar os seguintes métodos de atalho para especificar `INNER JOIN`, `LEFT JOIN` e `RIGHT JOIN`, respectivamente.

- `innerJoin()`
- `leftJoin()`
- `rightJoin()`

Por exemplo:

```
$query->leftJoin('post', 'post.user_id = user.id');
```

Para unir múltiplas tabelas, chame os métodos `join` acima multiplas vezes, uma para cada tabela. Além de unir tabelas, você também pode unir sub-queries. Para fazê-lo, especifique a sub-queries a ser unida como um objeto `yii\db\Query`. Por exemplo:

```
$subQuery = (new \yii\db\Query())->from('post');  
$query->leftJoin(['u' => $subQuery], 'u.id = author_id');
```

Neste caso, você deve colocar a sub-query em um array e usar as chaves do array para especificar o alias.

`union()`

O método `union()` especifica o fragmento de uma instrução SQL `UNION`. Por exemplo:

```

$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);

```

Você pode chamar `union()` múltiplas vezes para acrescentar mais fragmentos UNION.

6.1.2 Métodos Query

`yii\db\Query` fornece um conjunto de métodos para diferentes propósitos da consulta:

- `all()`: retorna um array de linhas sendo cada linha um array de pares nome-valor.
- `one()`: retorna a primeira linha do resultado.
- `column()`: retorna a primeira coluna do resultado.
- `scalar()`: retorna um valor escalar localizado na primeira linha e coluna do primeiro resultado.
- `exists()`: retorna um valor que indica se a consulta contém qualquer resultado.
- `count()`: retorna a quantidade de resultados da query.
- Outros métodos de agregação da query, incluindo `sum($q)`, `average($q)`, `max($q)`, `min($q)`. O parâmetro `$q` é obrigatório para estes métodos e pode ser um nome de uma coluna ou expressão DB. Por exemplo:

```

// 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();

```

Observação: O método `one()` retorna apenas a primeira linha do resultado da query. Ele não adiciona `LIMIT 1` para a geração da sentença SQL. Isso é bom e preferível se você souber que a query retornará apenas uma ou algumas linhas de dados (Ex. se você estiver consultando com algumas chaves primárias). Entretanto,

se a query pode retornar muitas linha de dados, você deve chamar `limit(1)` explicitamente para melhorar a performance. Ex., `(new \yii\db\Query()->from('user')->limit(1)->one()`.

Todos estes métodos query recebem um parâmetro opcional `$db` que representa a conexão do DB que deve ser usada para realizar uma consulta no DB. Se você omitir este parâmetro, o componente da aplicação `db` será usado como a conexão do DB. Abaixo está um outro exemplo do método `count()`:

```
// executes SQL: SELECT COUNT(*) FROM `user` WHERE `last_name`=:last_name
$count = (new \yii\db\Query())
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->count();
```

Quando você chamar um método de `yii\db\Query`, ele na verdade faz o seguinte trabalho por baixo dos panos:

- Chama `yii\db\QueryBuilder` para gerar uma instrução SQL com base na atual construção de `yii\db\Query`;
- Cria um objeto `yii\db\Command` com a instrução SQL gerada;
- Chama um método query (ex. `queryAll()`) do `yii\db\Command` para executar a instrução SQL e retornar os dados.

Algumas vezes, você pode querer examinar ou usar a instrução SQL construído a partir de um objeto `yii\db\Query`. Você pode atingir este objetivo com o seguinte código:

```
$command = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->createCommand();

// mostra a instrução SQL
echo $command->sql;

// Mostra os parâmetros que serão ligados
print_r($command->params);

// retorna todas as linhas do resultado da query
$rows = $command->queryAll();
```

Indexando os Resultados da Query

Quando você chama `all()`, será retornado um array de linhas que são indexadas por inteiros consecutivos. Algumas vezes você pode querer indexá-los de forma diferente, tal como indexar por uma coluna ou valor de expressão em particular. Você pode atingir este objetivo chamando `indexBy()` antes de `all()`. Por exemplo:

```
// retorna [100 => ['id' => 100, 'username' => '...', ...], 101 => [...],
103 => [...], ...]
$query = (new \yii\db\Query())
    ->from('user')
    ->limit(10)
    ->indexBy('id')
    ->all();
```

Para indexar através de valores de expressão, passe uma função anônima para o método `indexBy()`:

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy(function ($row) {
        return $row['id'] . $row['username'];
    })->all();
```

A função anônima recebe um parâmetro `$row` que contém os dados da linha atual e deve devolver um valor escalar que irá ser utilizada como índice para o valor da linha atual.

Batch Query (Consultas em Lote)

Ao trabalhar com grandes quantidades de dados, métodos tais como `yii\db\Query::all()` não são adequados porque eles exigem carregar todos os dados na memória. Para manter o requisito de memória baixa, Yii fornece o chamado suporte batch query. Um batch query faz uso do cursor de dados e obtém dados em lotes. Batch query pode ser usado como a seguir:

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
    ->orderBy('id');

foreach ($query->batch() as $users) {
    // $users é um array de 100 ou menos linha da tabela user
}

// ou se você quiser fazer uma iteração da linha uma por uma
foreach ($query->each() as $user) {
    // $user representa uma linha de dados a partir da tabela user
}
```

O método `yii\db\Query::batch()` and `yii\db\Query::each()` retorna um objeto `yii\db\BatchQueryResult` que implementa a interface `Iterator` e, assim, pode ser utilizado na construção do `foreach`. Durante a primeira iteração, uma consulta SQL é feita à base de dados. Os dados são, então, baixados em lotes nas iterações restantes. Por padrão, o tamanho do batch é 100, significando 100 linhas de dados que serão baixados a cada batch.

Você pode mudar o tamanho do batch passando o primeiro parâmetro para os métodos `batch()` ou `each()`.

Em comparação com o `yii\db\Query::all()`, o batch query somente carrega 100 linhas de dados na memória a cada vez. Se você processar os dados e, em seguida, descartá-lo imediatamente, o batch query pode ajudar a reduzir o uso de memória. Se você especificar o resultado da query a ser indexado por alguma coluna via `yii\db\Query::indexBy()`, o batch query ainda vai manter o índice adequado. Por exemplo:

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy('username');

foreach ($query->batch() as $users) {
    // $users é indexado pela coluna "username"
}

foreach ($query->each() as $username => $user) {
}
```

6.2 Active Record

O Active Record² fornece uma interface orientada a objetos para acessar e manipular dados armazenados em bancos de dados. Uma classe Active Record está associado a uma tabela da base de dados, uma instância do Active Record corresponde a uma linha desta tabela, e um *atributo* desta instância representa o valor de uma coluna desta linha. Em vez de escrever instruções SQL a mão, você pode acessar os atributos do Active Record e chamar os métodos do Active Record para acessar e manipular os dados armazenados nas tabelas do banco de dados.

Por exemplo, assumindo que `Customer` é uma classe Active Record que está associada com a tabela `customer` e `name` é uma coluna desta tabela. Você pode escrever o seguinte código para inserir uma nova linha na tabela `customer`:

```
$customer = new Customer();
$customer->name = 'Qiang';
$customer->save();
```

O código acima é equivalente a seguinte instrução SQL escrita à mão para MySQL, que é menos intuitiva, mais propenso a erros, e pode até ter problemas de compatibilidade se você estiver usando um tipo diferente de banco de dados:

```
$db->createCommand('INSERT INTO `customer` (`name`) VALUES (:name)', [
    ':name' => 'Qiang',
])->execute();
```

²https://pt.wikipedia.org/wiki/Active_record

O Yii fornece suporte Active Record para os seguintes bancos de dados relacionais:

- MySQL 4.1 ou superior: via `yii\db\ActiveRecord`
- PostgreSQL 8.4 ou superior: via `yii\db\ActiveRecord`
- SQLite 2 e 3: via `yii\db\ActiveRecord`
- Microsoft SQL Server 2008 ou superior: via `yii\db\ActiveRecord`
- Oracle: via `yii\db\ActiveRecord`
- CUBRID 9.3 ou superior: via `yii\db\ActiveRecord` (observe que devido a um bug³ na extensão PDO do cubrid, o tratamento de aspas não funciona, então você precisa do CUBRID 9.3 como o cliente, bem como servidor)
- Sphinx: via `yii\sphinx\ActiveRecord`, requer a extensão `yii2-sphinx`
- ElasticSearch: via `yii\elasticsearch\ActiveRecord`, requer a extensão `yii2-elasticsearch`. Adicionalmente, o Yii também suporta o uso de Active Record com os seguintes bancos de dados NoSQL:
- Redis 2.6.12 ou superior: via `yii\redis\ActiveRecord`, requer a extensão `yii2-redis`
- MongoDB 1.3.0 ou superior: via `yii\mongodb\ActiveRecord`, requer a extensão `yii2-mongodb`

Neste tutorial, vamos principalmente descrever o uso do Active Record para banco de dados relacionais. Todavia, maior parte do conteúdo descrito aqui também são aplicáveis a Active Record para bancos de dados NoSQL.

6.2.1 Declarando Classes Active Record

Para começar, declare uma classe Active Record estendendo `yii\db\ActiveRecord`. Porque cada classe Active Record é associada a uma tabela do banco de dados, nesta classe você deve sobrescrever o método `tableName()` para especificar a tabela que a classe está associada.

No exemplo abaixo, declaramos uma classe Active Record chamada `Customer` para a tabela do banco de dados `customer`.

```
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    const STATUS_INACTIVE = 0;
    const STATUS_ACTIVE = 1;

    /**
     * @return string the name of the table associated with this ActiveRecord
     * class.
     */
}
```

³<http://jira.cubrid.org/browse/API5-658>

```

    public static function tableName()
    {
        return 'customer';
    }
}

```

Instâncias de Active Record são consideradas como *models* (modelos). Por esta razão, geralmente colocamos as classes Active Record debaixo do namespace `app\models` (ou outros namespaces destinados a classes model).

Porque `yii\db\ActiveRecord` estende a partir de `yii\base\Model`, ele herda *todas* as características de *model*, tal como atributos, regras de validação, serialização de dados, etc.

6.2.2 Conectando ao Banco de Dados

Por padrão, o Active Record usa o *componente de aplicação* `db` com a DB *connection* para acessar e manipular os dados da base de dados. Como explicado em *Database Access Objects*, você pode configurar o componente `db` na configuração da aplicação como mostrado abaixo,

```

return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
        ],
    ],
];

```

Se você quiser usar uma conexão de banco de dados diferente do que o componente `db`, você deve sobrescrever o método `getDb()`:

```

class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        // use the "db2" application component
        return \Yii::$app->db2;
    }
}

```

6.2.3 Consultando Dados

Depois de declarar uma classe Active Record, você pode usá-lo para consultar dados da tabela de banco de dados correspondente. Este processo geralmente leva os seguintes três passos:

1. Crie um novo objeto query chamando o método `yii\db\ActiveRecord::find()`;
2. Construa o objeto query chamando os [métodos de query building](#);
3. Chame um [método de query](#) para recuperar dados em uma instância do Active Record.

Como você pode ver, isso é muito semelhante ao procedimento com [query builder](#). A única diferença é que em vez de usar o operador `new` para criar um objeto query, você chama `yii\db\ActiveRecord::find()` para retornar um novo objeto query que é da classe `yii\db\ActiveQuery`.

A seguir, estão alguns exemplos que mostram como usar Active Query para pesquisar dados:

```
// retorna um único customer cujo ID é 123
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::find()
    ->where(['id' => 123])
    ->one();

// retorna todos customers ativos e os ordena por seus IDs
// SELECT * FROM `customer` WHERE `status` = 1 ORDER BY `id`
$customers = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->orderBy('id')
    ->all();

// retorna a quantidade de customers ativos
// SELECT COUNT(*) FROM `customer` WHERE `status` = 1
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// retorna todos customers em um array indexado pelos seus IDs
// SELECT * FROM `customer`
$customers = Customer::find()
    ->indexBy('id')
    ->all();
```

No exemplo acima, `$customer` é um objeto `Customer` enquanto `$customers` é um array de objetos `Customer`. Todos são preenchidos com os dados recuperados da tabela `customer`.

Observação: Uma vez que o `yii\db\ActiveQuery` estende de `yii\db\Query`, você pode usar *todos* os métodos do query building e da query tal como descrito na seção [Query Builder](#).

Já que consultar por valores de chave primária ou um conjunto de valores de coluna é uma tarefa comum, o Yii fornece dois métodos de atalho para este propósito:

- `yii\db\ActiveRecord::findOne()`: retorna uma única instância de Active Record populado com a primeira linha do resultado da query.
- `yii\db\ActiveRecord::findAll()`: retorna um array de instâncias de Active Record populados com *todo* o resultado da query.

Ambos os métodos pode ter um dos seguintes formatos de parâmetro:

- Um valor escalar: o valor é tratado como uma chave primária que se deseja procurar. O Yii irá determinar automaticamente que coluna é a chave primária lendo o schema da base de dados.
- Um array de valores escalar: o array como uma chaves primárias que se deseja procurar.
- Um array associativo: as chaves são nomes de colunas e os valores são os valores correspondentes as colunas que se deseja procurar. Por favor, consulte o [Hash Format](#) para mais detalhes.

O código a seguir mostra como estes métodos podem ser usados:

```
// retorna um único customer cujo ID é 123
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// retorna customers cujo ID é 100, 101, 123 or 124
// SELECT * FROM `customer` WHERE `id` IN (100, 101, 123, 124)
$customers = Customer::findAll([100, 101, 123, 124]);

// retorna um customer ativo cujo ID é 123
// SELECT * FROM `customer` WHERE `id` = 123 AND `status` = 1
$customer = Customer::findOne([
    'id' => 123,
    'status' => Customer::STATUS_ACTIVE,
]);

// retorna todos os customers inativos
// SELECT * FROM `customer` WHERE `status` = 0
$customers = Customer::findAll([
    'status' => Customer::STATUS_INACTIVE,
]);
```

Observação: Nem o `yii\db\ActiveRecord::findOne()` ou o `yii\db\ActiveQuery::one()` irão adicionar `LIMIT 1` para a instrução SQL gerada. Se a sua query retornar muitas linhas de dados, você deve chamar `limit(1)` explicitamente para melhorar o desempenho, ex., `Customer::find()->limit(1)->one()`.

Além de usar os métodos do query building, você também pode escrever SQLs puros para pesquisar dados e preencher os objetos Active Record com o resultado. Você pode fazer isso chamando o método `yii\db\ActiveRecord::findBySql()`:

```
// retorna todos os customers inativos
$sql = 'SELECT * FROM customer WHERE status=:status';
```

```
$customers = Customer::findBySql($sql, ['status' =>
Customer::STATUS_INACTIVE])->all();
```

Não chame outros métodos do query building depois de chamar `findBySql()` pois eles serão ignorados.

6.2.4 Acessando Dados

Como já mencionado, os dados retornados da base de dados são populados em uma instância do Active Record, e cada linha do resultado da query corresponde a uma única instância do Active Record. Você pode acessar os valores das colunas acessando os atributos da instância do Active Record, Por exemplo,

```
// "id" and "email" são os nomes das colunas na tabela "customer"
$customer = Customer::findOne(123);
$id = $customer->id;
$email = $customer->email;
```

Observação: Os atributos Active Record são nomeados após a associação com as colunas da tabela de uma forma case-sensitive. O Yii automaticamente define um atributo no Active Record para todas as colunas da tabela associada. Você NÃO deve declarar novamente qualquer um dos atributos.

Uma vez que os atributos do Active Record são nomeados de acordo com as colunas das tabelas, você pode achar que está escrevendo código PHP como `$customer->first_name`, que usa sublinhados para separar palavras em nomes de atributos se as colunas da tabela forem nomeadas desta maneira. Se você está preocupado com um estilo de código consistente, você deve renomear suas colunas da tabela em conformidade (usar camelCase, por exemplo).

Transformação de Dados (Data Transformation)

Acontece frequentemente que os dados que estão sendo inseridos e/ou exibidos estão em um formato que é diferente do utilizado no momento da gravação na base de dados. Por exemplo, em uma base de dados você está gravando a data de aniversário do `customer` como UNIX timestamps (que não é muito amigável), embora, na maioria das vezes você gostaria de manipular aniversários como strings no formato de 'YYYY/MM/DD'. Para atingir este objetivo, você pode definir métodos de *transformação de dados* na classe Active Record `Customer` como a seguir:

```
class Customer extends ActiveRecord
{
    // ...
```

```
public function getBirthdayText()
{
    return date('Y/m/d', $this->birthday);
}

public function setBirthdayText($value)
{
    $this->birthday = strtotime($value);
}
}
```

Agora no seu código PHP, em vez de acessar `$customer->birthday`, você acessaria `$customer->birthdayText`, que lhe permitirá inserir e exibir data de aniversário dos `customers` no formato `'YYYY/MM/DD'`.

Dica: O exemplo acima mostra uma forma genérica de transformação de dados em diferentes formatos. Se você estiver trabalhando com valores de data, você pode usar o [DateValidator](#) e o `yii\jui\DatePicker`, que é mais fácil e mais poderoso.

Recuperando Dados em Arrays

Embora a recuperação de dados através de objetos Active Record seja conveniente e flexível, pode não ser a melhor opção caso você tenha que retornar uma grande quantidade de dados devido ao grande consumo de memória. Neste caso, você pode recuperar usando arrays do PHP chamando `asArray()` antes de executar um método query:

```
// retorna todos os `customers`
// cada `customer` retornado é associado a um array
$customers = Customer::find()
    ->asArray()
    ->all();
```

Observação: Enquanto este método economiza memória e melhora o desempenho, ele é muito próximo a camada de abstração do DB e você vai perder a maioria dos recursos do Active Record. Uma distinção muito importante reside no tipo dos valores de coluna de dados. Quando você retorna dados em uma instância de Active Record, valores de colunas serão automaticamente convertidos de acordo com os tipos de coluna reais; de outra forma quando você retorna dados em arrays, valores de colunas serão strings (uma vez que são o resultado do PDO sem nenhum processamento), independentemente seus tipos de coluna reais.

Recuperando Dados em Lote

No [Query Builder](#), explicamos que você pode usar *batch query* para minimizar o uso de memória quando pesquisar uma grande quantidade de dados do

banco de dados. Você pode utilizar a mesma técnica no Active Record. Por exemplo,

```
// descarrega 10 `customers` a cada vez
foreach (Customer::find()->batch(10) as $customers) {
    // $customers é um array de 10 ou menos objetos Customer
}

// descarrega 10 `customers` por vez e faz a iteração deles um por um
foreach (Customer::find()->each(10) as $customer) {
    // $customer é um objeto Customer
}

// batch query com carga antecipada
foreach (Customer::find()->with('orders')->each() as $customer) {
    // $customer é um objeto Customer
}
```

6.2.5 Salvando Dados

Usando Active Record, você pode facilmente salvar dados em um banco de dados realizando as seguintes etapas:

1. Preparar uma instância de Active Record
2. Atribuir novos valores aos atributos do Active Record
3. Chamar o método `yii\db\ActiveRecord::save()` para salvar as informações no banco de dados.

Por exemplo,

```
// insere uma nova linha de dados
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();

// atualiza uma linha de dados existente
$customer = Customer::findOne(123);
$customer->email = 'james@newexample.com';
$customer->save();
```

O método `save()` pode tanto inserir ou atualizar dados, dependendo do estado da instância do Active Record. Se a instância tiver sido recém criada através do operador `new`, ao chamar `save()` será realizado a inserção de uma nova linha; se a instância for o resultado de um método da query, ao chamar `save()` será realizado a atualização dos dados associados a instância.

Você pode diferenciar os dois estados de uma instância de Active Record verificando o valor da sua propriedade `isNewRecord`. Esta propriedade também é usada internamente pelo `save()` como mostrado abaixo:

```
public function save($runValidation = true, $attributeNames = null)
{
    if ($this->getIsNewRecord()) {
        return $this->insert($runValidation, $attributeNames);
    } else {
        return $this->update($runValidation, $attributeNames) !== false;
    }
}
```

Dica: Você pode chamar `insert()` ou `update()` diretamente para inserir ou atualizar dados.

Validação de Dados

Já que o `yii\db\ActiveRecord` estende de `yii\base\Model`, ele compartilha os mesmos recursos de [validação de dados](#). Você pode declarar regras de validação sobrescrevendo o método `rules()` e realizar a validação de dados chamando o método `validate()`.

Quando você chama `save()`, por padrão chamará `validate()` automaticamente. somente quando a validação passa, os dados são de fato salvos; do contrário, simplesmente retorna falso, e você pode verificar a propriedade `errors` para recuperar a mensagem de erro de validação.

Dica: Se você tiver certeza que os seus dados não precisam de validação (ex., os dados tem uma origem confiável), você pode chamar `save(false)` para pular a validação.

Atribuição Maciça

Como um [models](#) normal, instância de Active Record também oferece o [recurso de atribuição maciça](#). Usando este recurso, você pode atribuir valores para vários atributos de uma instância de Active Record em uma única declaração PHP, como mostrado a seguir. Lembre-se que somente [atributos de segurança](#) pode ser massivamente atribuídos.

```
$values = [
    'name' => 'James',
    'email' => 'james@example.com',
];

$customer = new Customer();

$customer->attributes = $values;
$customer->save();
```

Atualizando Contadores

Isto é uma tarefa comum para incrementar ou decrementar uma coluna em uma tabela do banco de dados. Chamamos essas colunas como colunas de

contador. Você pode usar `updateCounters()` para atualizar uma ou mais colunas de contadores. Por exemplo,

```
$post = Post::findOne(100);  
  
// UPDATE `post` SET `view_count` = `view_count` + 1 WHERE `id` = 100  
$post->updateCounters(['view_count' => 1]);
```

Observação: Se você usar `yii\db\ActiveRecord::save()` para atualizar uma coluna de contador, você pode acabar com um resultado impreciso, porque é provável que o mesmo contador esteja sendo salvo por várias solicitações que lêem e escrevem o mesmo valor do contador.

Atributos Sujos

Quando você chama `save()` para salvar uma instância de Active Record, somente *atributos sujos* serão salvos. Um atributo é considerado *sujo* se o seu valor foi modificado desde que foi carregado a partir de DB ou salvos em DB, mais recentemente. Note que a validação de dados será realizada, independentemente se a instância do Active Record tiver ou não atributos sujos.

O Active Record mantém automaticamente a lista de atributos sujos. Isto é feito mantendo uma versão antiga dos valores de atributos e comparando-as com as últimas informações. Você pode chamar `yii\db\ActiveRecord::getDirtyAttributes()` para pegar os atributos sujos correntes. Você também pode chamar `yii\db\ActiveRecord::markAttributeDirty()` para marcar explicitamente um atributo como sujo.

Se você estiver interessado nos valores de atributos antes da sua modificação mais recente, você pode chamar `getOldAttributes()` ou `getOldAttribute()`.

Observação: A comparação dos valores antigos e novos será feito usando o operador `===` portanto, um valor será considerado sujo mesmo se ele tiver o mesmo valor, mas um tipo diferente. Isto é comum quando o modelo recebe a entrada de dados do usuário a partir de um formulário HTML onde todos os valores são representados como string. Para garantir o tipo correto, por exemplo, valores inteiros você pode aplicar um filtro de validação: `['attributeName', 'filter', 'filter' => 'intval']`.

Valores Padrões de Atributos

Algumas de suas colunas de tabelas podem ter valores padrões definidos em um banco de dados. Algumas vezes, você pode querer preencher previamente o formulário Web para uma instância de Active Record com os seus valores padrões. Para evitar escrever os mesmos valores padrão novamente,

you can call `loadDefaultValues()` to populate the default values defined by the DB for the attributes corresponding to the Active Record:

```
$customer = new Customer();
$customer->loadDefaultValues();
// $customer->xyz será atribuído o valor padrão declarado na definição da
coluna "xyz"
```

Atualizando Múltiplas Linhas

The methods described above do all the work in a single instance of Active Record, causing insertion or update of lines of the table individually. To update multiple lines individually, you must call the static method `updateAll()`.

```
// UPDATE `customer` SET `status` = 1 WHERE `email` LIKE `%@example.com%`
Customer::updateAll(['status' => Customer::STATUS_ACTIVE], ['like', 'email',
'@example.com']);
```

In the same way you can call `updateAllCounters()` to update columns of counter of several lines at the same time.

```
// UPDATE `customer` SET `age` = `age` + 1
Customer::updateAllCounters(['age' => 1]);
```

6.2.6 Deletando Dados

To delete a single line of data, first recover the instance of Active Record corresponding to the line and then call the method `yii\db\ActiveRecord::delete()`.

```
$customer = Customer::findOne(123);
$customer->delete();
```

You can call `yii\db\ActiveRecord::deleteAll()` to delete multiple or all lines of data. For example,

```
Customer::deleteAll(['status' => Customer::STATUS_INACTIVE]);
```

Observação: Tenha muito cuidado quando chamar `deleteAll()` porque pode apagar todos os dados de sua tabela se você cometer um erro na especificação da condição.

6.2.7 Ciclo de Vida de um Active Record

It is important to understand the life cycle of an Active Record when it is used for different purposes. During each life cycle, a determined sequence of methods will be invoked, and you can substitute these methods to have the chance to customize the life cycle. You can also

responder certos eventos disparados pelo Active Record durante o ciclo de vida para injetar o seu código personalizado. Estes eventos são especialmente úteis quando você está desenvolvendo **behaviors** que precisam personalizar o ciclo de vida do Active Record.

A seguir, vamos resumir diversos ciclos de vida de um Active Record e os métodos/eventos que fazem parte do ciclo de vida.

Ciclo de Vida de uma Nova Instância

Quando se cria uma nova instância de Active Record através do operador `new`, acontece o seguinte ciclo de vida:

1. Construção da classe;
2. `init()`: dispara um evento `EVENT_INIT`.

Ciclo de Vida de uma Pesquisa de Dados

Quando se pesquisa dados através de um dos métodos de consulta, cada novo Active Record populado sofrerá o seguinte ciclo de vida:

1. Contrução da classe.
2. `init()`: dispara um evento `EVENT_INIT`.
3. `afterFind()`: dispara um evento `EVENT_AFTER_FIND`.

Ciclo de Vida da Persistência de Dados

Quando se chama `save()` para inserir ou atualizar uma instância de Active Record, acontece o seguinte ciclo de vida:

1. `beforeValidate()`: dispara um evento `EVENT_BEFORE_VALIDATE`. se o método retornar falso ou `yii\base\ModelEvent::$isValid` for falso, o restante das etapas serão ignoradas.
2. Executa a validação de dados. Se a validação de dados falhar, Os passos após o passo 3 serão ignorados.
3. `afterValidate()`: dispara um evento `EVENT_AFTER_VALIDATE`.
4. `beforeSave()`: dispara um evento `EVENT_BEFORE_INSERT` ou evento `EVENT_BEFORE_UPDATE`. Se o método retornar falso ou `yii\base\ModelEvent::$isValid` for falso, o restante dos passos serão ignorados.
5. Realiza a atual inserção ou atualização de dados;
6. `afterSave()`: dispara um evento `EVENT_AFTER_INSERT` ou evento `EVENT_AFTER_UPDATE`.

Ciclo de Vida da Deleção de Dados

Quando se chama `delete()` para deletar uma instância de Active Record, acontece o seguinte ciclo de vida:

1. `beforeDelete()`: dispara um evento `EVENT_BEFORE_DELETE`. Se o método retornar falso ou `yii\base\ModelEvent::$isValid` for falso, o restante dos passos serão ignorados.
2. Executa a atual deleção de dados.
3. `afterDelete()`: dispara um evento `EVENT_AFTER_DELETE`.

Observação: Chamar qualquer um dos seguintes métodos não iniciará qualquer um dos ciclos de vida listados acima:

- `yii\db\ActiveRecord::updateAll()`
- `yii\db\ActiveRecord::deleteAll()`
- `yii\db\ActiveRecord::updateCounters()`
- `yii\db\ActiveRecord::updateAllCounters()`

6.2.8 Trabalhando com Transações

Existem duas formas de usar transações quando se trabalha com Active Record. A primeira maneira é anexar explicitamente chamadas de método Active Record em um bloco transacional, como mostrado abaixo,

```
$customer = Customer::findOne(123);

Customer::getDb()->transaction(function($db) use ($customer) {
    $customer->id = 200;
    $customer->save();
    // ...outras operações DB...
});

// ou como alternativa

$transaction = Customer::getDb()->beginTransaction();
try {
    $customer->id = 200;
    $customer->save();
    // ...outras operações DB...
    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
}
```

A segunda maneira é listar as operações de banco de dados que exigem suporte transacional no método `yii\db\ActiveRecord::transactions()`. Por exemplo,

```

class Customer extends ActiveRecord
{
    public function transactions()
    {
        return [
            'admin' => self::OP_INSERT,
            'api' => self::OP_INSERT | self::OP_UPDATE | self::OP_DELETE,
            // o código acima é equivalente ao código abaixo:
            // 'api' => self::OP_ALL,
        ];
    }
}

```

O método `yii\db\ActiveRecord::transactions()` deve retornar um array cujas chaves são cenários nomes e valores das operações correspondentes que devem ser embutidas dentro de transações. Você deve usar as seguintes constantes para fazer referência a diferentes operações de banco de dados:

- `OP_INSERT`: operação de inserção realizada pelo `insert()`;
- `OP_UPDATE`: operação de atualização realizada pelo `update()`;
- `OP_DELETE`: operação de deleção realizada pelo `delete()`.

Utilize o operador `|` concatenar as constantes acima para indicar várias operações. Você também pode usar a constante de atalho `OP_ALL` para referenciar todas as três constantes acima.

6.2.9 Bloqueios Otimistas

O bloqueio otimista é uma forma de evitar conflitos que podem ocorrer quando uma única linha de dados está sendo atualizado por vários usuários. Por exemplo, se ambos os usuário A e B estiverem editando o mesmo artigo de Wiki ao mesmo tempo. Após o usuário A salvar suas alterações, o usuário B pressiona o botão “Save” em uma tentativa de salvar suas edições também. Uma vez que o usuário B está trabalhando em uma versão desatualizada do artigo, seria desejável ter uma maneira de impedi-lo de salvar o artigo e mostrar-lhe alguma mensagem.

Bloqueio otimista resolve o problema acima usando uma coluna para gravar o número de versão de cada registro. Quando um registro está sendo salvo com um número de versão desatualizado, uma exceção `yii\db\StaleObjectException` será lançada, que impede que o registro seja salvo. Bloqueio otimista só é suportado quando você atualiza ou exclui um registro de dados existente usando `yii\db\ActiveRecord::update()` ou `yii\db\ActiveRecord::delete()` respectivamente.

Para usar bloqueio otimista:

1. Crie uma coluna na tabela do banco de dados associada com a classe Active Record para armazenar o número da versão de cada registro. A coluna deve ser do tipo biginteger (no MySQL ela deve ser `BIGINT DEFAULT 0`).

2. Sobrescreva o método `yii\db\ActiveRecord::optimisticLock()` para retornar o nome desta coluna.
3. No formulário Web que recebe as entradas dos usuários, adicione um campo hidden para armazenar o número da versão corrente do registro que está sendo atualizado. Certifique-se que seu atributo versão possui regras de validação de entrada validadas com êxito.
4. Na ação do controller que atualiza o registro usando Active Record, mostre uma estrutura *try catch* da `yii\db\StaleObjectException` exceção. Implementar lógica de negócios necessária (ex. mesclar as mudanças, mostrar dados obsoletos) para resolver o conflito.

Por exemplo, digamos que a coluna de versão se chama `version`. Você pode implementar bloqueio otimista conforme o código abaixo.

```
// ----- view code -----

use yii\helpers\Html;

// ...Outros campos de entrada de dados
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) {
        // lógica para resolver o conflito
    }
}
```

6.2.10 Trabalhando com Dados Relacionais

Além de trabalhar com tabelas individuais, o Active Record também é capaz de trazer dados de várias tabelas relacionadas, tornando-os prontamente acessíveis através dos dados primários. Por exemplo, a tabela de clientes está relacionada a tabela de pedidos porque um cliente pode ter múltiplos

pedidos. Com uma declaração adequada desta relação, você pode ser capaz de acessar informações do pedido de um cliente utilizando a expressão `$customer->orders` que devolve informações sobre o pedido do cliente em um array de instâncias de Active Record de `Pedidos`.

Declarando Relações

Para trabalhar com dados relacionais usando Active Record, você primeiro precisa declarar as relações nas classes Active Record. A tarefa é tão simples como declarar um *método de relação* para cada relação desejada. Segue exemplos:

```
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']);
    }
}
```

No código acima, nós declaramos uma relação `orders` para a classe `Customer`, e uma relação `customer` para a classe `Order`.

Cada método de relação deve ser nomeado como `getXyz`. Nós chamamos de `xyz` (a primeira letra é em letras minúsculas) o *nome da relação*. Note que os nomes de relações são *case sensitive*.

Ao declarar uma relação, você deve especificar as seguintes informações:

- A multiplicidade da relação: especificada chamando tanto o método `hasMany()` quanto o método `hasOne()`. No exemplo acima você pode facilmente ler nas declarações de relação que um `customer` tem vários `orders` enquanto uma `order` só tem um `customer`.
- O nome da classe Active Record relacionada: especificada no primeiro parâmetro dos métodos `hasMany()` e `hasOne()`. Uma prática recomendada é chamar `Xyz::class` para obter o nome da classe para que você possa receber suporte do preenchimento automático de IDEs bem como detecção de erros.
- A ligação entre os dois tipos de dados: especifica a(s) coluna(s) por meio do qual os dois tipos de dados se relacionam. Os valores do array são as colunas da tabela primária (representada pela classe Active Record que você declarou as relações), enquanto as chaves do array são as colunas da tabela relacionada.

Uma regra fácil de lembrar é, como você pode ver no exemplo acima, você escreve a coluna que pertence ao Active Record relacionado diretamente ao lado dele. Você pode ver que `customer_id` é uma propriedade de `Order` e `id` é uma propriedade de `Customer`.

Acessando Dados Relacionais

Após declarar as relações, você pode acessar os dados relacionados através dos nomes das relações. Isto é como acessar uma [propriedade](#) de um objeto definida por um método de relação. Por esta razão, nós podemos chamá-la de *propriedade de relação*. Por exemplo,

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
// $orders is an array of Order objects
$orders = $customer->orders;
```

Observação: quando você declara uma relação chamada `xyz` através de um método getter `getXyz()`, você terá acesso a `xyz` como uma [propriedade de objeto](#). Note que o nome é case-sensitive.

Se a relação for declarada com `hasMany()`, acessar esta propriedade irá retornar um array de instâncias de Active Record relacionais; Se a relação for declarada com `hasOne()`, acessar esta propriedade irá retornar a instância de Active Record relacional ou `null` se não encontrar dados relacionais.

Quando você acessa uma propriedade de relação pela primeira vez, uma instrução SQL será executada, como mostrado no exemplo acima. Se a mesma propriedade for acessada novamente, o resultado anterior será devolvido sem executar novamente a instrução SQL. Para forçar a execução da instrução SQL, você deve primeiramente remover a configuração da propriedade de relação: `unset($customer->orders)`.

Observação: Embora este conceito é semelhante ao recurso de [propriedade de objeto](#), existe uma diferença importante. Em uma propriedade de objeto normal o valor da propriedade é do mesmo tipo que o método getter definindo. Já o método de relação retorna uma instância de `yii\db\ActiveQuery`, embora o acesso a uma propriedade de relação retorne uma instância de `yii\db\ActiveRecord` ou um array deste tipo.

```
$customer->orders; // é um array de objetos `Order`
$customer->getOrders(); // retorna uma instância de ActiveQuery
```

Isso é útil para a criação de consultas personalizadas, que está descrito na próxima seção.

Consulta Relacional Dinâmica

Uma vez que um método de relação retorna uma instância de `yii\db\ActiveQuery`, você pode construir uma query usando os métodos de `query building` antes de executá-la. Por exemplo,

```
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `subtotal` > 200 ORDER BY `id`
$orders = $customer->getOrders()
    ->where(['>', 'subtotal', 200])
    ->orderBy('id')
    ->all();
```

Diferente de acessar uma propriedade de relação, cada vez que você executar uma consulta relacional dinâmica através de um método de relação, uma instrução SQL será executada, mesmo que a mesma consulta relacional dinâmica tenha sido executada anteriormente.

Algumas vezes você pode querer parametrizar uma relação para que possa executar mais facilmente uma consulta relacional dinâmica. Por exemplo, você pode declarar uma relação `bigOrders` da seguinte forma,

```
class Customer extends ActiveRecord
{
    public function getBigOrders($threshold = 100)
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id'])
            ->where('subtotal > :threshold', [':threshold' => $threshold])
            ->orderBy('id');
    }
}
```

Em seguida, você será capaz de executar as seguintes consultas relacionais:

```
// SELECT * FROM `order` WHERE `subtotal` > 200 ORDER BY `id`
$orders = $customer->getBigOrders(200)->all();

// SELECT * FROM `order` WHERE `subtotal` > 100 ORDER BY `id`
$orders = $customer->bigOrders;
```

Relações Através de Tabela de Junção

Em uma modelagem de banco de dados, quando a multiplicidade entre duas tabelas relacionadas é `many-to-many`, geralmente é criada uma tabela de junção⁴. Por exemplo, a tabela `order` e a tabela `item` podem se relacionar através da tabela de junção chamada `order_item`. Um `order`, então, corresponderá a múltiplos `order items`, enquanto um `product item` também corresponderá a múltiplos `order items`.

⁴https://pt.wikipedia.org/wiki/Entidade_associativa

Ao declarar tais relações, você chamaria `via()` ou `viaTable()` para especificar a tabela de junção. A diferença entre `via()` e `viaTable()` é que o primeiro especifica a tabela de junção em função a uma relação existente enquanto o último faz referência diretamente a tabela de junção. Por exemplo,

```
class Order extends ActiveRecord
{
  public function getItems()
  {
    return $this->hasMany(Item::class, ['id' => 'item_id'])
      ->viaTable('order_item', ['order_id' => 'id']);
  }
}
```

ou alternativamente,

```
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');
  }
}
```

A utilização das relações declaradas com uma tabela de junção é a mesma que a das relações normais. Por exemplo,

```
// 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 (...)
// returns an array of Item objects
$items = $order->items;
```

Lazy e Eager Loading

Na seção *Acessando Dados Relacionais*, nós explicamos que você pode acessar uma propriedade de relação de uma instância de Active Record como se acessa uma propriedade de objeto normal. Uma instrução SQL será executada somente quando você acessar a propriedade de relação pela primeira vez. Chamamos esta forma de acessar estes dados de *lazy loading*. Por exemplo,

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$orders = $customer->orders;

// nenhum SQL é executado
$orders2 = $customer->orders;
```

O uso de lazy loading é muito conveniente. Entretanto, pode haver um problema de desempenho caso você precise acessar a mesma propriedade de relação a partir de múltiplas instâncias de Active Record. Considere o exemplo a seguir. Quantas instruções SQL serão executadas?

```
// SELECT * FROM `customer` LIMIT 100
$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {
    // SELECT * FROM `order` WHERE `customer_id` = ...
    $orders = $customer->orders;
}
```

Como você pode ver no comentário do código acima, há 101 instruções SQL sendo executadas! Isto porque cada vez que você acessar a propriedade de relação `orders` com um objeto `Customer` diferente no bloco `foreach`, uma instrução SQL será executada.

Para resolver este problema de performance, você pode usar o *eager loading*, como mostrado abaixo,

```
// SELECT * FROM `customer` LIMIT 100;
// SELECT * FROM `orders` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->with('orders')
    ->limit(100)
    ->all();

foreach ($customers as $customer) {
    // nenhum SQL é executado
    $orders = $customer->orders;
}
```

Ao chamar `yii\db\ActiveQuery::with()`, você instrui ao Active Record trazer os `orders` dos primeiros 100 `customers` em uma única instrução SQL. Como resultado, você reduz o número de instruções SQL executadas de 101 para 2.

Você pode utilizar esta abordagem com uma ou várias relações. Você pode até mesmo utilizar *eager loading* com *relações aninhadas*. Uma relação aninhada é uma relação que é declarada dentro de uma classe Active Record de relação. Por exemplo, `Customer` se relaciona com `Order` através da relação

`orders`, e `Order` está relacionado com `Item` através da relação `items`. Quando pesquisar por `Customer`, você pode fazer o *eager loading* de `items` usando a notação de relação aninhada `orders.items`.

O código a seguir mostra diferentes formas de utilizar `with()`. Assumimos que a classe `Customer` tem duas relações `orders` e `country`, enquanto a classe `Order` tem uma relação `items`.

```
// carga antecipada de "orders" e "country"
$customers = Customer::find()->with('orders', 'country')->all();
// equivalente à sintaxe de array abaixo
$customers = Customer::find()->with(['orders', 'country'])->all();
// nenhum SQL executado
$orders= $customers[0]->orders;
// nenhum SQL executado
$country = $customers[0]->country;

// carga antecipada de "orders" e a relação aninhada "orders.items"
$customers = Customer::find()->with('orders.items')->all();
// acessa os "items" do primeiro "order" e do primeiro "customer"
// nenhum SQL executado
$items = $customers[0]->orders[0]->items;
```

Você pode carregar antecipadamente relações profundamente aninhadas, tais como `a.b.c.d`. Todas as relações serão carregadas antecipadamente. Isto é, quando você chamar `with()` usando `a.b.c.d`, você fará uma carga antecipada de `a`, `a.b`, `a.b.c` e `a.b.c.d`.

Observação: Em geral, ao fazer uma carga antecipada de N relações em que M relações são definidas com uma tabela de junção, um total de instruções SQL $N+M+1$ serão executadas. Note que uma relação aninhada `a.b.c.d` conta como 4 relações.

Ao carregar antecipadamente uma relação, você pode personalizar a consulta relacional correspondente utilizando uma função anônima. Por exemplo,

```
// procura "customers" trazendo junto "country" e "orders" ativas
// 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();
```

Ao personalizar a consulta relacional para uma relação, você deve especificar o nome da relação como uma chave de array e usar uma função anônima como o valor de array correspondente. A função anônima receberá um parâmetro

`$query` que representa o objeto `yii\db\ActiveQuery` utilizado para realizar a consulta relacional para a relação. No exemplo acima, modificamos a consulta relacional, acrescentando uma condição adicional sobre o status de ‘orders’.

Observação: Se você chamar `select()` ao carregar antecipadamente as relações, você tem que certificar-se de que as colunas referenciadas na relação estão sendo selecionadas. Caso contrário, os modelos relacionados podem não ser carregados corretamente. Por exemplo,

```
$orders = Order::find()->select(['id',
'amount'])->with('customer')->all();
// $orders[0]->customer é sempre nulo. Para corrigir o problema, você
deve fazer o seguinte:
$orders = Order::find()->select(['id', 'amount',
'customer_id'])->with('customer')->all();
```

Relações utilizando JOIN

Observação: O conteúdo descrito nesta subseção é aplicável apenas aos bancos de dados relacionais, tais como MySQL, PostgreSQL, etc.

As consultas relacionais que temos descrito até agora só fizeram referência as chave primária ao consultar os dados. Na realidade, muitas vezes precisamos referenciar colunas nas tabelas relacionadas. Por exemplo, podemos querer trazer os ‘customers’ que tenham no mínimo um ‘order’ ativo. Para solucionar este problema, podemos fazer uma query com join como a seguir:

```
// 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();
```

Observação: É importante evitar ambiguidade de nomes de colunas ao criar queries com JOIN. Uma prática comum é prefixar os nomes das colunas com os nomes de tabela correspondente. Entretanto, uma melhor abordagem é a de explorar as declarações de relação existente chamando `yii\db\ActiveQuery::joinWith()`:

```
$customers = Customer::find()
->joinWith('orders')
->where(['order.status' => Order::STATUS_ACTIVE])
->all();
```

Ambas as formas executam o mesmo conjunto de instruções SQL. Embora a última abordagem seja mais elegante.

Por padrão, `joinWith()` usará `LEFT JOIN` para juntar a tabela primária com a tabela relacionada. Você pode especificar um tipo de join diferente (ex. `RIGHT JOIN`) através do seu terceiro parâmetro `$joinType`. Se o tipo de join que você quer for `INNER JOIN`, você pode apenas chamar `innerJoinWith()`.

Chamando `joinWith()` os dados relacionais serão carregados antecipadamente por padrão. Se você não quiser trazer o dados relacionados, você pode especificar o segundo parâmetro `$eagerLoading` como falso.

Assim como `with()`, você pode juntar uma ou várias relações; você pode personalizar as queries relacionais dinamicamente; você pode utilizar join com relações aninhadas; e pode misturar o uso de `with()` e `joinWith()`. Por exemplo,

```
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->andWhere(['>', 'subtotal', 100]);
    },
])->with('country')
->all();
```

Algumas vezes ao juntar duas tabelas, você pode precisar especificar alguma condição extra na estrutura SQL `ON` do `JOIN`. Isto pode ser feito chamando o método `yii\db\ActiveQuery::onCondition()` como a seguir:

```
// 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();
```

A query acima retorna *todos* customers, e para cada customer retorna todos orders ativos. Note que isto difere do nosso primeiro exemplo onde retornávamos apenas customers que tinham no mínimo um order ativo.

Observação: Quando `yii\db\ActiveQuery` é especificada com uma condição usando `onCondition()`, a condição será colocada no `ON` se a query tiver um `JOIN`. Se a query não utilizar `JOIN`, a condição será colocada no `WHERE` da query.

Relações Inversas

Declarações de relação são geralmente recíprocas entre duas classes de Active Record. Por exemplo, `Customer` se relaciona com `Order` através da relação `orders`, e `Order` se relaciona com `Customer` através da relação `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']);
    }
}
```

Agora considere a seguinte parte do código:

```
// 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';
```

Podemos pensar que `$customer` e `$customer2` são a mesma coisa, mas não são! Na verdade eles contêm a mesma informação de `customer`, mas são objetos diferentes. Ao acessar `$order->customer`, uma instrução SQL extra é executada para popular um novo objeto `$customer2`.

Para evitar esta redundância de execução de SQL no exemplo acima, devemos dizer para o Yii que `customer` é uma *relação inversa* de `orders` chamando o método `inverseOf()` como mostrado abaixo:

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

Com esta modificação na declaração de relação, podemos ter:

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// Nenhum SQL será executado
$customer2 = $order->customer;

// exibe "same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

Observação: relações inversas não podem ser definidas por relações que envolvam uma tabela de junção. Isto é, se uma relação for definida com `via()` ou `viaTable()`, você não deve chamar `inverseOf()`.

6.2.11 Salvando Relações

Ao trabalhar com dados relacionais, muitas vezes você precisa estabelecer relações entre dados diferentes ou destruir relações existentes. Isso requer a definição de valores apropriados para as colunas que definem as relações. Usando Active Record, você pode acabar escrevendo o seguinte código:

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

// definindo o atributo que define a relação "customer" em "Order"
$order->customer_id = $customer->id;
$order->save();
```

Active Record fornece o método `link()` que lhe permite realizar essa tarefa de uma forma mais elegante:

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

$order->link('customer', $customer);
```

O método `link()` requer que você especifique o nome da relação e a instância de Active Record alvo cuja relação deve ser estabelecida. O método irá modificar os valores dos atributos que apontam para duas instâncias de Active Record e salvá-los no banco de dados. No exemplo acima, o atributo `customer_id` da instância `Order` será definido para ser o valor do atributo `id` da instância `Customer` e depois salvá-lo no banco de dados.

Observação: Você não pode conectar duas instâncias de Active Record recém-criadas.

Os benefícios de usar `link()` é ainda mais óbvio quando uma relação é definida por meio de uma tabela de junção. Por exemplo, você pode usar o código a seguir para ligar uma instância de `Order` com uma instância de `Item`:

```
$order->link('items', $item);
```

O código acima irá inserir automaticamente uma linha na tabela de junção `order_item` para relacionar a `order` com o `item`.

Observação: O método `link()` não realizará nenhuma validação de dados ao salvar a instância de Active Record afetada. É de sua responsabilidade validar todos os dados de entrada antes de chamar esse método.

A operação inversa para `link()` é `unlink()` que quebra uma relação existente entre duas instâncias de Active Record. Por exemplo,

```
$customer = Customer::find()->with('orders')->where(['id' => 123])->one();
$customer->unlink('orders', $customer->orders[0]);
```

Por padrão, o método `unlink()` irá definir o(s) valor(es) da(s) chave(s) estrangeira(s) que especificam a relação existente para `null`. Você pode, entretanto, optar por excluir a linha da tabela que contém a chave estrangeira passando o parâmetro `$delete` como `true` para o método.

Quando uma tabela de junção está envolvida numa relação, chamar o método `unlink()` fará com que as chaves estrangeiras na tabela de junção sejam apagadas, ou a deleção da linha correspondente na tabela de junção se `$delete` for `true`.

6.2.12 Relações Entre Banco de Dados Diferentes

O Active Record lhe permite declarar relações entre classes Active Record que são alimentados por diferentes bancos de dados. As bases de dados podem ser de diferentes tipos (ex. MySQL e PostgreSQL, ou MS SQL e MongoDB) e podem rodar em diferentes servidores. Você pode usar a mesma sintaxe para executar consultas relacionais. Por exemplo,

```
// Customer está associado a tabela "customer" em um banco de dados
relacional (ex. MySQL)
class Customer extends \yii\db\ActiveRecord
{
    public static function tableName()
    {
        return 'customer';
    }
}
```

```

    }

    public function getComments()
    {
        // a customer tem muitos comments
        return $this->hasMany(Comment::class, ['customer_id' => 'id']);
    }
}

// Comment está associado com a coleção "comment" em um banco de dados
MongoDB
class Comment extends \yii\mongodb\ActiveRecord
{
    public static function collectionName()
    {
        return 'comment';
    }

    public function getCustomer()
    {
        // um comment tem um customer
        return $this->hasOne(Customer::class, ['id' => 'customer_id']);
    }
}

$customers = Customer::find()->with('comments')->all();

```

Você pode usar a maioria dos recursos de consulta relacional que foram descritos nesta seção.

Observação: O uso de `joinWith()` está limitado às bases de dados que permitem JOIN entre diferentes bancos de dados (cross-database). Por esta razão, você não pode usar este método no exemplo acima porque MongoDB não suporta JOIN.

6.2.13 Personalizando Classes de Consulta

Por padrão, todas as consultas do Active Record são suportadas por `yii\db\ActiveQuery`. Para usar uma classe de consulta customizada em uma classe Active Record, você deve sobrescrever o método `yii\db\ActiveRecord::find()` e retornar uma instância da sua classe de consulta customizada. Por exemplo,

```

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());
    }
}

class CommentQuery extends ActiveRecord
{
    // ...
}

```

Agora sempre que você realizar uma consulta (ex. `find()`, `findOne()`) ou definir uma relação (ex. `hasOne()`) com `Comment`, você estará trabalhando com a instância de `CommentQuery` em vez de `ActiveQuery`.

Dica: Em grandes projetos, é recomendável que você use classes de consulta personalizadas para manter a maioria dos códigos de consultas relacionadas de modo que a classe `Active Record` possa se manter limpa.

Você pode personalizar uma classe de consulta de várias formas criativas afim de melhorar a sua experiência na construção de consultas. Por exemplo, você pode definir um novo método `query building` em uma classe de consulta customizada:

```

class CommentQuery extends ActiveRecord
{
    public function active($state = true)
    {
        return $this->andWhere(['active' => $state]);
    }
}

```

Observação: Em vez de chamar `where()`, você geralmente deve chamar `andWhere()` ou `orWhere()` para anexar condições adicionais ao definir um novo método de `query building` de modo que as condições existentes não serão sobrescritas.

Isto permite que você escreva códigos de `query building` conforme o exemplo abaixo:

```

$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();

```

Você também pode usar um novo método de `query building` ao definir relações sobre `Comment` ou executar uma consulta relacional:

```

class Customer extends \yii\db\ActiveRecord
{
    public function getActiveComments()

```



```

    {
        return $this->hasMany(Comment::class, ['customer_id' =>
            'id'])->active();
    }
}

$customers = Customer::find()->with('activeComments')->all();

// ou alternativamente

$customers = Customer::find()->with([
    'comments' => function($q) {
        $q->active();
    }
])->all();

```

Observação: No Yii 1.1, existe um conceito chamado *scope*. Scope já não é suportado no Yii 2.0, e você deve usar classes de consulta personalizadas e métodos de consulta para atingir o mesmo objetivo.

6.2.14 Selecionando Campos Extras

Quando uma instância de Active Record é populada através do resultado de uma consulta, seus atributos são preenchidos pelos valores correspondentes das colunas do conjunto de dados recebidos.

Você é capaz de buscar colunas ou valores adicionais da consulta e armazená-lo dentro do Active Record. Por exemplo, suponha que tenhamos uma tabela chamada ‘room’, que contém informações sobre quartos disponíveis em um hotel. Cada ‘room’ grava informações sobre seu tamanho geométrico usando os campos ‘length’, ‘width’, ‘height’. Imagine que precisemos de uma lista de todos os quartos disponíveis com o seu volume em ordem decrescente. Então você não pode calcular o volume usando PHP, porque precisamos ordenar os registros pelo seu valor, mas você também quer que o ‘volume’ seja mostrado na lista. Para alcançar este objetivo, você precisa declarar um campo extra na sua classe Active Record ‘Room’, que vai armazenar o valor do campo ‘volume’:

```

class Room extends \yii\db\ActiveRecord
{
    public $volume;

    // ...
}

```

Então você precisa criar uma consulta, que calcule o volume da sala e realize a ordenação:

```

$rooms = Room::find()
    ->select([

```

```

        '{{room}}.*', // select all columns
        '([[length]] * [[width]].* [[height]]) AS volume', // calculate a
        volume
    ])
    ->orderBy('volume DESC') // apply sort
    ->all();

foreach ($rooms as $room) {
    echo $room->volume; // contains value calculated by SQL
}

```

Capacidade de selecionar campos extras pode ser extremamente útil para consultas de agregação. Suponha que você precise exibir uma lista de clientes com a contagem de seus pedidos. Em primeiro lugar, você precisa declarar uma classe `Customer` com uma relação com ‘orders’ e um campo extra para armazenamento da contagem:

```

class Customer extends \yii\db\ActiveRecord
{
    public $ordersCount;

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}

```

Então você pode criar uma consulta que faça um JOIN com ‘orders’ e calcule a quantidade:

```

$customers = Customer::find()
    ->select([
        '{{customer}}.*', // select all customer fields
        'COUNT('{{order}}.id) AS ordersCount' // calculate orders count
    ])
    ->joinWith('orders') // ensure table junction
    ->groupBy('{{customer}}.id') // group the result to ensure aggregation
    function works
    ->all();

```

6.3 Migrações de Dados (Migrations)

Durante o curso de desenvolvimento e manutenção de uma aplicação orientada a banco de dados, a estrutura de banco sendo usada evolui ao mesmo tempo em que o código. Por exemplo, durante o desenvolvimento de uma aplicação, a criação de uma nova tabela pode ser necessária; após ser feito o deploy da aplicação em produção, pode ser descoberto que um índice deveria

ser criado para melhorar a performance de alguma query; entre outros. Como a mudança de uma estrutura de banco de dados normalmente necessita de alguma mudança no código, o Yii suporta a então chamada funcionalidade de *migração de dados* que permite que você mantenha um registro das mudanças feitas no banco de dados em termos de *migrações de dados* que são versionadas em conjunto com o código fonte da aplicação.

Os seguintes passos mostram como uma migração de dados pode ser usada pela equipe durante o desenvolvimento:

1. João cria uma nova migração (ex. cria uma nova tabela, muda a definição de uma coluna, etc.).
2. João comita a nova migração no sistema de controle de versão (ex. Git, Mercurial).
3. Pedro atualiza seu repositório a partir do sistema de controle de versão e recebe a nova migração.
4. Pedro aplica a nova migração ao seu banco de dados local em seu ambiente de desenvolvimento, e assim, sincronizando seu banco de dados para refletir as mudanças que João fez.

E os seguintes passos mostram como fazer o deploy para produção de uma nova versão:

1. Luiz cria uma nova tag para o repositório do projeto que contem algumas novas migrações de dados.
2. Luiz atualiza o código fonte no servidor em produção para a tag criada.
3. Luiz aplica todas as migrações de dados acumuladas para o banco de dados em produção.

O Yii oferece um conjunto de ferramentas de linha de comando que permitem que você:

- crie novas migrações;
- aplique migrações;
- reverta migrações;
- reaplique migrações;
- exiba um histórico das migrações;

Todas estas ferramentas são acessíveis através do comando `yii migrate`. Nesta seção nós iremos descrever em detalhes como realizar várias tarefas usando estas ferramentas. Você também pode descobrir como usar cada ferramenta através do comando de ajuda `yii help migrate`.

Observação: os migrations (migrações) podem afetar não só o esquema do banco de dados, mas também ajustar os dados existentes para se conformar ao novo esquema, como criar novas hierarquias de RBAC ou limpar dados de cache.

6.3.1 Criando Migrações

Para criar uma nova migração, execute o seguinte comando:

```
yii migrate/create <nome>
```

O argumento obrigatório `nome` serve como uma breve descrição sobre a nova migração. Por exemplo, se a migração é sobre a criação de uma nova tabela chamada *noticias*, você pode usar o nome `criar_tabela_noticias` e executar o seguinte comando:

```
yii migrate/create criar_tabela_noticias
```

Observação: Como o argumento `nome` será usado como parte do nome da classe de migração gerada, este deve conter apenas letras, dígitos, e/ou underline.

O comando acima criará um novo arquivo contendo uma classe PHP chamada `m150101_185401_criar_tabela_noticias.php` na pasta `@app/migrations`. O arquivo contém o seguinte código que declara a classe de migração `m150101_185401_criar_tabela_noticias` com o código esqueleto:

```
<?php

use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_criar_tabela_noticias extends Migration
{
    public function up()
    {
    }

    public function down()
    {
        echo "m101129_185401_criar_tabela_noticias cannot be reverted.\n";
        return false;
    }
}
```

Cada migração de dados é definida como uma classe PHP estendida de `yii\db\Migration`. O nome da classe de migração é automaticamente gerado no formato `m<YYMMDD_HHMMSS>_<Nome>`, onde

- `<YYMMDD_HHMMSS>` refere-se a data UTC em que o comando de criação da migração foi executado.
- `<Nome>` é igual ao valor do argumento `nome` que você passou no comando.

Na classe de migração, é esperado que você escreva no método `up()` as mudanças a serem feitas na estrutura do banco de dados. Você também pode escrever códigos no método `down()` para reverter as mudanças feitas por `up()`.

O método `up()` é invocado quando você atualiza o seu banco de dados com esta migração, enquanto o método `down()` é invocado quando você reverte as mudanças no banco. O seguinte código mostra como você pode implementar a classe de migração para criar a tabela `noticias`:

```
use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_criar_tabela_noticias extends \yii\db\Migration
{
    public function up()
    {
        $this->createTable('noticias', [
            'id' => Schema::TYPE_PK,
            'titulo' => Schema::TYPE_STRING . ' NOT NULL',
            'conteudo' => Schema::TYPE_TEXT,
        ]);
    }

    public function down()
    {
        $this->dropTable('noticias');
    }
}
```

Observação: Nem todas as migrações são reversíveis. Por exemplo, se o método `up()` deleta um registro de uma tabela, você possivelmente não será capaz de recuperar este registro com o método `down()`. Em alguns casos, você pode ter tido muita preguiça e não ter implementado o método `down()`, porque não é muito comum reverter migrações de dados. Neste caso, você deve retornar `false` no método `down()` para indicar que a migração não é reversível.

A classe base `yii\db\Migration` expõe a conexão ao banco através da propriedade `db`. Você pode usá-la para manipular o esquema do banco de dados usando os métodos como descritos em *Trabalhando com um Esquema de Banco de Dados*.

Ao invés de usar tipos físicos, ao criar uma tabela ou coluna, você deve usar *tipos abstratos* para que suas migrações sejam independentes do SGBD. A classe `yii\db\Schema` define uma gama de constantes para representar os tipos abstratos suportados. Estas constantes são nomeadas no formato `TYPE_<NOME>`. Por exemplo, `TYPE_PK` refere-se ao tipo chave primária auto incrementável; `TYPE_STRING` refere-se ao tipo string. Quando a migração for aplicada a um banco de dados em particular, os tipos abstratos serão traduzidos nos respectivos tipos físicos. No caso do MySQL, `TYPE_PK` será traduzida para `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY`, enquanto `TYPE_STRING` será `varchar(255)`.

Você pode adicionar algumas constraints ao usar tipos abstratos. No exemplo acima, `NOT NULL` é adicionado a `Schema::TYPE_STRING` para especificar que a coluna não pode ser nula.

Observação: O mapeamento entre tipos abstratos e tipos físicos é especificado pela propriedade `$typeMap` em cada classe `QueryBuilder`.

Migrações Transacionais

Ao realizar migrações de dados complexas, é importante assegurar que cada migração terá sucesso ou falhará por completo para que o banco não perca sua integridade e consistência. Para atingir este objetivo, recomenda-se que você encapsule suas operações de banco de dados de cada migração em uma transação.

Um jeito mais fácil de implementar uma migração transacional é colocar o seu código de migração nos métodos `safeUp()` e `safeDown()`. Estes métodos diferem de `up()` e `down()` porque eles estão implicitamente encapsulados em uma transação. Como resultado, se qualquer operação nestes métodos falhar, todas as operações anteriores sofrerão roll back automaticamente.

No exemplo a seguir, além de criar a tabela `noticias` nós também inserimos um registro inicial a esta tabela.

```
use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_criar_tabela_noticias extends Migration
{
    public function safeUp()
    {
        $this->createTable('noticias', [
            'id' => 'pk',
            'titulo' => Schema::TYPE_STRING . ' NOT NULL',
            'conteudo' => Schema::TYPE_TEXT,
        ]);

        $this->insert('noticias', [
            'titulo' => 'título 1',
            'conteudo' => 'conteúdo 1',
        ]);
    }

    public function safeDown()
    {
        $this->delete('noticias', ['id' => 1]);
        $this->dropTable('noticias');
    }
}
```

Observe que normalmente quando você realiza múltiplas operações em `safeUp()`, você deverá reverter a ordem de execução em `safeDown()`. No exemplo acima

nós primeiramente criamos a tabela e depois inserimos uma tupla em `safeUp()`; enquanto em `safeDown()` nós primeiramente apagamos o registro e depois eliminamos a tabela.

Observação: Nem todos os SGBDs suportam transações. E algumas requisições de banco não podem ser encapsuladas em uma transação. Para alguns exemplos, referir a commit implícito⁵. Se este for o caso, implemente os métodos `up()` e `down()`.

Métodos de Acesso ao Banco de Dados

A classe base `yii\db\Migration` entrega vários métodos que facilitam o acesso e a manipulação de bancos de dados. Você deve achar que estes métodos são nomeados similarmente a métodos DAO encontrados na classe `yii\db\Command`. Por exemplo, o método `yii\db\Migration::createTable()` permite que você crie uma nova tabela assim como `yii\db\Command::createTable()` o faz.

O benefício ao usar os métodos encontrados em `yii\db\Migration` é que você não precisa criar explicitamente instancias de `yii\db\Command` e a execução de cada método automaticamente exibirá mensagens úteis que dirão a você quais operações estão sendo feitas e quanto tempo elas estão durando.

Abaixo está uma lista de todos estes métodos de acesso ao banco de dados:

- `execute()`: executando um SQL
- `insert()`: inserindo um novo registro
- `batchInsert()`: inserindo vários registros
- `update()`: atualizando registros
- `delete()`: apagando registros
- `createTable()`: criando uma tabela
- `renameTable()`: renomeando uma tabela
- `dropTable()`: removendo uma tabela
- `truncateTable()`: removendo todos os registros em uma tabela
- `addColumn()`: adicionando uma coluna
- `renameColumn()`: renomeando uma coluna
- `dropColumn()`: removendo uma coluna
- `alterColumn()`: alterando uma coluna
- `addPrimaryKey()`: adicionando uma chave primária
- `dropPrimaryKey()`: removendo uma chave primária
- `addForeignKey()`: adicionando uma chave estrangeira
- `dropForeignKey()`: removendo uma chave estrangeira
- `createIndex()`: criando um índice
- `dropIndex()`: removendo um índice

⁵<https://dev.mysql.com/doc/refman/5.1/en/implicit-commit.html>

Observação: `yii\db\Migration` não possui um método de consulta ao banco de dados. Isto porque você normalmente não precisará exibir informações extras ao recuperar informações de um banco de dados. E além disso você pode usar o poderoso [Query Builder](#) para construir e executar consultas complexas.

6.3.2 Aplicando Migrações

Para atualizar um banco de dados para a sua estrutura mais atual, você deve aplicar todas as migrações disponíveis usando o seguinte comando:

```
yii migrate
```

Este comando listará todas as migrações que não foram aplicadas até agora. Se você confirmar que deseja aplicar estas migrações, cada nova classe de migração executará os métodos `up()` ou `safeUp()` um após o outro, na ordem relacionada à data marcada em seus nomes. Se qualquer uma das migrações falhar, o comando terminará sem aplicar o resto das migrações.

Para cada migração aplicada com sucesso, o comando inserirá um registro numa tabela no banco de dados chamada `migration` para registrar uma aplicação de migração. Isto permitirá que a ferramenta de migração identifique quais migrações foram aplicadas e quais não foram.

Observação: Esta ferramenta de migração automaticamente criará a tabela `migration` no banco de dados especificado pela opção do comando `db`. Por padrão, o banco de dados é especificado por `db` em [Componentes de Aplicação](#).

Eventualmente, você desejará aplicar apenas uma ou algumas migrações, em vez de todas as disponíveis. Você pode fazê-lo especificando o número de migrações que deseja aplicar ao executar o comando. Por exemplo, o comando a seguir tentará aplicar as próximas 3 migrações disponíveis:

```
yii migrate 3
```

Você também pode especificar para qual migração em particular o banco de dados deve ser migrado usando o comando `migrate/to` em um dos formatos seguintes:

```
yii migrate/to 150101_185401                # usando a marcação de
data para especificar a migração
yii migrate/to "2015-01-01 18:54:01"        # usando uma string que
pode ser analisada por strtotime()
yii migrate/to m150101_185401_criar_tabela_noticias # usando o nome completo
yii migrate/to 1392853618                    # usando uma marcação de
data no estilo UNIX
```


Se existirem migrações mais recentes do que a especificada, elas serão todas aplicadas antes da migração definida.

Se a migração especificada já tiver sido aplicada, qualquer migração posterior já aplicada será revertida.

6.3.3 Revertendo Migrações

Para reverter uma ou múltiplas migrações que tenham sido aplicadas antes, você pode executar o seguinte comando:

```
yii migrate/down      # reverter a última migração aplicada
yii migrate/down 3     # reverter as 3 últimas migrações aplicadas
```

Observação: Nem todas as migrações são reversíveis. Tentar reverter tais migrações causará um erro que cancelará todo o processo de reversão.

6.3.4 Refazendo Migrações

Refazer as migrações significa primeiramente reverter migrações especificadas e depois aplicá-las novamente. Isto pode ser feito da seguinte maneira:

```
yii migrate/redo      # refazer a última migração aplicada
yii migrate/redo 3     # refazer as 3 últimas migrações aplicadas
```

Observação: Se a migração não for reversível, você não poderá refazê-la.

6.3.5 Listando Migrações

Para listar quais migrações foram aplicadas e quais não foram, você deve usar os seguintes comandos:

```
yii migrate/history   # exibir as 10 últimas migrações aplicadas
yii migrate/history 5  # exibir as 5 últimas migrações aplicadas
yii migrate/history all # exibir todas as migrações aplicadas

yii migrate/new       # exibir as 10 primeiras novas migrações
yii migrate/new 5     # exibir as 5 primeiras novas migrações
yii migrate/new all   # exibir todas as novas migrações
```

6.3.6 Modificando o Histórico das Migrações

Ao invés de aplicar ou reverter migrações, pode ser que você queira apenas definir que o seu banco de dados foi atualizado para uma migração em particular. Isto normalmente acontece quando você muda manualmente o banco de dados para um estado em particular, e não deseja que as mudanças para aquela migração sejam reaplicadas posteriormente. Você pode alcançar este objetivo com o seguinte comando:

```
yii migrate/mark 150101_185401 # usando a marcação de
data para especificar a migração
yii migrate/mark "2015-01-01 18:54:01" # usando uma string
que pode ser analisada por strtotime()
yii migrate/mark m150101_185401_criar_tabela_noticias # usando o nome
completo
yii migrate/mark 1392853618 # usando uma marcação
de data no estilo UNIX
```

O comando modificará a tabela `migration` adicionando ou deletando certos registros para indicar que o banco de dados sofreu as migrações especificadas. Nenhuma migração será aplicada ou revertida por este comando.

6.3.7 Customizando Migrações

Existem várias maneiras de customizar o comando de migração.

Usando Opções na Linha de Comando

O comando de migração vem com algumas opções de linha de comando que podem ser usadas para customizar o seu comportamento:

- **interactive:** boolean (o padrão é `true`), especifica se as migrações serão executadas em modo interativo. Quando for `true`, ao usuário será perguntado se a execução deve continuar antes de o comando executar certas ações. Você provavelmente marcará isto para falso se o comando estiver sendo feito em algum processo em segundo plano.
- **migrationPath:** string (o padrão é `@app/migrations`), especifica o diretório em que os arquivos das classes de migração estão. Isto pode ser especificado ou como um diretório ou como um [alias](#). Observe que o diretório deve existir, ou o comando disparará um erro.
- **migrationTable:** string (o padrão é `migration`), especifica o nome da tabela no banco de dados para armazenar o histórico das migrações. A tabela será automaticamente criada pelo comando caso não exista. Você também pode criá-la manualmente usando a estrutura `version varchar(255) primary key, apply_time integer`.
- **db:** string (o padrão é `db`), especifica o banco de dados do [componente de aplicação](#). Representa qual banco sofrerá as migrações usando este comando.
- **templateFile:** string (o padrão é `@yii/views/migration.php`), especifica o caminho do arquivo de modelo que é usado para gerar um esqueleto para os arquivos das classes de migração. Isto pode ser especificado por um caminho de arquivo ou por um [alias](#). O arquivo modelo é um script PHP em que você pode usar uma variável pré-definida `$className` para obter o nome da classe de migração.

O seguinte exemplo exhibe como você pode usar estas opções.

Por exemplo, se nós quisermos migrar um módulo `forum` cujo os arquivos de migração estão localizados dentro da pasta `migrations` do módulo, nós podemos usar o seguinte comando:

```
# migrate the migrations in a forum module non-interactively
yii migrate --migrationPath=@app/modules/forum/migrations --interactive=0
```

Configurando o Comando Globalmente

Ao invés de fornecer opções todas as vezes que você executar o comando de migração, você pode configurá-lo de uma vez por todas na configuração da aplicação como exibido a seguir:

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationTable' => 'backend_migration',
        ],
    ],
];
```

Com a configuração acima, toda a vez que você executar o comando de migração, a tabela `backend_migration` será usada para gravar o histórico de migração. Você não precisará mais fornecê-la através da opção `migrationTable`.

6.3.8 Migrando Múltiplos Bancos De Dados

Por padrão, as migrações são aplicadas no mesmo banco de dados especificado por `db` do [componente de aplicação](#). Se você quiser que elas sejam aplicadas em um banco de dados diferente, você deve especificar na opção `db` como exibido a seguir:

```
yii migrate --db=db2
```

O comando acima aplicará as migrações para o banco de dados `db2`.

Algumas vezes pode ocorrer que você queira aplicar *algumas* das migrações para um banco de dados, e outras para outro banco de dados. Para atingir este objetivo, ao implementar uma classe de migração você deve especificar a ID do componente DB que a migração usará, como o seguinte:

```
use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_criar_tabela_noticias extends Migration
{
    public function init()
    {
        $this->db = 'db2';
    }
}
```

```
        parent::init();  
    }  
}
```

A migração acima será aplicada a `db2`, mesmo que você especifique um banco de dados diferente através da opção `db`. Observe que o histórico da migração continuará sendo registrado no banco especificado pela opção `db`. Se você tiver múltiplas migrações que usam o mesmo banco de dados, é recomendase criar uma classe de migração base com o código acima em `init()`. Então cada classe de migração poderá ser estendida desta classe base.

Dica: Apesar de definir a propriedade `db`, você também pode operar em diferentes bancos de dados ao criar novas conexões de banco para eles em sua classe de migração. Você então usará os [métodos DAO](#) com estas conexões para manipular diferentes bancos de dados.

Outra estratégia que você pode seguir para migrar múltiplos bancos de dados é manter as migrações para diferentes bancos de dados em diferentes pastas de migrações. Então você poderá migrar estes bancos de dados em comandos separados como os seguintes:

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1  
yii migrate --migrationPath=@app/migrations/db2 --db=db2  
...
```

O primeiro comando aplicará as migrações em `@app/migrations/db1` para o banco de dados `db1`, e o segundo comando aplicará as migrações em `@app/migrations/db2` para `db2`, e assim sucessivamente.

Capítulo 7

Coletando Dados de Usuários

Error: not existing file: input-forms.md

Error: not existing file: input-validation.md

Error: not existing file: input-file-upload.md

Error: not existing file: input-tabular-input.md

Error: not existing file: input-multiple-models.md

Error: not existing file: input-form-javascript.md

Capítulo 8

Exibindo Dados

Error: not existing file: output-formatting.md

8.1 Paginação

Quando existem muitos dados para serem exibidos em uma única página, uma estratégia comum é mostrá-los em várias páginas e em cada página exibir uma porção pequena dos dados. Esta estratégia é conhecida como *paginação*. O Yii usa o objeto `yii\data\Pagination` para representar as informações sobre um esquema de paginação. Em particular,

- **contagem total** especifica o número total de itens de dados. Note que este é geralmente muito maior do que o número de itens de dados necessários para exibir em uma única página.
- **quantidade por página** especifica quantos itens cada página contém. O padrão é 20.
- **página atual** retorna a página corrente (baseada em zero). O valor padrão é 0, ou seja, a primeira página.

Com o objeto `yii\data\Pagination` totalmente especificado, você pode recuperar e exibir dados parcialmente. Por exemplo, se você está buscando dados a partir de um banco de dados, você pode especificar as cláusulas `OFFSET` e `LIMIT` da query com os valores correspondentes fornecidos pela paginação. Abaixo está um exemplo,

```
use yii\data\Pagination;

// Cria uma query para pegar todos os artigos com status = 1
$query = Article::find()->where(['status' => 1]);

// pega o total de artigos (mas não baixa os dados ainda)
$count = $query->count();

// cria um objeto pagination com o total em $count
$pagination = new Pagination(['totalCount' => $count]);

// Lima a query usando a paginação e recupera os artigos
$articles = $query->offset($pagination->offset)
    ->limit($pagination->limit)
    ->all();
```

Qual página de artigos será devolvido no exemplo acima? Depende se um parâmetro da query chamado `page` for fornecido. Por padrão, a paginação tentará definir a **página atual** com o valor do parâmetro `page`. Se o parâmetro não for fornecido, então o padrão será 0. Para facilitar a construção de um elemento UI que suporta a paginação, Yii fornece o widget `yii\widgets\LinkPager` que exibe uma lista de botões de página na qual

os usuários podem clicar para indicar qual a página de dados deve ser exibido. O widget recebe um objeto de paginação para que ele saiba qual é a sua página corrente e quantas botões de páginas devem ser exibido. Por exemplo,

```
use yii\widgets\LinkPager;

echo LinkPager::widget([

    'pagination' => $pagination,

]);
```

Se você quer construir elemento UI manualmente, você pode utilizar `yii\data\Pagination::createUrl()` para criar URLs que conduziria a diferentes páginas. O método requer um parâmetro página e criará um formatado apropriado de URL Contendo o parâmetro página. Por exemplo,

```
// especifica a rota que o URL a ser criada deve usar

// Se você não a especificar, a atual rota requerida será usado

$pagination->route = 'article/index';

// exibe: /index.php?r=article/index&page=100

echo $pagination->createUrl(100);

// exibe: /index.php?r=article/index&page=101

echo $pagination->createUrl(101);
```

Dica: Você pode personalizar o nome do parâmetro de consulta `page` configurando a propriedade `pageParam` ao criar o objeto de paginação.

8.2 Ordenação

Ao exibir várias linhas de dados, muitas vezes é necessário que os dados sejam ordenados de acordo com algumas colunas especificadas pelos usuários finais. O Yii utiliza um objeto `yii\data\Sort` para representar as informações sobre um esquema de ordenação. Em particular,

- **attributes** especifica os *atributos* através dos quais os dados podem ser ordenados. Um atributo pode ser simples como um [atributo do model](#). Ele também pode ser um composto por uma combinação de múltiplos atributos de model ou colunas do DB. Mais detalhes serão mostrados logo a seguir.

- `attributeOrders` dá as instruções de ordenação requisitadas para cada atributo.
- `orders` dá a direção da ordenação das colunas.

Para usar `yii\data\Sort`, primeiro declare quais atributos podem ser ordenados. Em seguida, pegue a requisição com as informações de ordenação através de `attributeOrders` ou `orders`. E use-os para personalizar a consulta de dados. Por exemplo:

```
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();
```

No exemplo abaixo, dois atributos são declarados para o objeto `Sort`: `age` e `name`.

O atributo `age` é um atributo *simples* que corresponde ao atributo `age` da classe Active Record `Article`. É equivalente a seguinte declaração:

```
'age' => [
    'asc' => ['age' => SORT_ASC],
    'desc' => ['age' => SORT_DESC],
    'default' => SORT_ASC,
    'label' => Inflector::camel2words('age'),
]
```

O atributo `name` é um atributo *composto* definido por `first_name` e `last_name` de `Article`. Declara-se com a seguinte estrutura de array:

- Os elementos `asc` e `desc` determina a direção da ordenação dos atributos em ascendente ou descendente respectivamente. Seus valores representam as colunas e as direções pelas quais os dados devem ser classificados. Você pode especificar uma ou várias colunas para indicar uma ordenação simples ou composta.
- O elemento `default` especifica a direção pela qual o atributo deve ser ordenado quando requisitado. O padrão é a ordem crescente, ou seja, se a ordenação não for definida previamente e você pedir para ordenar

por esse atributo, os dados serão ordenados por esse atributo em ordem crescente.

- O elemento `label` especifica o rótulo deve ser usado quando executar `yii\data\Sort::link()` para criar um link de ordenação. Se não for definida, `yii\helpers\Inflector::camel2words()` será chamado para gerar um rótulo do nome do atributo. Perceba que não será HTML-encoded.

Observação: Você pode alimentar diretamente o valor de `orders` para a consulta do banco de dados para implementar a sua cláusula `ORDER BY`. Não utilize `attributeOrders` porque alguns dos atributos podem ser compostos e não podem ser reconhecidos pela consulta do banco de dados.

Você pode chamar `yii\data\Sort::link()` para gerar um hyperlink em que os usuários finais podem clicar para solicitar a ordenação dos dados pelo atributo especificado. Você também pode chamar `yii\data\Sort::createUrl()` para criar um URL ordenáveis. Por exemplo:

```
// especifica a rota que a URL a ser criada deve usar
// Se você não especificar isso, a atual rota requisitada será utilizada
$sort->route = 'article/index';

// exibe links direcionando a ordenação por 'name' e 'age', respectivamente
echo $sort->link('name') . ' | ' . $sort->link('age');

// exibe: /index.php?r=article/index&sort=age
echo $sort->createUrl('age');
```

O `yii\data\Sort` verifica o parâmetro `sort` da consulta para determinar quais atributos estão sendo requisitados para ordenação. Você pode especificar uma ordenação padrão através de `yii\data\Sort::$defaultOrder` quando o parâmetro de consulta não está fornecido. Você também pode personalizar o nome do parâmetro de consulta configurando propriedade `sortParam`.

8.3 Data Providers (Provedores de Dados)

Nas seções [Paginação](#) e [Ordenação](#), descrevemos como os usuários finais podem escolher uma determinada página de dados para exibir e ordená-los por determinadas colunas. Uma vez que esta tarefa de paginação e ordenação de dados é muito comum, o Yii fornece um conjunto de classes *data provider* para encapsular estes recursos.

Um data provider é uma classe que implementa `yii\data\DataProviderInterface`. Ele suporta principalmente a recuperação de dados paginados e ordenados.

Geralmente é usado para trabalhar com [widgets de dados](#) de modo que os usuários finais possam interativamente paginar e ordenar dados.

O Yii fornece as seguintes classes de data provider:

- `yii\data\ActiveDataProvider`: Utilize `yii\db\Query` ou `yii\db\ActiveQuery` para consultar dados de um database e retorná-los na forma de array ou uma instância de [Active Record](#).
- `yii\data\SqlDataProvider`: executa uma instrução SQL e retorna os dados do banco de dados como array.
- `yii\data\ArrayDataProvider`: pega um grande array e retorna apenas uma parte deste baseado na paginação e ordenação especificada.

O uso de todos estes data providers compartilham o seguinte padrão comum:

```
// cria o data provider configurando suas propriedades de paginação e
ordenação
$provider = new XyzDataProvider([
    'pagination' => [...],
    'sort' => [...],
]);

// recupera dados paginados e ordenados
$models = $provider->getModels();

// obtém o número de itens de dados na página atual
$count = $provider->getCount();

// obtém o número total de itens de dados de todas as páginas
$totalCount = $provider->getTotalCount();
```

Você define o comportamento da paginação e da ordenação do data provider configurando suas propriedades `pagination` e `sort` que correspondem às configurações `yii\data\Pagination` e `yii\data\Sort` respectivamente. Você também pode configurá-los como `false` para desativar os recursos de paginação e/ou ordenação.

Os [widgets de dados](#), assim como `yii\grid\GridView`, tem uma propriedade chamada `dataProvider` que pode receber uma instância de data provider e exibir os dados que ele fornece. Por exemplo:

```
echo yii\grid\GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

Estes data providers variam principalmente conforme a fonte de dados é especificada. Nas subseções seguintes, vamos explicar o uso detalhado de cada um dos data providers.

8.3.1 Active Data Provider

Para usar `yii\data\ActiveDataProvider`, você deve configurar sua propriedade `query`. Ele pode receber qualquer um dos objetos `yii\db\Query` ou

`yii\db\ActiveQuery`. Se for o primeiro, os dados serão retornados em array; se for o último, os dados podem ser retornados em array ou uma instância de `Active Record`. Por exemplo:

```
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,
        ],
    ],
]);

// retorna um array de objetos Post
$posts = $provider->getModels();
```

Se `$query` no exemplo acima fosse criada usando o código a seguir, então o data provider retornaria um array.

```
use yii\db\Query;

$query = (new Query())->from('post')->where(['status' => 1]);
```

Observação: Se uma query já especificou a cláusula `orderBy`, as novas instruções de ordenação dadas por usuários finais (através da configuração `sort`) será acrescentada a cláusula `orderBy` existente. Existindo qualquer uma das cláusulas `limit` e `offset` será substituído pelo request de paginação dos usuários finais (através da configuração `pagination`).

Por padrão, `yii\data\ActiveDataProvider` utiliza o componente da aplicação `db` como a conexão de banco de dados. Você pode usar uma conexão de banco de dados diferente, configurando a propriedade `yii\data\ActiveDataProvider::$db`.

8.3.2 SQL Data Provider

O `yii\data\SqlDataProvider` trabalha com uma instrução SQL, que é usado para obter os dados necessários. Com base nas especificações de `sort` e `pagination`, o provider ajustará as cláusulas `ORDER BY` e `LIMIT` da instrução SQL em conformidade para buscar somente a página de dados solicitada na ordem desejada.

Para usar `yii\data\SqlDataProvider`, você deve especificar a propriedade `sql` bem como a propriedade `totalCount`. Por exemplo:

```
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',
        ],
    ],
]);

// retorna um array de linha de dados
$models = $provider->getModels();
```

Observação: A propriedade `totalCount` é requerida somente se você precisar paginar os dados. Isto porque a instrução SQL definida por `sql` será modificada pelo provider para retornar somente a página atual de dados solicitada. O provider ainda precisa saber o número total de dados a fim de calcular corretamente o número de páginas disponíveis.

8.3.3 Array Data Provider

O `yii\data\ArrayDataProvider` é melhor usado quando se trabalha com um grande array. O provider permite-lhe retornar uma página dos dados do array ordenados por uma ou várias colunas. Para usar `yii\data\ArrayDataProvider`, você precisa especificar a propriedade `allModels` como um grande array. Elementos deste array podem ser outros arrays associados (por exemplo, resultados de uma query do [DAO](#)) ou objetos (por exemplo, uma instância do [Active Record](#)). Por exemplo:

```
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'],
    ],
]);

// obter as linhas na página corrente
$rows = $provider->getModels();

```

Observação: Comparando o Active Data Provider com o SQL Data Provider, o array data provider é menos eficiente porque requer o carregamento de *todo* os dados na memória.

8.3.4 Trabalhando com Chave de Dados

Ao usar os itens de dados retornados por um data provider, muitas vezes você precisa identificar cada item de dados com uma chave única. Por exemplo, se os itens de dados representam as informações do cliente, você pode querer usar o ID do cliente como a chave para cada dado do cliente. Data providers podem retornar uma lista das tais chaves correspondentes aos itens de dados retornados por `yii\data\DataProviderInterface::getModels()`. Por exemplo:

```

use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
]);

// retorna uma array de objetos Post
$posts = $provider->getModels();

// retorna os valores de chave primária correspondente a $posts
$ids = $provider->getKeys();

```

No exemplo abaixo, como você fornece um objeto `yii\db\ActiveQuery` para o `yii\data\ActiveDataProvider`, ele é inteligente o suficiente para retornar os valores de chave primária como chaves no resultado. Você também pode especificar explicitamente como os valores das chaves devem ser calculados configurando a propriedade `yii\data\ActiveDataProvider::$key` com um

nome de coluna ou com uma função callback que retorna os valores das chaves. Por exemplo:

```
// usa a coluna "slug" como valor da chave
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => 'slug',
]);

// usa o resultados do md5(id) como valor da chave
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => function ($model) {
        return md5($model->id);
    }
]);
```

8.3.5 Criado Data Provider Personalizado

Para criar sua própria classe de data provider personalizada, você deve implementar o `yii\data\DataProviderInterface`. Um caminho fácil é estender de `yii\data\BaseDataProvider`, o que lhe permite concentrar-se na lógica principal do data provider. Em particular, você precisa principalmente implementar os seguintes métodos:

- **prepareModels()**: prepara o data models que será disponibilizado na página atual e as retorna como um array.
- **prepareKeys()**: recebe um array de data models disponíveis e retorna chaves que lhes estão associados.
- **prepareTotalCount**: retorna um valor que indica o número total de data models no data provider.

Abaixo está um exemplo de um data provider que lê dados em CSV eficientemente:

```
<?php
use yii\data\BaseDataProvider;

class CsvDataProvider extends BaseDataProvider
{
    /**
     * @var string nome do arquivo CSV que será lido
     */
    public $filename;

    /**
     * @var string/nome da coluna chave ou função que a retorne
     */
    public $key;

    /**
     * @var SplFileObject
```

```

    */
    protected $fileObject; // SplFileObject é muito conveniente para procurar
    uma linha específica em um arquivo

    /**
     * {@inheritdoc}
     */
    public function init()
    {
        parent::init();

        // abre o arquivo
        $this->fileObject = new SplFileObject($this->filename);
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareModels()
    {
        $models = [];
        $pagination = $this->getPagination();
        if ($pagination === false) {
            // no caso não há paginação, lê todas as linhas
            while (!$this->fileObject->eof()) {
                $models[] = $this->fileObject->fgetcsv();
                $this->fileObject->next();
            }
        } else {
            // no caso existe paginação, lê somente uma página
            $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;
}
}
```

Error: not existing file: output-data-widgets.md

Error: not existing file: output-client-scripts.md

8.4 Temas

Tema é uma forma de substituir um conjunto de `views` por outras, sem a necessidade de tocar no código de renderização de view original. Você pode usar tema para alterar sistematicamente a aparência de uma aplicação.

Para usar tema, você deve configurar a propriedade `theme` da `view` (visão) da aplicação. A propriedade configura um objeto `yii\base\Theme` que rege a forma como os arquivos de views serão substituídos. Você deve principalmente especificar as seguintes propriedades de `yii\base\Theme`:

- `yii\base\Theme::$basePath`: determina o diretório de base que contém os recursos temáticos (CSS, JS, images, etc.)
- `yii\base\Theme::$baseUrl`: determina a URL base dos recursos temáticos.
- `yii\base\Theme::$pathMap`: determina as regras de substituição dos arquivos de view. Mais detalhes serão mostradas nas subseções logo a seguir.

Por exemplo, se você chama `$this->render('about')` no `SiteController`, você estará renderizando a view `@app/views/site/about.php`. Todavia, se você habilitar tema na seguinte configuração da aplicação, a view `@app/themes/basic/site/about.php` será renderizada, no lugar da primeira.

```
return [
    'components' => [
        'view' => [
            'theme' => [
                'basePath' => '@app/themes/basic',
                'baseUrl' => '@web/themes/basic',
                'pathMap' => [
                    '@app/views' => '@app/themes/basic',
                ],
            ],
        ],
    ],
];
```

Observação: Aliases de caminhos são suportados por temas. Ao fazer substituição de view, aliases de caminho serão transformados nos caminhos ou URLs reais.

Você pode acessar o objeto `yii\base\Theme` através da propriedade `yii\base\View::$theme`. Por exemplo, na view, você pode escrever o seguinte código, pois `$this` refere-se ao objeto view:

```
$theme = $this->theme;

// retorno: $theme->baseUrl . '/img/logo.gif'
$url = $theme->getUrl('img/logo.gif');

// retorno: $theme->basePath . '/img/logo.gif'
$file = $theme->getPath('img/logo.gif');
```

A propriedade `yii\base\Theme::$pathMap` rege como a view deve ser substituída. É preciso um array de pares de valores-chave, onde as chaves são os caminhos originais da view que serão substituídos e os valores são os caminhos dos temas correspondentes. A substituição é baseada na correspondência parcial: Se um caminho de view inicia com alguma chave no array `pathMap`, a parte correspondente será substituída pelo valor do array. Usando o exemplo de configuração acima, `@app/views/site/about.php` corresponde parcialmente a chave `@app/views`, ele será substituído por `@app/themes/basic/site/about.php`.

8.4.1 Tema de Módulos

A fim de configurar temas por módulos, `yii\base\Theme::$pathMap` pode ser configurado da seguinte forma:

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/modules' => '@app/themes/basic/modules', // <-- !!!
],
```

Isto lhe permitirá tematizar `@app/modules/blog/views/comment/index.php` com `@app/themes/basic/modules/blog/views/comment/index.php`.

8.4.2 Tema de Widgets

A fim de configurar temas por widgets, você pode configurar `yii\base\Theme::$pathMap` da seguinte forma:

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/widgets' => '@app/themes/basic/widgets', // <-- !!!
],
```

Isto lhe permitirá tematizar `@app/widgets/currency/views/index.php` com `@app/themes/basic/widgets/currency/vi`

8.4.3 Herança de Tema

Algumas vezes você pode querer definir um tema que contém um visual básico da aplicação, e em seguida, com base em algum feriado, você pode querer variar o visual levemente. Você pode atingir este objetivo usando herança de tema que é feito através do mapeamento de um único caminho de view para múltiplos alvos. Por exemplo:

```
'pathMap' => [
    '@app/views' => [
        '@app/themes/christmas',
        '@app/themes/basic',
    ],
],
```

Neste caso, a view `@app/views/site/index.php` seria tematizada tanto como `@app/themes/christmas/site/index.php` ou `@app/themes/basic/site/index.php`, dependendo de qual arquivo de tema existir. Se os dois arquivos existirem, o primeiro terá precedência. Na prática, você iria manter mais arquivos de temas em `@app/themes/basic` e personalizar alguns deles em `@app/themes/christmas`.

Capítulo 9

Segurança

Error: not existing file: security-overview.md

9.1 Autenticação

Autenticação é o processo de verificação da identidade do usuário. Geralmente é usado um identificador (ex. um nome de usuário ou endereço de e-mail) e um token secreto (ex. uma senha ou um token de acesso) para determinar se o usuário é quem ele diz ser. Autenticação é a base do recurso de login.

O Yii fornece um framework de autenticação com vários componentes que dão suporte ao login. Para usar este framework, você precisará primeiramente fazer o seguinte:

- Configurar o componente `user` da aplicação;
- Criar uma classe que implementa a interface `yii\web\IdentityInterface`.

9.1.1 Configurando o `yii\web\User`

O componente `user` da aplicação gerencia o status de autenticação dos usuários. Ele requer que você especifique uma **classe de identidade** que contém a atual lógica de autenticação. Na configuração abaixo, a **classe de identidade** do `user` é configurada para ser `app\models\User` cuja implementação é explicada na próxima subseção:

```
return [  
    'components' => [  
        'user' => [  
            'identityClass' => 'app\models\User',  
        ],  
    ],  
];
```

9.1.2 Implementação do `yii\web\IdentityInterface`

A **classe de identidade** deve implementar a interface `yii\web\IdentityInterface` que contém os seguintes métodos:

- `findIdentity()`: ele procura por uma instância da classe de identidade usando o ID do usuário especificado. Este método é usado quando você precisa manter o status de login via sessão.
- `findIdentityByAccessToken()`: ele procura por uma instância da classe de identidade usando o token de acesso informado. Este método é usado quando você precisa autenticar um usuário por um único token secreto (por exemplo, em uma aplicação stateless RESTful).
- `getId()`: Retorna o ID do usuário representado por essa instância da classe de identidade.
- `getAuthKey()`: retorna uma chave para verificar o login via cookie. A chave é mantida no cookie do login e será comparada com o informação do lado servidor para atestar a validade do cookie.

- `validateAuthKey()`: Implementa a lógica de verificação da chave de login via cookie.

Se um método em particular não for necessário, você pode implementá-lo com um corpo vazio. Por exemplo, se a sua aplicação é somente stateless RESTful, você só precisa implementar `findIdentityByAccessToken()` e `getId()` deixando todos os outros métodos com um corpo vazio.

No exemplo a seguir, uma classe de identidade é implementado como uma classe [Active Record](#) associada com a tabela `user` do banco de dados.

```
<?php

use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function tableName()
    {
        return 'user';
    }

    /**
     * Localiza uma identidade pelo ID informado
     *
     * @param string/int $id o ID a ser localizado
     * @return IdentityInterface|null o objeto da identidade que corresponde
     ao ID informado
     */
    public static function findIdentity($id)
    {
        return static::findOne($id);
    }

    /**
     * Localiza uma identidade pelo token informado
     *
     * @param string $token o token a ser localizado
     * @return IdentityInterface|null o objeto da identidade que corresponde
     ao token informado
     */
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }

    /**
     * @return int/string o ID do usuário atual
     */
    public function getId()
    {
        return $this->id;
    }
}
```

```

/**
 * @return string a chave de autenticação do usuário atual
 */
public function getAuthKey()
{
    return $this->auth_key;
}

/**
 * @param string $authKey
 * @return bool se a chave de autenticação do usuário atual for válida
 */
public function validateAuthKey($authKey)
{
    return $this->getAuthKey() === $authKey;
}
}

```

Como explicado anteriormente, você só precisa implementar `getAuthKey()` e `validateAuthKey()` se a sua aplicação usa recurso de login via cookie. Neste caso, você pode utilizar o seguinte código para gerar uma chave de autenticação para cada usuário e gravá-la na tabela `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;
    }
}

```

Observação: Não confunda a classe de identidade `User` com `yii\web\User`. O primeiro é a classe que implementa a lógica de autenticação. Muitas vezes, é implementado como uma classe [Active Record](#) associado com algum tipo de armazenamento persistente para armazenar as informações de credenciais do usuário. O último é um componente da aplicação responsável pela gestão do estado da autenticação do usuário.

9.1.3 Usando o yii\web\User

Você usa o yii\web\User principalmente como um componente `user` da aplicação.

É possível detectar a identidade do usuário atual utilizando a expressão `Yii::$app->user->identity`. Ele retorna uma instância da classe de identidade representando o atual usuário logado, ou `null` se o usuário corrente não estiver autenticado (acessando como convidado). O código a seguir mostra como recuperar outras informações relacionadas à autenticação de yii\web\User:

```
// identidade do usuário atual. Null se o usuário não estiver autenticado.
$identity = Yii::$app->user->identity;

// o ID do usuário atual. Null se o usuário não estiver autenticado.
$id = Yii::$app->user->id;

// se o usuário atual é um convidado (não autenticado)
$isGuest = Yii::$app->user->isGuest;
```

Para logar um usuário, você pode usar o seguinte código:

```
// encontrar uma identidade de usuário com o nome de usuário especificado.
// observe que você pode querer checar a senha se necessário
$identity = User::findOne(['username' => $username]);

// logar o usuário
Yii::$app->user->login($identity);
```

O método `yii\web\User::login()` define a identidade do usuário atual para o yii\web\User. Se a sessão estiver **habilitada**, ele vai manter a identidade na sessão para que o status de autenticação do usuário seja mantido durante toda a sessão. Se o login via cookie (ex. login “remember me”) estiver **habilitada**, ele também guardará a identidade em um cookie para que o estado de autenticação do usuário possa ser recuperado a partir do cookie enquanto o cookie permanece válido.

A fim de permitir login via cookie, você pode configurar `yii\web\User::$enableAutoLogin` como `true` na configuração da aplicação. Você também precisará fornecer um parâmetro de tempo de duração quando chamar o método `yii\web\User::login()`.

Para realizar o logout de um usuário, simplesmente chame:

```
Yii::$app->user->logout();
```

Observe que o logout de um usuário só tem sentido quando a sessão está habilitada. O método irá limpar o status de autenticação do usuário na memória e na sessão. E por padrão, ele também destruirá *todos* os dados da sessão do usuário. Se você quiser guardar os dados da sessão, você deve chamar `Yii::$app->user->logout(false)`.

9.1.4 Eventos de Autenticação

A classe `yii\web\User` dispara alguns eventos durante os processos de login e logout:

- **EVENT_BEFORE_LOGIN**: disparado no início de `yii\web\User::login()`. Se o manipulador de evento define a propriedade `isValid` do objeto de evento para `false`, o processo de login será cancelado.
- **EVENT_AFTER_LOGIN**: dispara após de um login com sucesso.
- **EVENT_BEFORE_LOGOUT**: dispara no início de `yii\web\User::logout()`. Se o manipulador de evento define a propriedade `isValid` do objeto de evento para `false`, o processo de logout será cancelado.
- **EVENT_AFTER_LOGOUT**: dispara após um logout com sucesso.

Você pode responder a estes eventos implementando funcionalidades, tais como auditoria de login, estatísticas de usuários on-line. Por exemplo, no manipulador **EVENT_AFTER_LOGIN**, você pode registrar o tempo de login e endereço IP na tabela `user`.

9.2 Autorização

Autorização é o processo que verifica se um usuário tem permissão para fazer alguma coisa. O Yii fornece dois métodos de autorização: Filtro de Controle de Acesso (ACF) e Controle de Acesso Baseado em Role (RBAC).

9.2.1 Filtro de Controle de Acesso

O Filtro de Controle de Acesso (ACF) é um método simples de autorização implementado como `yii\filters\AccessControl` que é mais indicado para aplicações que só precisam de algum controle de acesso simples. Como o próprio nome indica, ACF é uma ação de [filtro](#) que pode ser usada em um controller (controlador) ou um módulo. Enquanto um usuário faz uma solicitação para executar uma ação, ACF verificará a lista de **regras de acesso** para determinar se o usuário tem permissão para acessar a ação solicitada.

O código a seguir mostra como usar ACF no controller (controlador) `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' => ['@'],
        ],
    ],
],
];
}
// ...
}

```

No código acima do ACF é anexado ao controller (controlador) `site` como um behavior (comportamento). Esta é a maneira habitual de utilizar uma ação filtro. A opção `only` determina que o ACF só deva ser aplicado nas ações `login`, `logout` e `signup`. Todas as outras ações no controller (controlador) não estão sujeitas ao controle de acesso. A opção `rules` lista as **regras de acesso**, onde se lê da seguinte forma:

- Permite todos os usuários convidados (ainda não autorizados) acessar as ações `login` e `signup`. A opção `roles` contém um ponto de interrogação `?` que é um token especial que representa “usuários convidados”.
- Permite que usuários autenticados acessem a ação `logout`. O caractere `@` é outro token especial que representa “usuários autenticados”.

O ACF executa a verificação de autorização examinando as regras de acesso, uma por uma de cima para baixo até encontrar uma regra que corresponda ao contexto de execução atual. O valor da opção `allow` corresponde a regra que irá dizer se o usuário está autorizado ou não. Se nenhuma das regras forem correspondidas, significa que o usuário não está autorizado, e o ACF vai parar a próxima execução.

Quando o ACF determina que um usuário não está autorizado a acessar a ação atual, ele toma a seguinte medida por padrão:

- Se o usuário é convidado, será chamado `yii\web\User::loginRequired()` para redirecionar o navegador do usuário para a página de login.
- Se o usuário já está autenticado, ele lançará um `yii\web\ForbiddenHttpException`.

Você pode personalizar este behavior configurando a propriedade `yii\filters\AccessControl::$denyCallback` da seguinte forma:

```

[
    'class' => AccessControl::class,
    ...
    'denyCallback' => function ($rule, $action) {
        throw new \Exception('Você não está autorizado a acessar esta
            página');
    }
]

```

```
    }
]
```

As **regras de acesso** suporta muitas opções. A seguir, está um resumo das opções suportadas. Você também pode estender `yii\filters\AccessRule` para criar suas próprias classes personalizadas de regras de acesso.

- **allow**: especifica se é uma regra para “permitir” ou “negar”.
- **actions**: especifica quais ações essa regra corresponde. Deve ser um array de IDs das ações. A comparação é case-sensitive. Se esta opção estiver vazia ou não definida, isso significa que a regra se aplica a todas as ações.
- **controllers**: especifica que controllers (controlador) esta regra corresponde. Deve ser um array de IDs de controller. A comparação é case-sensitive. Se esta opção estiver vazia ou não definida, isso significa que a regra se aplica a todos controllers.
- **roles**: especifica quais roles de usuários que esta regra corresponde. Dois caracteres especiais são reconhecidos, e eles são verificados através `yii\web\User::$isGuest`:
 - `?`: corresponde a um usuário convidado (ainda não autenticado)
 - `@`: corresponde a um usuário autenticado

A utilização de outros nomes invocará o método `yii\web\User::can()`, que requer RBAC permitindo (a ser descrito na próxima subsecção). Se esta opção estiver vazia ou não definida, significa que esta regra se aplica a todas as roles.

- **ips**: especifica quais `client IP addresses` esta regra corresponde. Um endereço de ip pode conter o coringa `*` no final para que ele corresponda endereços IP com o mesmo prefixo. Por exemplo, `'192.168.*'` corresponde a todos os endereços IPs no seguimento `'192.168.'`. Se esta opção estiver vazia ou não definida, significa que esta regra se aplica a todos os endereços IPs.
- **verbs**: especifica quais métodos de request (ex. `GET`, `POST`) esta regra corresponde. A comparação é case-insensitive.
- **matchCallback**: especifica um PHP callable que deve ser chamado para determinar se esta regra deve ser aplicada.
- **denyCallback**: especifica um PHP callable que deve ser chamado quando esta regra negar o acesso.

Abaixo está um exemplo que mostra como fazer uso da opção `matchCallback`, que lhe permite escrever uma lógica arbitrária de validação de acesso:

```
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';
                    }
                ],
            ],
        ],
    ],
];

}

// Match callback chamada! Esta página pode ser acessado somente a cada
31 de outubro
public function actionSpecialCallback()
{
    return $this->render('happy-halloween');
}
}

```

9.2.2 Controle de Acesso Baseado em Role (RBAC)

Controle de Acesso Baseado em Role (RBAC) fornece um simples porém poderoso controle de acesso centralizado. Por favor, consulte Wikipedia¹ para obter detalhes sobre comparação de RBAC com outros sistemas de controle de acesso mais tradicionais.

Yii implementa um RBAC Hierárquico genérico, conforme NIST RBAC model². Ele fornece as funcionalidades RBAC através do **componente de aplicação** `authManager`.

O uso do RBAC divide-se em duas partes. A primeira parte é construir os dados de autorização RBAC, e a segunda parte é usar os dados de autorização para executar verificação de acesso em locais onde ela é necessária.

Para facilitar a próxima descrição, vamos primeiro introduzir alguns conceitos básicos do RBAC.

Conceitos Básicos

Uma role representa uma coleção de *permissões* (ex. criar posts, atualizar posts). Uma role pode ser atribuído a um ou vários usuários. Para verificar se um usuário tem uma permissão específica, podemos verificar se o usuário está associado a uma role que contém esta permissão.

¹https://pt.wikipedia.org/wiki/Controle_de_acesso_baseado_em_fun%C3%A7%C3%B5es

²<https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1992/10/13/role-based-access-controls/documents/ferraiolo-kuhn-92.pdf>

Associado com cada role ou permissão, pode haver uma *regra*. Uma regra representa uma parte do código que será executado durante verificação de acesso para determinar se a role ou permissão correspondentes se aplicam ao usuário corrente. Por exemplo, a permissão para “atualizar post” pode ter uma regra que verifica se o usuário corrente é quem criou o post. Durante a verificação de acesso, se o usuário NÃO for quem criou o post, ele não terá permissão para “atualizar o post”.

Ambos roles e permissões podem ser organizadas numa hierarquia. Em particular, uma role pode constituída de outras roles ou permissões; e uma permissão pode consistir em outras permissões. Yii implementa uma hierarquia de *ordem parcial* que inclui a hierarquia de *árvore* mais especial. Enquanto uma role pode conter uma permissão, o inverso não é verdadeiro.

Configurando RBAC

Antes de partimos para definir dados de autorização e realizar a verificação de acesso, precisamos configurar o componente de aplicação `authManager`. Yii oferece dois tipos de gerenciadores de autorização: `yii\rbac\PhpManager` e `yii\rbac\DbManager`. O primeiro utiliza um script PHP para armazena os dados de autorização, enquanto o último armazena os dados de autorização no banco. Você pode considerar o uso do primeiro se a sua aplicação não requerer um gerenciamento muito dinâmico das role e permissões.

Usando O código a seguir mostra como configurar o `authManager` na configuração da aplicação utilizando a classe `yii\rbac\PhpManager`:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
        // ...
    ],
];
```

O `authManager` agora pode ser acessado via `\Yii::$app->authManager`.

Por padrão, `yii\rbac\PhpManager` armazena os dados RBAC em arquivos sob o diretório `@app/rbac`. Verifique se o diretório e todos os arquivos estão com direito de escrita pelo processo do servidor da Web caso seja necessário realizar alteração on-line.

Usando O código a seguir mostra como configurar o `authManager` na configuração da aplicação utilizando a classe `yii\rbac\DbManager`:

```
return [
    // ...
```

```

        'components' => [
            'authManager' => [
                'class' => 'yii\rbac\DbManager',
            ],
            // ...
        ],
    ];

```

`DbManager` usa quatro tabelas de banco de dados para armazenar seus dados:

- `itemTable`: tabela para armazenar itens de autorização. O padrão é “auth_item”.
 - `itemChildTable`: tabela para armazenar hierarquia de itens de autorização. O padrão é “auth_item_child”.
 - `assignmentTable`: tabela para armazenar tarefas de itens de autorização. O padrão é “auth_assignment”.
 - `ruleTable`: tabela para armazenar as regras. O padrão é “auth_rule”.
- Antes de começar é preciso criar essas tabelas no banco de dados. Para fazer isto, você pode usar o migration armazenado em `@yii/rbac/migrations`:

```
yii migrate --migrationPath=@yii/rbac/migrations
```

O `authManager` já pode ser acessado via `\Yii::$app->authManager`.

Construindo Dados de Autorização

Para construir dados de autorização devem ser realizadas as seguintes tarefas:

- definir roles e permissões;
- estabelecer relações entre roles e permissões;
- definir regras;
- associar regras com roles e permissões;
- atribuir roles a usuários.

Dependendo dos requisitos de flexibilidade de autorização das tarefas acima poderia ser feito de maneiras diferentes.

Se a sua hierarquia de permissões não se altera e você tem um número fixo de usuários pode-se criar um console command que irá iniciar os dados de autorização uma vez através das APIs oferecidas pelo `authManager`:

```

<?php
namespace app\commands;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
{
    public function actionInit()
    {
        $auth = Yii::$app->authManager;

        // adiciona a permissão "createPost"
    }
}

```

```
$createPost = $auth->createPermission('createPost');
$updatePost->description = 'Create a post';
$auth->add($createPost);

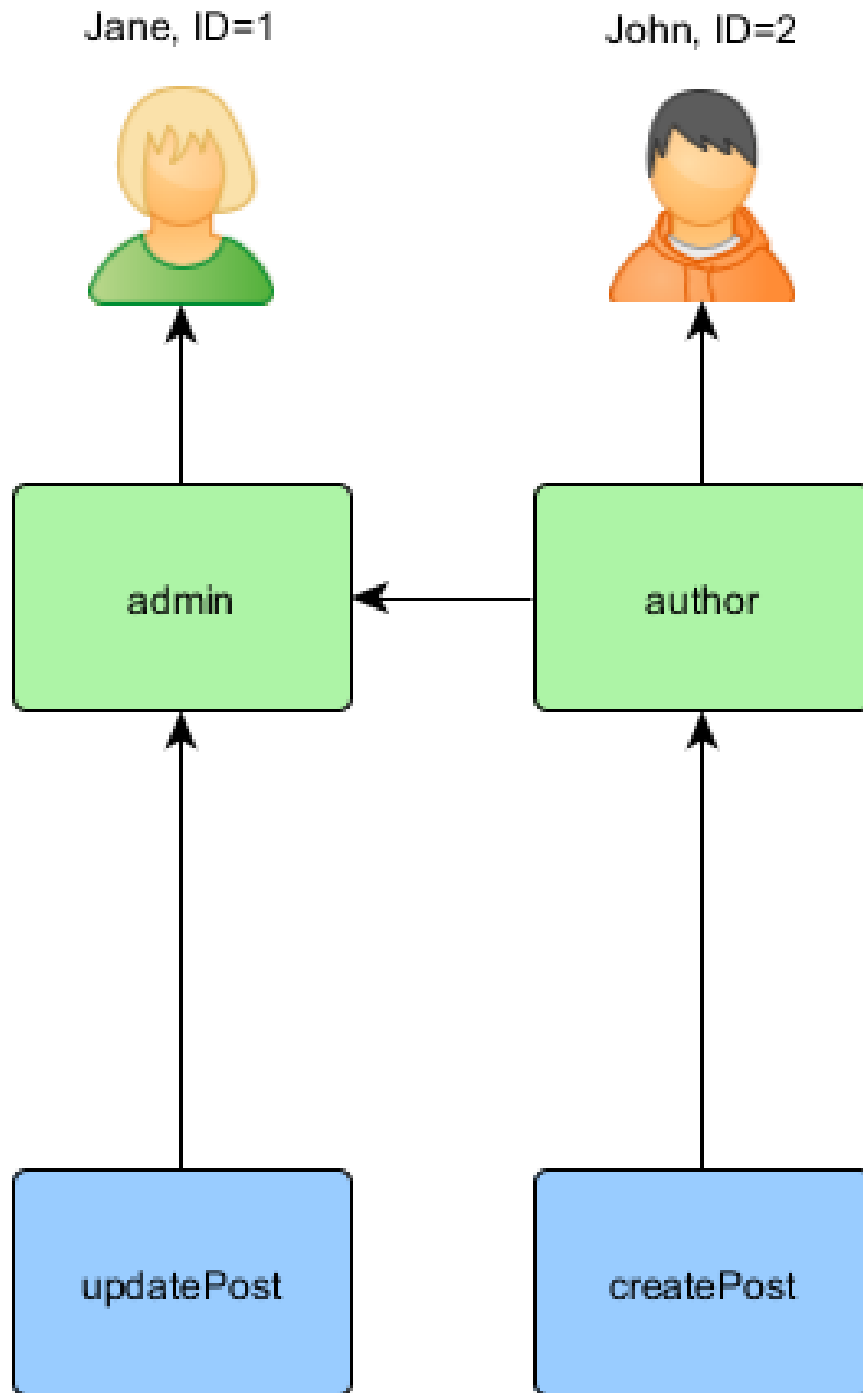
// adiciona a permissão "updatePost"
$updatePost = $auth->createPermission('updatePost');
$updatePost->description = 'Update post';
$auth->add($updatePost);

// adiciona a role "author" e da a esta role a permissão "createPost"
$author = $auth->createRole('author');
$auth->add($author);
$auth->addChild($author, $createPost);

// adiciona a role "admin" e da a esta role a permissão "updatePost"
// bem como as permissões da role "author"
$admin = $auth->createRole('admin');
$auth->add($admin);
$auth->addChild($admin, $updatePost);
$auth->addChild($admin, $author);

// Atribui roles para usuários. 1 and 2 são IDs retornados por
IdentityInterface::getId()
// normalmente implementado no seu model User.
$auth->assign($author, 2);
$auth->assign($admin, 1);
}
}
```

Depois de executar o comando com `yii rbac/init` nós vamos chegar a seguinte hierarquia:



Author pode criar post, admin pode atualizar post e fazer tudo que author pode.

Se a sua aplicação permite inscrição de usuários, você precisa atribuir roles a esses novos usuários. Por exemplo, para que todos os usuários inscritos tornem-se authors, no seu template avançado de projeto você precisa modificar o `frontend\models\SignupForm::signup()` conforme abaixo:

```
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);

        // foram adicionadas as seguintes três linhas:
        $auth = Yii::$app->authManager;
        $authorRole = $auth->getRole('author');
        $auth->assign($authorRole, $user->getId());

        return $user;
    }

    return null;
}
```

Para aplicações que requerem controle de acesso complexo com dados de autorização atualizados dinamicamente, interfaces de usuário especiais (Isto é: painel de administração) pode ser necessário desenvolver usando APIs oferecidas pelo `authManager`.

Usando Regras

Como já mencionado, regras coloca restrição adicional às roles e permissões. Uma regra é uma classe que se estende de `yii\rbac\Rule`. Ela deve implementar o método `execute()`. Na hierarquia que criamos anteriormente, author não pode editar seu próprio post. Vamos corrigir isto. Primeiro nós precisamos de uma regra para verificar se o usuário é o autor do post:

```
namespace app\rbac;

use yii\rbac\Rule;

/**
 * Verifica se o authorID corresponde ao usuário passado via parâmetro
 */
class AuthorRule extends Rule
{
    public $name = 'isAuthor';

    /**
```

```

        * @param string/int $user the user ID.
        * @param Item $item the role or permission that this rule is associated
with
        * @param array $params parameters passed to
ManagerInterface::checkAccess().
        * @return bool a value indicating whether the rule permits the role or
permission it is associated with.
    */
    public function execute($user, $item, $params)
    {
        return isset($params['post']) ? $params['post']->createdBy == $user :
false;
    }
}

```

A regra acima verifica se o post foi criado pelo \$user. Nós vamos criar uma permissão especial updateOwnPost no comando que usamos previamente:

```

$auth = Yii::$app->authManager;

// adiciona a regra
$rule = new \app\rbac\AuthorRule;
$auth->add($rule);

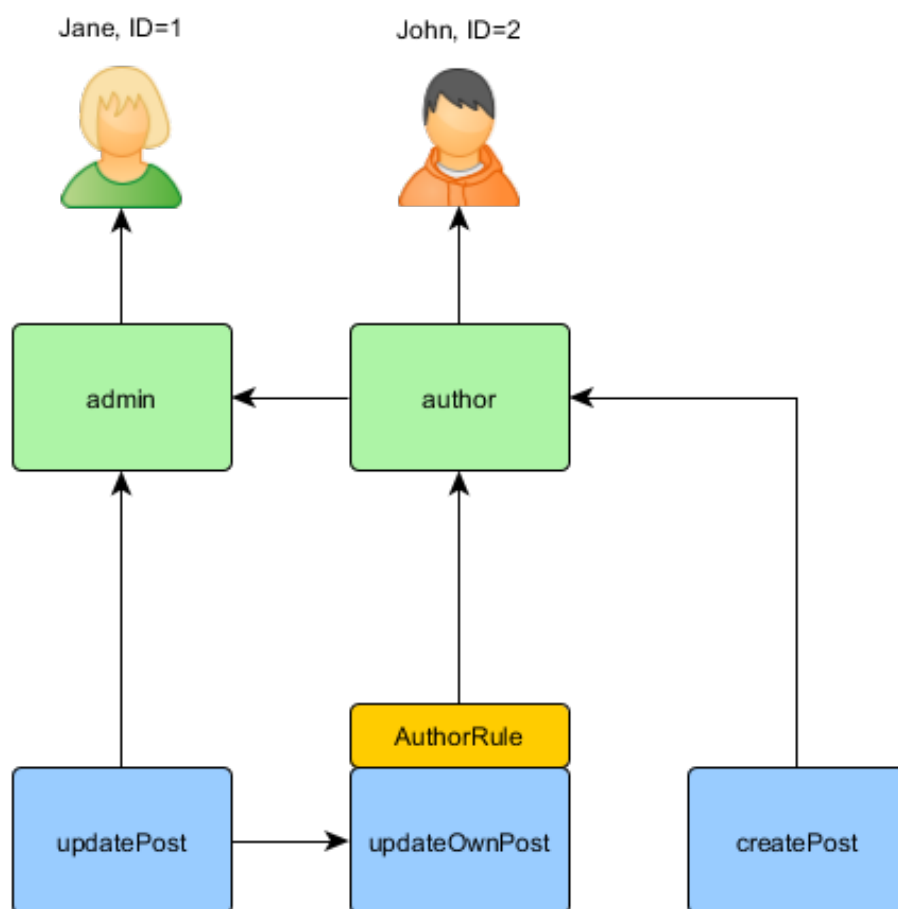
// adiciona a permissão "updateOwnPost" e associar a regra com ela.
$updateOwnPost = $auth->createPermission('updateOwnPost');
$updateOwnPost->description = 'Update own post';
$updateOwnPost->ruleName = $rule->name;
$auth->add($updateOwnPost);

// "updateOwnPost" será usado no "updatePost"
$auth->addChild($updateOwnPost, $updatePost);

// autoriza "author" a atualizar seus próprios posts
$auth->addChild($author, $updateOwnPost);

```

Agora temos a seguinte hierarquia:

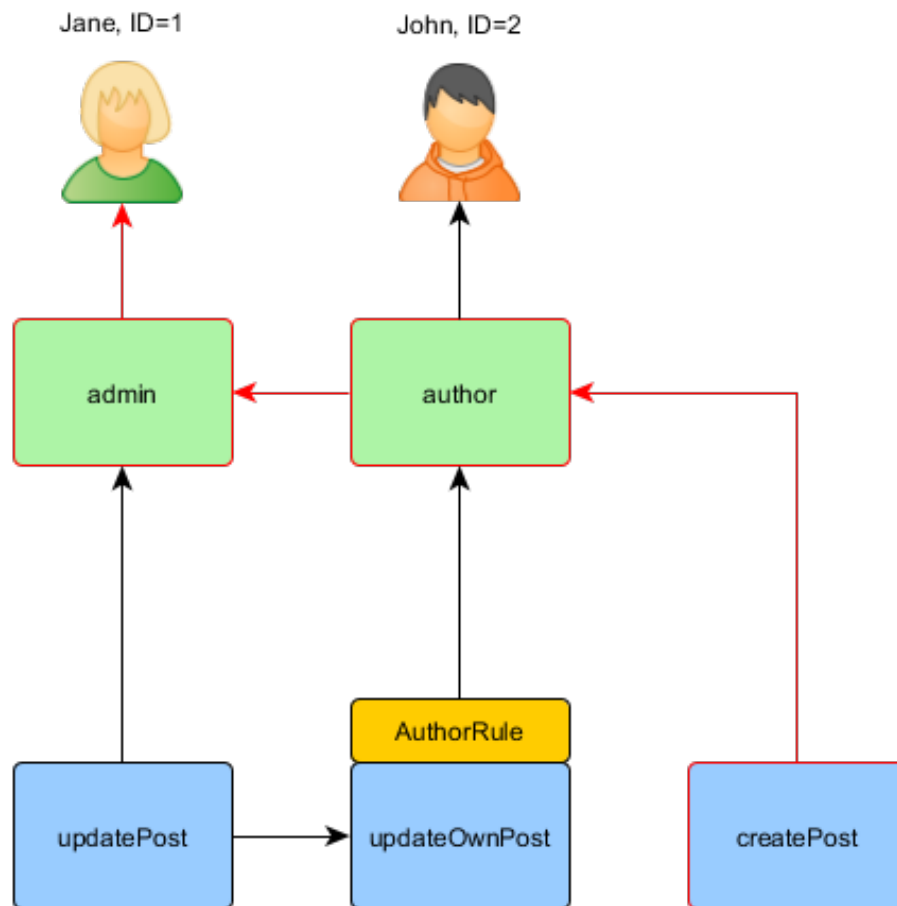


Verificação de Acesso

Com os dados de autorização prontos, você pode verificar o acesso simplesmente chamando o método `yii\rbac\ManagerInterface::checkAccess()`. Como a maioria das verificações de acesso é sobre o usuário corrente, por conveniência, o Yii fornece um método de atalho `yii\web\User::can()`, que pode ser usado como a seguir:

```
if (\Yii::$app->user->can('createPost')) {  
    // create post  
}
```

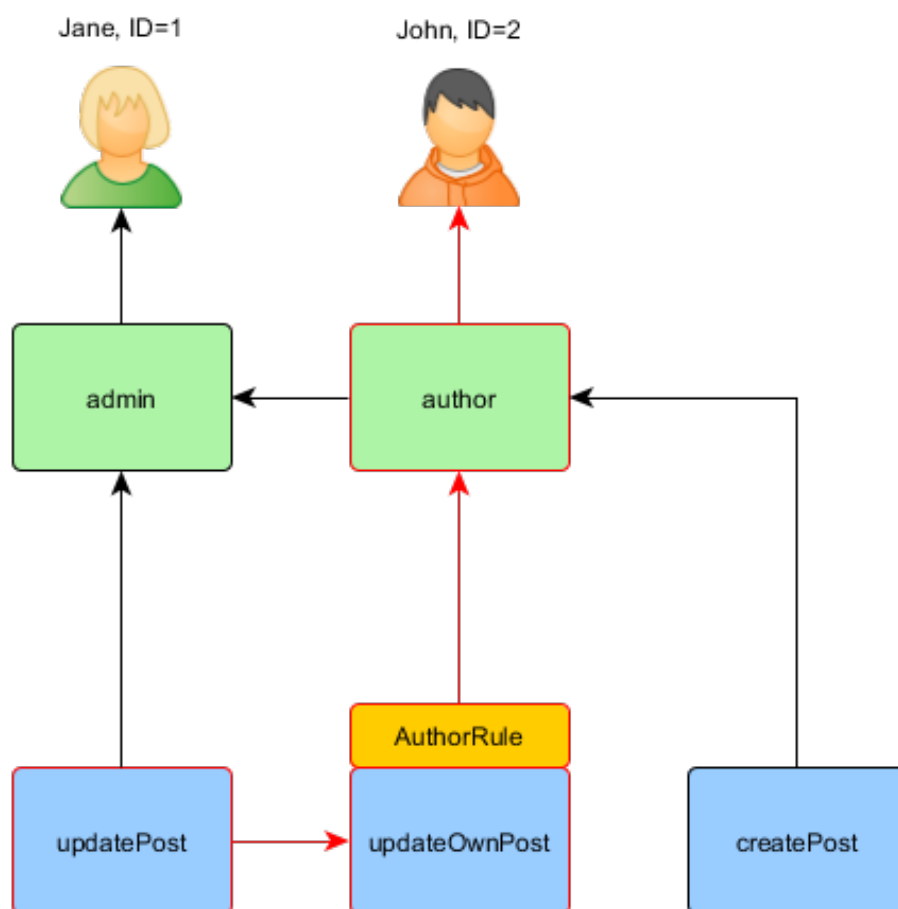
Se o usuário corrente é Jane com ID=1 começaremos com `createPost` e tentaremos chegar à Jane:



A fim de verificar se o usuário pode atualizar um post, precisamos passar um parâmetro extra que é requerido por `AuthorRule` descrito abaixo:

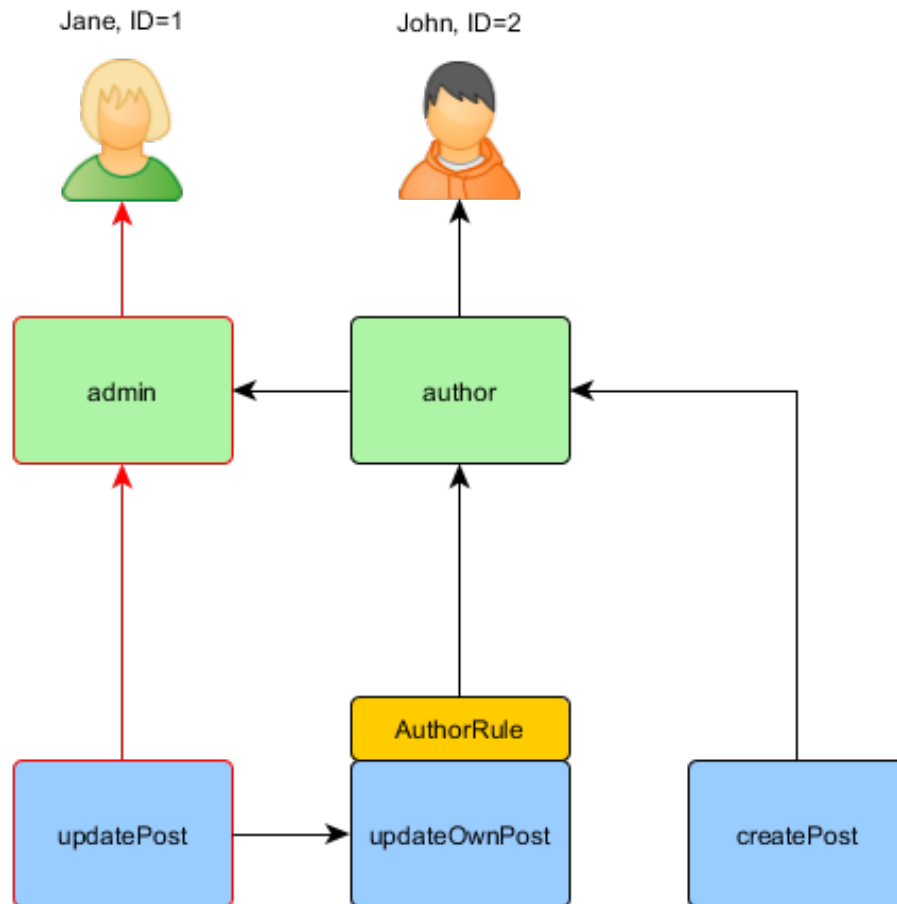
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {  
    // update post  
}
```

Aqui está o que acontece se o usuário atual é John:



Nós começamos com o `updatePost` e passamos por `updateOwnPost`. Para passar pela verificação de acesso, `AuthorRule` deve retornar `true` no seu método `execute()`. O método recebe `$params` da chamada do método `can()` de modo que o valor é `['post' => $post]`. Se tudo estiver correto, vamos chegar a `author` que é atribuído a John.

No caso de Jane é um pouco mais simples, uma vez que ela é um administrador:



Usando Roles Padrões

Uma role padrão é uma role que é *implicitamente* atribuída a *todos* os usuários. A chamada a `yii\rbac\ManagerInterface::assign()` não é necessária, e os dados de autorização não contém informação de atribuição.

Uma role padrão é geralmente associada com uma regra que determina se a role aplica-se ao do usuário que está sendo verificado.

Roles padrões são muitas vezes utilizados em aplicações que já têm algum tipo de atribuição de role. Por exemplo, uma aplicação pode ter uma coluna de “grupo” em sua tabela de usuário para representar a que grupo de privilégio cada usuário pertence. Se cada grupo privilégio pode ser mapeado para uma RBAC role, você pode usar o recurso de role padrão para associar automaticamente cada usuário ao role de RBAC. Vamos usar um exemplo para mostrar como isso pode ser feito.

Suponha que na tabela `user`, você tem uma coluna `group` que usa 1 para representar o grupo `administrator` e 2 o grupo `author`. Você pretende ter

duas roles RBAC `admin` and `author` para representar as permissões para estes dois grupos, respectivamente. Você pode configurar os dados da RBAC da seguinte forma,

```
namespace app\rbac;

use Yii;
use yii\rbac\Rule;

/**
 * Verifica se o grupo de usuário corresponde
 */
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);
// ... adiciona permissões como filhas de $author ...

$admin = $auth->createRole('admin');
$admin->ruleName = $rule->name;
$auth->add($admin);
$auth->addChild($admin, $author);
// ... adiciona permissões como filhas de $admin ...
```

Note que no exemplo acima, porque “author” é adicionado como filho de “admin”, quando você implementar o método `execute()` da classe `rule`, você também precisa respeitar essa hierarquia. É por isso que quando o nome da role é “author”, o método `execute()` retornará `true` se o grupo de usuário for 1 or 2 (significa que o usuário está no grupo “admin” ou “author”).

Em seguida, configure `authManager` listando as duas roles `yii\rbac\BaseManager`

```
::$defaultRoles:

return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
            'defaultRoles' => ['admin', 'author'],
        ],
        // ...
    ],
];
```

Agora, se você executar uma verificação de acesso, ambas as roles `admin` e `author` serão verificadas através da avaliação das regras associado com elas. se a regra retornar `true`, isso significa que a role se aplica ao usuário atual. A partir da implementação da regra acima, isto significa que se o valor do ‘grupo’ de um usuário for 1, a role `admin` seria aplicável ao usuário; e se o valor do `grupo` for 2, seria a role `author`.

Error: not existing file: security-passwords.md

Error: not existing file: security-cryptography.md

Error: not existing file: security-best-practices.md

Capítulo 10

Cache

10.1 Cache

Cache é uma maneira simples e eficiente de melhorar o desempenho de uma aplicação Web. Ao gravar dados relativamente estáticos em cache e servindo os do cache quando requisitados, a aplicação economiza o tempo que seria necessário para renderizar as informações do zero todas as vezes.

Cache pode ocorrer em diferentes níveis e locais em uma aplicação Web. No servidor, no baixo nível, cache pode ser usado para armazenar dados básicos, como a informação de uma lista de artigos mais recentes trazidos do banco de dados; e no alto nível, cache pode ser usado para armazenar fragmentos ou páginas Web inteiras, como o resultado da renderização dos artigos mais recentes. No cliente, cache HTTP pode ser usado para manter o conteúdo da última página acessada no cache do navegador.

Yii suporta todos os quatro métodos de cache:

- [Cache de Dados](#)
- [Cache de Fragmento](#)
- [Cache de Página](#)
- [Cache de HTTP](#)

10.2 Cache de Dados

O Cache de Dados é responsável por armazenar uma ou mais variáveis PHP em um arquivo temporário para ser recuperado posteriormente. Este também é a fundação para funcionalidades mais avançadas do cache, como cache de consulta e [cache de página](#).

O código a seguir é um padrão de uso típico de cache de dados, onde `$cache` refere-se a um Componente de Cache:

```
// tentar recuperar $data do cache  
$data = $cache->get($key);
```

```

if ($data === false) {

    // $data não foi encontrado no cache, calculá-la do zero

    // armazenar $data no cache para que esta possa ser recuperada na
    próxima vez
    $cache->set($key, $data);
}

// $data é acessível a partir daqui

```

10.2.1 Componentes de Cache

O cache de dados se baseia nos, então chamados, *Componentes de Cache* que representam vários armazenamentos de cache, como memória, arquivos, bancos de dados.

Componentes de Cache são normalmente registrados como *componentes de aplicação* para que possam ser globalmente configuráveis e acessíveis. O código a seguir exhibe como configurar o componente de aplicação `cache` para usar `memcached`¹ com dois servidores de cache:

```

'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
        'servers' => [
            [
                'host' => 'servidor1',
                'port' => 11211,
                'weight' => 100,
            ],
            [
                'host' => 'servidor2',
                'port' => 11211,
                'weight' => 50,
            ],
        ],
    ],
],

```

Você pode então, acessar o componente de cache acima usando a expressão `Yii::$app->cache`.

Já que todos os componentes de cache suportam as mesmas APIs, você pode trocar o componente de cache por outro reconfigurando-o nas configurações da aplicação sem modificar o código que usa o cache. Por exemplo, você pode modificar a configuração acima para usar `APC cache`:

```

'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
],

```

¹<https://memcached.org/>

1,
1,

Dica: Você pode registrar múltiplos componentes de cache na aplicação. O componente chamado `cache` é usado por padrão por muitas classes dependentes de cache (ex., `yii\web\UrlManager`).

Sistemas de Cache Suportados

Yii suporta uma ampla gama de sistemas de cache. A seguir um resumo:

- `yii\caching\ApcCache`: usa a extensão do PHP APC². Esta opção pode ser considerada a mais rápida ao se implementar o cache de uma aplicação densa e centralizada (por exemplo, um servidor, sem balanceadores de carga dedicados, etc.).
- `yii\caching\DbCache`: usa uma tabela no banco de dados para armazenar os dados em cache. Para usar este cache você deve criar uma tabela como especificada em `yii\caching\DbCache::$cacheTable`.
- `yii\caching\DummyCache`: serve apenas como um substituto e não faz nenhum cache na realidade. O propósito deste componente é simplificar o código que precisa checar se o cache está disponível. Por exemplo, durante o desenvolvimento, se o servidor não suporta cache, você pode configurar um componente de cache para usar este cache. Quando o suporte ao cache for habilitado, você pode trocá-lo para o componente correspondente. Em ambos os casos, você pode usar o mesmo código `Yii::$app->cache->get($key)` para tentar recuperar os dados do cache sem se preocupar que `Yii::$app->cache` possa ser null.
- `yii\caching\FileCache`: usa arquivos para armazenar os dados em cache. Este é particularmente indicado para armazenar grandes quantidades de dados como o conteúdo da página.
- `yii\caching\MemCache`: usa o memcache³ do PHP e as extensões memcached⁴. Esta opção pode ser considerada a mais rápida ao se implementar o cache em aplicações distribuídas (ex., vários servidores, balanceadores de carga, etc.)
- `yii\redis\Cache`: implementa um componente de cache baseado em armazenamento chave-valor Redis⁵ (requer redis versão 2.6.12 ou mais recente).
- `yii\caching\WinCache`: usa a extensão PHP WinCache⁶ (veja também⁷).
- `yii\caching\XCache` (*deprecated*): usa a extensão PHP XCache⁸.

²https://www.php.net/manual/pt_BR/book.apcu.php

³https://www.php.net/manual/pt_BR/book.memcache.php

⁴https://www.php.net/manual/pt_BR/book.memcached.php

⁵<https://redis.io/>

⁶<https://iis.net/downloads/microsoft/wincache-extension>

⁷https://www.php.net/manual/pt_BR/book.wincache.php

⁸https://en.wikipedia.org/wiki/List_of_PHP_accelerators#XCache

- `yii\caching\ZendDataCache` (*deprecated*): usa Cache de Dados Zend⁹ como o meio de cache subjacente.

Dica: Você pode usar vários tipos de cache na mesma aplicação. Uma estratégia comum é usar caches baseados em memória para armazenar dados pequenos mas constantemente usados (ex., dados estatísticos), e usar caches baseados em arquivo ou banco de dados para armazenar dados que são maiores mas são menos usados (ex., conteúdo da página).

10.2.2 APIs De Cache

Todos os componentes de caches estendem a mesma classe base `yii\caching\Cache` e assim suportam as seguintes APIs:

- `get()`: recupera um registro no cache usando uma chave específica. Retorna `false` caso o item não for encontrado no cache ou se o registro está expirado/invalidado.
- `set()`: armazena um registro no cache identificado por uma chave.
- `add()`: armazena um registro no cache identificado por uma chave se a chave não for encontrada em cache.
- `mget()`: recupera múltiplos registros do cache com as chaves especificadas.
- `mset()`: armazena múltiplos registros no cache. Cada item identificado por uma chave.
- `madd()`: armazena múltiplos registros no cache. Cada item identificado por uma chave. Se a chave já existir em cache, o registro é ignorado.
- `exists()`: retorna se a chave específica é encontrada no cache.
- `delete()`: remove um registro do cache identificado por uma chave.
- `flush()`: remove todos os registros do cache.

Observação: Não armazene o valor booleano `false` diretamente, porque o método `get()` retorna `false` para indicar que o registro não foi encontrado em cache. Você pode armazenar `false` em um array e armazenar este em cache para evitar este problema.

Alguns tipos de cache como MemCache, APC, suportam recuperar em lote múltiplos registros em cache, o que pode reduzir o custo de processamento envolvido ao recuperar informações em cache. As APIs `mget()` e `madd()` são equipadas para explorar esta funcionalidade. Em caso do cache em questão não suportar esta funcionalidade, ele será simulado.

Como `yii\caching\Cache` implementa `ArrayAccess`, um componente de cache pode ser usado como um array. A seguir alguns exemplos:

⁹https://files.zend.com/help/Zend-Server-6/zend-server.htm#data_cache_component.htm

```
$cache['var1'] = $valor1; // equivalente a: $cache->set('var1', $valor1);
$valor2 = $cache['var2']; // equivalente a: $valor2 = $cache->get('var2');
```

Chaves de Cache

Cada registro armazenado no cache é identificado por uma chave única. Quando você armazena um registro em cache, você deve especificar uma chave para ele. Mais tarde, quando você quiser recuperar o registro do cache, você deve fornecer a chave correspondente.

Você pode usar uma string ou um valor arbitrário como uma chave do cache. Quando a chave não for uma string, ela será automaticamente serializada em uma string.

Uma estratégia comum ao definir uma chave de cache é incluir todos os fatores determinantes na forma de um array. Por exemplo, `yii\db\Schema` usa a seguinte chave para armazenar a informação de um esquema de uma tabela do banco de dados.

```
[
    __CLASS__,           // nome da classe do esquema
    $this->db->dsn,        // nome da fonte de dados da conexão BD
    $this->db->username,    // usuário da conexão BD
    $name,               // nome da tabela
];
```

Como você pode ver, a chave inclui toda a informação necessária para especificar unicamente uma tabela do banco.

Quando o cache de diferentes aplicações é armazenado no mesmo lugar, é aconselhável especificar, para cada aplicação, um prefixo único a chave do cache para evitar conflitos entre elas. Isto pode ser feito ao configurar a propriedade `yii\caching\Cache::$keyPrefix`. Por exemplo, na configuração da aplicação você pode escrever o seguinte código:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
        'keyPrefix' => 'minhaapp', // um prefixo de chave único
    ],
],
```

Para assegurar interoperabilidade, apenas caracteres alfanuméricos devem ser usados.

Expiração de Cache

Um registro armazenado em cache não será apagado a menos que seja removido por alguma política aplicada (por exemplo, espaço determinado para o cache esteja cheio e os registros mais antigos sejam removidos). Para alterar este comportamento, você pode fornecer um parâmetro de expiração ao

chamar `set()` para armazenar um registro. O parâmetro indica por quantos segundos um registro pode permanecer validado no cache. Quando você chamar `get()` para recuperar um registro, se o tempo de expiração houver passado, o método retornará `false`, indicando que o registro não foi encontrado no cache. Por exemplo,

```
// Manter o registro em cache por até 45 segundos
$cache->set($chave, $registro, 45);

sleep(50);

$data = $cache->get($chave);
if ($registro === false) {
    // $registro está expirado ou não foi encontrado no sistema
}
```

Dependências de Cache

Além da definição de expiração, um registro em cache pode também ser invalidado por mudanças nas, então chamadas, *dependências de cache*. Por exemplo, `yii\caching\FileDependency` representa a dependência na data de modificação de um arquivo. Quando esta dependência muda, significa que o arquivo correspondente foi mudado. Como um resultado, qualquer arquivo com data ultrapassada encontrado no cache deve ser invalidado e a chamada de `get()` retornará `false`.

Dependências de Cache são representadas como objetos de classes dependentes de `yii\caching\Dependency`. Quando você chamar `set()` para armazenar um registro em cache, você pode passar um objeto de dependência. Por exemplo,

```
// Criar uma dependência sobre a data de modificação do arquivo
exemplo.txt.
$dependencia = new \yii\caching\FileDependency(['fileName' =>
'exemplo.txt']);

// O registro expirará em 30 segundos.
// Ele também pode ser invalidado antes, caso o exemplo.txt seja
modificado.
$cache->set($key, $data, 30, $dependencia);

// O cache verificará se o registro expirou.
// E também verificará se a dependência associada foi alterada.
// Ele retornará false se qualquer uma dessas condições seja atingida.
$data = $cache->get($key);
```

Abaixo um sumário das dependências de cache disponíveis:

- `yii\caching\ChainedDependency`: a dependência muda caso alguma das dependências na cadeia for alterada.

- `yii\caching\DbDependency`: a dependência muda caso o resultado da consulta especificada pela instrução SQL seja alterado.
- `yii\caching\ExpressionDependency`: a dependência muda se o resultado da expressão PHP especificada for alterado.
- `yii\caching\CallbackDependency`: a dependência é alterada se o resultado do callback PHP especificado for alterado..
- `yii\caching\FileDependency`: A dependência muda se a data da última alteração do arquivo for alterada.
- `yii\caching\TagDependency`: associa um registro em cache com uma ou múltiplas tags. Você pode invalidar os registros em cache com a tag especificada ao chamar `yii\caching\TagDependency::invalidate()`.

10.2.3 Cache de Consulta

Cache de consulta é uma funcionalidade especial de cache construída com o cache de dados. Ela é fornecida para armazenar em cache consultas ao banco de dados.

O cache de consulta requer uma **conexão ao banco de dados** e um componente de aplicação de **cache** válido. A seguir uma utilização básica do cache de consulta, assumindo que `$bd` é uma instância de `yii\db\Connection`:

```
$resultado = $bd->cache(function ($bd) {

    // O resultado da consulta SQL será entregue pelo cache
    // se o cache de consulta estiver sido habilitado e o resultado da
    // consulta for encontrado em cache
    return $bd->createCommand('SELECT * FROM clientes WHERE
    id=1')->queryOne();

});
```

Cache de consulta pode ser usado pelo **DAO** da mesma forma que um **ActiveRecord**:

```
$resultado = Cliente::getDb()->cache(function ($bd) {
    return Cliente::find()->where(['id' => 1])->one();
});
```

Informação: Alguns SGBDs (ex., MySQL¹⁰) também suportam o cache de consulta no servidor. Você pode escolher usá-lo ao invés do mecanismo de cache de consulta. O cache de consulta descrito acima tem a vantagem de poder especificar dependências de cache flexíveis e assim sendo potencialmente mais eficiente.

¹⁰<https://dev.mysql.com/doc/refman/5.6/en/query-cache.html>

Configurações

Cache de consulta tem três opções configuráveis globalmente através de `yii\db\Connection`:

- **enableQueryCache**: Configura se o cache de consulta está habilitado. O padrão é `true`. Observe que para ter efetivamente o cache de consulta habilitado, você também deve ter um cache válido como especificado por `queryCache`.
- **queryCacheDuration**: representa o número de segundos que o resultado de uma consulta pode se manter válido em cache. Você pode usar 0 para indicar que o resultado da consulta deve permanecer no cache indefinidamente. Este é o valor padrão usado quando `yii\db\Connection::cache()` é chamado sem nenhuma especificação de duração.
- **queryCache**: representa a ID do componente de aplicação de cache. Seu padrão é `'cache'`. Cache de consulta é habilitado apenas se houver um componente de aplicação de cache válido.

Usando o Cache de Consulta

Você pode usar `yii\db\Connection::cache()` se tiver múltiplas consultas SQL que precisam ser armazenadas no cache de consulta. Utilize da seguinte maneira,

```
$duracao = 60;      // armazenar os resultados em cache por 60 segundos
$dependencia = ...; // alguma dependência opcional

$result = $db->cache(function ($db) {

    // ... executar consultas SQL aqui ...

    return $result;

}, $duracao, $dependencia);
```

Qualquer consulta SQL na função anônima será armazenada em cache pela duração especificada com a dependência informada. Se o resultado da consulta for encontrado em cache e for válido, a consulta não será necessária e o resultado será entregue pelo cache. Se você não especificar o parâmetro `$duracao`, o valor de `queryCacheDuration` será usado.

Ocasionalmente em `cache()`, você pode precisar desabilitar o cache de consulta para algumas consultas em particular. Você pode usar `yii\db\Connection::noCache()` neste caso.

```
$result = $db->cache(function ($db) {

    // consultas SQL que usarão o cache de consulta
```



```
$db->noCache(function ($db) {  
    // consultas SQL que não usarão o cache de consulta  
  
});  
  
// ...  
  
return $result;  
});
```

Se você apenas deseja usar o cache de consulta para apenas uma consulta, você pode chamar `yii\db\Command::cache()` ao construir o comando. Por exemplo,

```
// usar cache de consulta e definir duração do cache para 60 segundos  
$customer = $db->createCommand('SELECT * FROM customer WHERE  
id=1')->cache(60)->queryOne();
```

Você pode também usar `yii\db\Command::noCache()` para desabilitar o cache de consulta para um único comando, Por exemplo,

```
$result = $db->cache(function ($db) {  
    // consultas SQL que usam o cache de consulta  
  
    // não usar cache de consulta para este comando  
    $customer = $db->createCommand('SELECT * FROM customer WHERE  
id=1')->noCache()->queryOne();  
  
    // ...  
  
    return $result;  
});
```

Limitações

O cache de consulta não funciona com resultados de consulta que contêm <i>manipuladores de recursos</i> (resource handlers). Por exemplo, ao usar o tipo de coluna `BLOB` em alguns SGBDs, o resultado da consulta retornará um <i>manipulador de recurso</i> (resource handler) para o registro na coluna.

Alguns armazenamentos em cache têm limitações de tamanho. Por exemplo, memcache limita o uso máximo de espaço de 1MB para cada registro. Então, se o tamanho do resultado de uma consulta exceder este limite, o cache falhará.

10.3 Cache de Fragmentos

Cache de fragmentos é responsável por armazenar em cache um fragmento de uma página Web. Por exemplo, se uma página exibe o sumário de vendas anuais em uma tabela, você pode armazenar esta tabela em cache para eliminar o tempo necessário para gerar esta tabela em cada requisição. O Cache de Fragmentos é construído a partir do [cache de dados](#).

Para usar o cache de fragmentos, utilize o seguinte modelo em uma [view](#):

```
if ($this->beginCache($id)) {  
  
    // ... gere o conteúdo aqui ...  
  
    $this->endCache();  
}
```

Ou seja, encapsule a lógica de geração do conteúdo entre as chamadas `beginCache()` e `endCache()`. Se o conteúdo for encontrado em cache, `beginCache()` renderizará o conteúdo em cache e retornará falso, e assim não executará a lógica de geração de conteúdo. Caso contrário, o conteúdo será gerado, e quando `endCache()` for chamado, o conteúdo gerado será capturado e armazenado no cache.

Assim como [cache de dados](#), uma `$id` única é necessária para identificar um conteúdo no cache.

10.3.1 Opções do Cache

Você poderá especificar opções adicionais sobre o cache de fragmentos passando um array de opções como o segundo parâmetro do método `beginCache()`. Por trás dos panos, este array de opções será usado para configurar um widget `yii\widgets\FragmentCache` que implementa, por sua vez, a funcionalidade de cache de fragmentos.

Duração

Talvez a opção que tenha mais frequência de uso seja a `duration`. Ela especifica por quantos segundos o conteúdo pode permanecer válido no cache. O código a seguir armazena em cache o fragmento do conteúdo por até uma hora:

```
if ($this->beginCache($id, ['duration' => 3600])) {  
  
    // ... gerar o conteúdo aqui ...  
  
    $this->endCache();  
}
```

Se a opção não for definida, o padrão definido é 60, que significa que o conteúdo em cache expirará em 60 segundos.

Dependências

Assim como *cache de dados*, o fragmento de conteúdo sendo armazenado em cache pode ter dependências. Por exemplo, o conteúdo de um *post* sendo exibido depende de ele ter sido ou não modificado.

Para especificar uma dependência, defina a opção **dependency**, que pode ser um objeto `yii\caching\Dependency` ou um array de configuração para criar um objeto de dependência. O código a seguir especifica que o conteúdo do fragmento depende do valor da coluna `atualizado_em`:

```
$dependency = [
    'class' => 'yii\caching\DbDependency',
    'sql' => 'SELECT MAX(atualizado_em) FROM post',
];

if ($this->beginCache($id, ['dependency' => $dependency])) {

    // ... gere o conteúdo aqui ...

    $this->endCache();
}
```

Variações

O conteúdo armazenado em cache pode variar de acordo com alguns parâmetros. Por exemplo, para uma aplicação Web que suporta múltiplos idiomas, a mesma porção de código de uma view pode gerar conteúdo diferente para cada idioma. Desta forma, você pode desejar que o código em cache exibisse um conteúdo diferente para a idioma exibido na requisição.

Para especificar variações de cache, defina a opção **variations**, que pode ser um array de valores escalares, cada um representando um fator de variação particular. Por exemplo, para fazer o conteúdo em cache variar em função da linguagem, você pode usar o seguinte código:

```
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {

    // ... gerar o conteúdo aqui ...

    $this->endCache();
}
```

Cache Alternante (Toggling Caching)

Em alguns casos, você pode precisar habilitar o cache de fragmentos somente quando certas condições se aplicam. Por exemplo, para uma página exibindo um formulário, e você deseja armazenar o formulário em cache apenas na primeira requisição (via requisição GET). Qualquer exibição subsequente (via requisição POST) ao formulário não deve ser armazenada em cache

porque o formulário pode conter os dados submetidos pelo usuário. Para assim fazê-lo, você pode definir a opção `enabled`, da seguinte maneira:

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {  
    // ... gerar conteúdo aqui ...  
    $this->endCache();  
}
```

10.3.2 *Cache Aninhado* (Nested Caching)

Cache de fragmentos pode ser aninhado. Isto é, um fragmento em cache pode estar contido em outro fragmento que também está em cache. Por exemplo, os comentários estão sendo armazenados em um cache de fragmento inserido em conjunto com o conteúdo do *post* em outro cache de fragmento. O código a seguir exibe como dois caches de fragmento podem ser aninhados.

```
if ($this->beginCache($id1)) {  
    // ...lógica de geração de conteúdo...  
    if ($this->beginCache($id2, $options2)) {  
        // ...lógica de geração de conteúdo...  
        $this->endCache();  
    }  
    // ...lógica de geração de conteúdo...  
    $this->endCache();  
}
```

Diferentes opções de cache podem ser definidas para os caches aninhados. Por exemplo, o cache interior e o cache exterior podem ter tempos diferentes de expiração. Mesmo quando um registro no cache exterior é invalidado, o cache interior ainda pode permanecer válido. Entretanto, o inverso não pode acontecer. Se o cache exterior é identificado como validado, ele continuará a servir a mesma copia em cache mesmo após o conteúdo no cache interior ter sido invalidado. Desta forma, você deve ser cuidadoso ao definir durações ou dependências para os caches aninhados, já que os fragmentos interiores ultrapassados podem ser mantidos no fragmento externo.

10.3.3 Conteúdo Dinâmico

Ao usar o cache de fragmentos, você pode encontrar-se na situação em que um grande fragmento de conteúdo é relativamente estático exceto por alguns lugares. Por exemplo, um cabeçalho de uma página pode exibir a barra

do menu principal junto ao nome do usuário logado. Outro problema é que o conteúdo sendo armazenado em cache pode conter código PHP que deve ser executado para cada requisição (ex., o código para registrar um *pacote de recursos estáticos* (asset bundles)). Ambos os problemas podem ser resolvidos com a funcionalidade, então chamada de *Conteúdos Dinâmicos*.

Um conteúdo dinâmico compreende um fragmento de uma saída que não deveria ser armazenada em cache mesmo que esteja encapsulada em um cache de fragmento. Para fazer o conteúdo dinâmico indefinidamente, este deve ser gerado pela execução de algum código PHP em cada requisição, mesmo que o conteúdo encapsulado esteja sendo servido do cache.

Você pode chamar `yii\base\View::renderDynamic()` dentro de um cache de fragmento para inserir conteúdo dinâmico no local desejado, como o seguinte,

```
if ($this->beginCache($id1)) {  
  
    // ...lógica de geração de conteúdo...  
  
    echo $this->renderDynamic('return Yii::$app->user->identity->name;');  
  
    // ...lógica de geração de conteúdo...  
  
    $this->endCache();  
}
```

O método `renderDynamic()` recebe uma porção de código PHP como parâmetro. O valor retornado pelo código PHP é tratado como conteúdo dinâmico. O mesmo código PHP será executado em cada requisição, não importando se este esteja encapsulado em um fragmento em cache ou não.

10.4 Cache de Página

O Cache de página é responsável por armazenar em cache o conteúdo de uma página inteira no servidor. Mais tarde, quando a mesma página é requisitada novamente, seu conteúdo será servido do cache em vez de ela ser gerada novamente do zero.

O Cache de página é implementado pela classe `yii\filters\PageCache`, um *filtro de ações*. Esta pode ser usada da seguinte maneira em uma classe de controller:

```
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',
        ],
    ],
];
}

```

O código acima afirma que o cache da página deve ser usado apenas para a ação `index`; o conteúdo da página deve ser armazenado em cache por, no máximo, 60 segundos e deve variar de acordo com a linguagem atual da aplicação; e esta página em cache deve ser invalidada se o número total de *posts* for alterado.

Como você pode observar, o cache de página é bastante similar ao [cache de fragmentos](#). Ambos suportam opções como `duration`, `dependencies`, `variations`, e `enabled`. Sua principal diferença é que o cache de página é implementado como um [filtro de ações](#) enquanto que o cache de fragmentos é um [widget](#).

Você pode usar o [cache de fragmentos](#) ou [conteúdo dinâmico](#) em conjunto com o cache de página.

10.5 Cache HTTP

Além do cache no servidor que nós descrevemos nas seções anteriores, aplicações Web pode também aproveitar-se de cache no cliente para economizar o tempo na montagem e transmissão do mesmo conteúdo de uma página.

Para usar o cache no cliente, você poderá configurar `yii\filters\HttpCache` como um filtro de ações de um controller ao qual o resultado de sua renderização possa ser armazenado em cache no navegador do cliente. A classe `HttpCache` funciona apenas para requisições `GET` e `HEAD`. Ele pode manipular três tipos de cache relacionados a cabeçalhos HTTP para estas requisições:

- `Last-Modified`
- `Etag`
- `Cache-Control`

10.5.1 Cabeçalho

O cabeçalho `Last-modified` usa uma data (timestamp) para indicar se a página foi modificada desde que o cliente a armazenou em cache.

Você pode configurar a propriedade `yii\filters\HttpCache::$lastModified` para permitir enviar o cabeçalho `Last-modified`. A propriedade deve ser um PHP *callable* retornando uma data (UNIX timestamp) sobre o tempo de modificação. A declaração do PHP *callable* deve ser a seguinte,

```
/**
 * @param Action $action O Objeto da ação que está sendo manipulada no
 momento
 * @param array $params o valor da propriedade "params"
 * @return int uma data(timestamp) UNIX timestamp representando o tempo da
 * última modificação na página
 */
function ($action, $params)
```

A seguir um exemplo que faz o uso do cabeçalho 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');
            },
        ],
    ];
}
```

O código acima afirma que o cache HTTP deve ser habilitado apenas para a ação `index`. Este deve gerar um cabeçalho HTTP `last-modified` baseado na última data de alteração dos *posts*. Quando um navegador visitar a página `index` pela primeira vez, a página será gerada no servidor e enviada para o navegador; Se o navegador visitar a mesma página novamente e não houver modificação dos *posts* durante este período, o servidor não irá remontar a página e o navegador usará a versão em cache no cliente. Como resultado, a renderização do conteúdo na página não será executada no servidor.

10.5.2 Cabeçalho

O cabeçalho “*Entity Tag*” (ou **Etag** abreviado) usa um hash para representar o conteúdo de uma página. Se a página for alterada, o hash irá mudar também. Ao comparar o hash mantido no cliente com o hash gerado no servidor, o cache pode determinar se a página foi alterada e se deve ser retransmitida.

Você pode configurar a propriedade `yii\filters\HttpCache::$etagSeed` para habilitar o envio do cabeçalho **Etag**. A propriedade deve ser um PHP *callable* retornando um conteúdo ou dados serializados (chamado também de *seed* na referência americana) para a geração do hash do Etag. A declaração do PHP *callable* deve ser como a seguinte,

```
/**
 * @param Action $action o objeto da ação que está sendo manipulada no
 momento
```

```

* @param array $params o valor da propriedade "params"
* @return string uma string usada como um conteúdo para gerar o hash ETag
*/
function ($action, $params)

```

A seguir um exemplo que faz o uso do cabeçalho 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]);
            },
        ],
    ];
}

```

O código acima afirma que o cache de HTTP deve ser habilitado apenas para a ação `view`. Este deve gerar um cabeçalho HTTP ETag baseado no título e no conteúdo do `post` requisitado. Quando um navegador visitar a página `view` pela primeira vez, a página será gerada no servidor e enviada para ele; Se o navegador visitar a mesma página novamente e não houver alteração para o título e o conteúdo do `post`, o servidor não remontará a página e o navegador usará a versão que estiver no cache do cliente. Como resultado, a renderização do conteúdo na página não será executada no servidor.

ETags permite estratégias mais complexas e/ou mais precisas do que o uso do cabeçalho de `Last-modified`. Por exemplo, um ETag pode ser invalidado se o site tiver sido alterado para um novo tema.

Gerações muito complexas de ETags podem contrariar o propósito de se usar `HttpCache` e introduzir despesas desnecessárias ao processamento, já que eles precisam ser reavaliados a cada requisição. Tente encontrar uma expressão simples que invalida o cache se o conteúdo da página for modificado.

Observação: Em concordância com a RFC 7232¹¹, o `HttpCache` enviará os cabeçalhos ETag e `Last-Modified` se ambos forem assim configurados. E se o cliente enviar ambos o cabeçalhos `If-None-Match` e `If-Modified-Since`, apenas o primeiro será respeitado.

10.5.3 Cabeçalho

O cabeçalho `Cache-Control` especifica políticas de cache gerais para as páginas. Você pode enviá-lo configurando a propriedade `yii\filters\HttpCache::`

¹¹<https://datatracker.ietf.org/doc/html/rfc7232#section-2.4>

`$cacheControlHeader` com o valor do cabeçalho. Por padrão, o seguinte cabeçalho será enviado:

```
Cache-Control: public, max-age=3600
```

10.5.4 Limitador de Cache na Sessão

Quando uma página usa sessão, o PHP automaticamente enviará alguns cabeçalhos HTTP relacionados ao cache como especificado na configuração `session.cache_limiter` do PHP.INI. Estes cabeçalhos podem interferir ou desabilitar o cache que você deseja do `HttpCache`. Para prevenir-se deste problema, por padrão, o `HttpCache` desabilitará o envio destes cabeçalhos automaticamente. Se você quiser modificar estes comportamentos, deve configurar a propriedade `yii\filters\HttpCache::$sessionCacheLimiter`. A propriedade pode receber um valor string, como: `public`, `private`, `private_no_expire` e `nocache`. Por favor, consulte o manual do PHP sobre `session_cache_limiter()`¹² para mais explicações sobre estes valores.

10.5.5 Implicações para SEO

Os bots do motor de buscas tendem a respeitar cabeçalhos de cache. Já que alguns rastreadores têm um limite sobre a quantidade de páginas por domínio que eles processam em um certo espaço de tempo, introduzir cabeçalhos de cache podem ajudar na indexação do seu site já que eles reduzem o número de páginas que precisam ser processadas.

¹²https://www.php.net/manual/pt_BR/function.session-cache-limiter.php

Capítulo 11

Web Services RESTful

11.1 Introdução

O Yii fornece um conjunto de ferramentas para simplificar a tarefa de implementar APIs RESTful Web Service. Em particular, o Yii suporta os seguintes recursos sobre APIs RESTful:

- Prototipagem rápida com suporte para APIs comuns de [Active Record](#);
- Negociação de formato do Response (suporte JSON e XML por padrão);
- Serialização de objeto configurável com suporte a campos de saída selecionáveis;
- Formatação adequada para a coleção e dados e validação de erros;
- Suporte a HATEOAS¹;
- Roteamento eficiente com verificação dos verbs (métodos) HTTP;
- Construído com suporte aos métodos `OPTIONS` e `HEAD`;
- Autenticação e autorização;
- Data caching e HTTP caching;
- Limitação de taxa;

Abaixo, utilizamos um exemplo para ilustrar como você pode construir um conjunto de APIs RESTful com um mínimo de codificação.

Suponha que você deseja expor os dados do usuário via APIs RESTful. Os dados do usuário estão guardados na tabela `user` e você já criou a classe `active record app\models\User` para acessar os dados do usuário.

11.1.1 Criando um Controller (Controlador)

Primeiramente, crie uma classe `controller app\controllers\UserController` como a seguir,

```
namespace app\controllers;  
  
use yii\rest\ActiveController;
```

¹<https://en.wikipedia.org/wiki/HATEOAS>

```
class UserController extends ActionController
{
    public $modelClass = 'app\models\User';
}
```

A classe controller estende de `yii\rest\ActiveController`, que implementa um conjunto comum de ações RESTful. Especificando `modelClass` como `app\models\User`, o controller sabe qual o model que pode ser usado para a recuperação e manipulação de dados.

11.1.2 Configurando Regras de URL

Em seguida, modifique a configuração do componente `urlManager` na configuração da aplicação:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

A configuração acima primeiramente adiciona uma regra de URL para o controller `user` de modo que os dados do usuário podem ser acessados e manipulados com URLs amigáveis e métodos HTTP significativos.

11.1.3 Ativando o Input via JSON

Para fazer a API aceitar dados no formato JSON, configure a propriedade `parsers` do [componente de aplicação request](#) para usar o `yii\web\JsonParser` para realizar input via JSON:

```
'request' => [
    'parsers' => [
        'application/json' => 'yii\web\JsonParser',
    ],
]
```

Observação: A configuração acima é opcional. Sem esta configuração, a API só iria reconhecer os formatos de input `application/x-www-form-urlencoded` e `multipart/form-data`.

11.1.4 Testando

Com o mínimo de esforço acima, você já terminou sua tarefa de criar as APIs RESTful para acessar os dados do usuário. As APIs que você criou incluem:

- GET /users: listar todos os usuários página por página;
- HEAD /users: mostrar a informações gerais da listagem de usuários;
- POST /users: criar um novo usuário;
- GET /users/123: retorna detalhes do usuário 123;
- HEAD /users/123: mostra informações gerais do usuário 123;
- PATCH /users/123 e PUT /users/123: atualiza o usuário 123;
- DELETE /users/123: deleta o usuário 123;
- OPTIONS /users: mostra os métodos suportados em relação à URL /users;
- OPTIONS /users/123: mostra os métodos suportados em relação à URL /users/123.

Observação: O Yii vai pluralizar automaticamente nomes de controllers para uso em URLs (também chamadas *endpoints*). Você pode configurar isso usando a propriedade `yii\rest\UrlRule::$pluralize`.

Você pode acessar suas APIs com o comando `curl` mostrado abaixo,

```
$ curl -i -H "Accept:application/json" "http://localhost/users"
```

```
HTTP/1.1 200 OK
...
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

[
  {
    "id": 1,
    ...
  },
  {
    "id": 2,
    ...
  },
  ...
]
```

Tente alterar o tipo de conteúdo para `application/xml` e você vai ver o resultado retornado em formato XML:

```
$ curl -i -H "Accept:application/xml" "http://localhost/users"
```

```
HTTP/1.1 200 OK
```

```
...
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <item>
    <id>1</id>
    ...
  </item>
  <item>
    <id>2</id>
    ...
  </item>
  ...
</response>
```

O seguinte comando irá criar um novo usuário, enviando uma solicitação POST com os dados do usuário em formato JSON:

```
$ curl -i -H "Accept:application/json" -H "Content-Type:application/json"
-XPOST "http://localhost/users" -d '{"username": "example", "email":
"user@example.com"}'
```

```
HTTP/1.1 201 Created
...
Location: http://localhost/users/1
Content-Length: 99
Content-Type: application/json; charset=UTF-8
```

```
{"id":1,"username":"example","email":"user@example.com","created_at":1414674789,"updated_at":1414674789}
```

Dica: Você também pode acessar suas APIs via navegador, digitando a URL `http://localhost/users`. No entanto, você pode precisar de alguns plugins do navegador para enviar cabeçalhos de solicitações específicas.

Como você pode ver, no cabeçalho da resposta, há informações sobre a contagem total, número de páginas, etc. Há também links que permitem navegar para outras páginas de dados. Por exemplo, `http://localhost/users?page=2` lhe daria a próxima página dos dados de usuário.

Usando os parâmetros `fields` e `expand`, você também pode especificar os campos que devem ser incluídos no resultado. Por exemplo, a URL `http://localhost/users?fields=id,email` só retornará os campos `id` e `email`.

Observação: Você deve ter notado que o resultado de `http://localhost/users` inclui alguns campos confidenciais, Tal como `password_hash`, `auth_key`. Você certamente não quer que eles apareçam no resultado da sua API. Você pode e deve filtrar esses campos, conforme descrito na seção [Response Formatting](#).

11.1.5 Resumo

Usando o framework API RESTful do Yii, você implementa uma URL desses campos, conforme descrito na seção de ações do controller, um controller para organizar as ações que implementam as URLs para um único tipo de recurso.

Os recursos são representados como modelos de dados, que se estendem a partir da classe `yii\base\Model`. Se você estiver trabalhando com bancos de dados (relacional ou NoSQL), é recomendado que você use `ActiveRecord` para representar recursos.

Você pode usar `yii\rest\UrlRule` para simplificar o roteamento para suas URLs da API.

Embora não seja exigido, é recomendável que você desenvolva suas APIs RESTful como uma aplicação separada, diferente do seu frontend e backend para facilitar a manutenção.

11.2 Recursos

APIs RESTful tratam de como acessar e manipular *recursos*. Você pode ver recursos como [models](#) no paradigma MVC.

Embora não haja restrição na forma de representar um recurso, no Yii você normalmente representaria recursos como objetos de `yii\base\Model` ou de uma classe filha (ex. `yii\db\ActiveRecord`), pelas seguintes razões:

- `yii\base\Model` implementa a interface `yii\base\Arrayable`, que permite que você personalize como você deseja expor dados de recursos através das APIs RESTful.
- `yii\base\Model` suporta [validação de dados de entrada](#), que é importante se as suas APIs RESTful precisarem suportar entrada de dados.
- `yii\db\ActiveRecord` fornece acesso poderoso a banco de dados com suporte a manipulação dos dados, o que o torna um ajuste perfeito se seus dados de recursos estiverem armazenado em bases de dados.

Nesta seção, vamos principalmente descrever como uma classe de recurso que se estende de `yii\base\Model` (ou alguma classe filha) pode especificar quais os dados podem ser retornados via APIs RESTful. Se a classe de recurso não estender de `yii\base\Model`, então todas as suas variáveis públicas serão retornadas.

11.2.1 Campos

Ao incluir um recurso em uma resposta da API RESTful, o recurso precisa ser serializado em uma string. O Yii quebra este processo em duas etapas. Primeiro, o recurso é convertido em um array utilizando `yii\rest\Serializer`. Por último, o array é serializado em uma string no formato solicitado (ex. JSON, XML) através do `response formatters`. O primeiro passo é o que você deve centrar-se principalmente no desenvolvimento de uma classe de recurso.

Sobrescrevendo `fields()` e/ou `extraFields()`, você pode especificar quais os dados, chamados *fields*, no recurso podem ser colocados no array. A diferença entre estes dois métodos é que o primeiro especifica o conjunto padrão de campos que devem ser incluídos no array, enquanto que o último especifica campos adicionais que podem ser incluídos no array, se um usuário final solicitá-los via o parâmetro de pesquisa `expand`. Por exemplo:

```
// retorna todos os campos declarados em fields()
http://localhost/users

// retorna apenas os campos id e email, desde que estejam declarados em
fields()
http://localhost/users?fields=id,email

// retorna todos os campos de fields() e o campo profile se este estiver no
extraFields()
http://localhost/users?expand=profile

// retorna apenas o campo id, email e profile, desde que estejam em fields()
e extraFields()
http://localhost/users?fields=id,email&expand=profile
```

Sobrescrevendo

Por padrão, `yii\base\Model::fields()` retorna todos os atributos do modelo como campos, enquanto `yii\db\ActiveRecord::fields()` só retorna os atributos que tenham sido preenchidos a partir do DB.

Você pode sobrescrever `fields()` para adicionar, remover, renomear ou redefinir campos. O valor do retorno de `fields()` deve ser um array. As chaves do array são os nomes dos campos e os valores são as definições dos campos correspondentes, que podem ser tanto nomes de propriedade/atributo ou funções anônimas retornando o valor do campo correspondente. No caso especial de um nome de um campo for o mesmo que sua definição de nome de atributo, você pode omitir a chave do array. Por exemplo:

```
// explicitamente lista todos os campos,
// melhor usado quando você quer ter certeza de que as alterações
// na sua tabela ou atributo do modelo não causaram alterações
// nos seus campos (Manter compatibilidade da API).
public function fields()
```



```

{
    return [
        // Nome do campo é igual ao nome do atributo
        'id',
        // nome do campo é "email", o nome do atributo correspondente é
        "email_address"
        'email' => 'email_address',
        // nome do campo é "name", seu valor é definido por um PHP callback
        'name' => function ($model) {
            return $model->first_name . ' ' . $model->last_name;
        },
    ];
}

// filtrar alguns campos, melhor usado quando você deseja herdar a
// implementação do pai
// e deseja esconder alguns campos confidenciais.
public function fields()
{
    $fields = parent::fields();

    // remove campos que contém informações confidenciais
    unset($fields['auth_key'], $fields['password_hash'],
        $fields['password_reset_token']);

    return $fields;
}

```

Aviso: Como o padrão é ter todos os atributos de um model incluídos no resultados da API, você deve examinar os seus dados para certificar-se de que eles não contenham informações confidenciais. Se existirem tais informações, você deve sobrescrever `fields()` para filtrá-los. No exemplo acima, nós escolhemos filtrar `auth_key`, `password_hash` e `password_reset_token`.

Sobrescrevendo

Por padrão, o `yii\base\Model::extraFields()` não retorna nada, enquanto o `yii\db\ActiveRecord::extraFields()` retorna os nomes das relações que foram populadas a partir do DB.

O formato do retorno dos dados do `extraFields()` é o mesmo de `fields()`. Geralmente, `extraFields()` é mais usado para especificar os campos cujos valores são objetos. Por exemplo, dada a seguinte declaração de campo,

```

public function fields()
{
    return ['id', 'email'];
}

public function extraFields()
{

```

```

    return ['profile'];
}

```

o request com `http://localhost/users?fields=id,email&expand=profile` pode retornar o seguinte dados em formato JSON:

```

[
  {
    "id": 100,
    "email": "100@example.com",
    "profile": {
      "id": 100,
      "age": 30,
    }
  },
  ...
]

```

11.2.2 Links

HATEOAS² é uma abreviação de “Hypermedia as the Engine of Application State”, que promove as APIs Restfull retornarem informações para permitir aos clientes descobrirem quais ações são suportadas pelos recursos retornados. O sentido de HATEOAS é retornar um conjunto de hiperlinks em relação às informações quando os recursos de dados são servidos pelas APIs.

Suas classes de recursos podem suportar HATEOAS implementando a interface `yii\web\Linkable`. Esta interface contém um único método `getLinks()` que deve retornar uma lista de `links`. Tipicamente, você deve retornar pelo menos o link `self` representando a URL para o mesmo objeto de recurso. Por exemplo:

```

use yii\db\ActiveRecord;
use yii\web\Link;
use yii\web\Linkable;
use yii\helpers\Url;

class User extends ActiveRecord implements Linkable
{
    public function getLinks()
    {
        return [
            Link::REL_SELF => Url::to(['user/view', 'id' => $this->id],
                true),
        ];
    }
}

```

Quando o objeto `User` for retornado em uma resposta, será composto de um elemento `_links` representando os links relacionados ao `user`, por exemplo:

²<https://en.wikipedia.org/wiki/HATEOAS>

```
{
  "id": 100,
  "email": "user@example.com",
  // ...
  "_links" => {
    "self": {
      "href": "https://example.com/users/100"
    }
  }
}
```

11.2.3 Collections (Coleções)

Objetos de recursos podem ser agrupados em *collections*. Cada collection contém uma lista de objetos de recurso do mesmo tipo.

Embora os collections podem ser representados como arrays, normalmente, é preferível representá-los como [data providers](#). Isto porque data providers suportam ordenação e paginação de recursos, que é um recurso comumente necessário para APIs RESTful retornarem collections. Por exemplo, ação a seguir retorna um data provider sobre o recurso *post*:

```
namespace app\controllers;

use yii\rest\Controller;
use yii\data\ActiveDataProvider;
use app\models\Post;

class PostController extends Controller
{
    public function actionIndex()
    {
        return new ActiveDataProvider([
            'query' => Post::find(),
        ]);
    }
}
```

Quando um data provider está enviando uma resposta com a API RESTful, o `yii\rest\Serializer` pegará a página atual de recursos e a serializa como um array de objetos de recurso. Adicionalmente, o `yii\rest\Serializer` também incluirá as informações de paginação pelo seguinte cabeçalho HTTP:

- **X-Pagination-Total-Count**: O número total de recursos;
- **X-Pagination-Page-Count**: O número de páginas;
- **X-Pagination-Current-Page**: A página atual (a primeira página é 1);
- **X-Pagination-Per-Page**: O numero de recursos em cada página;
- **Link**: Um conjunto de links de navegação, permitindo que o cliente percorra os recursos página por página.

Um exemplo pode ser encontrado na seção [Introdução](#).

11.3 Controllers (Controladores)

Depois de criar as classes de recursos e especificar como os dados de recursos devem ser formatados, a próxima coisa a fazer é criar ações do controller para expor os recursos para os usuários finais através das APIs RESTful.

O Yii fornece duas classes básicas de controller para simplificar seu trabalho de criar ações RESTful: `yii\rest\Controller` e `yii\rest\ActiveController`. A diferença entre os dois controllers é que o último fornece um conjunto padrão de ações que são especificamente concebidos para lidar com recursos do [Active Record](#). Então, se você estiver usando [Active Record](#) e está confortável com as ações fornecidas, você pode considerar estender suas classes de controller de `yii\rest\ActiveController`, que permitirá criar poderosas APIs RESTful com um mínimo de código.

Ambas classes `yii\rest\Controller` e `yii\rest\ActiveController` fornecem os seguintes recursos, algumas das quais serão descritas em detalhes nas próximas seções:

- Validação de Método HTTP;
- [Negociação de conteúdo e formatação de dados](#);
- [Autenticação](#);
- [Limitação de taxa](#).

O `yii\rest\ActiveController` oferece também os seguintes recursos:

- Um conjunto de ações comumente necessárias: `index`, `view`, `create`, `update`, `delete`, `options`;
- Autorização do usuário em relação à ação solicitada e recursos.

11.3.1 Criando Classes Controller

Ao criar uma nova classe de controller, uma convenção na nomenclatura da classe é usar o nome do tipo de recurso no singular. Por exemplo, para disponibilizar as informações do usuário, o controlador pode ser nomeado como `UserController`. Criar uma nova ação é semelhante à criação de uma ação de uma aplicação Web. A única diferença é que em vez de renderizar o resultado usando uma view e chamando o método `render()`, para ações RESTful você retorna diretamente os dados. O `serializer` e o `objeto response` vão converter os dados originais para o formato solicitado. Por exemplo:

```
public function actionView($id)
{
    return User::findOne($id);
}
```

11.3.2 Filtros

A maioria dos recursos da API RESTful fornecidos por `yii\rest\Controller` são implementadas por [filtros](#). Em particular, os seguintes filtros serão exe-

cutados na ordem em que estão listados:

- **contentNegotiator**: suporta a negociação de conteúdo, a ser explicado na seção [Formatação de Resposta](#);
- **verbFilter**: suporta validação de métodos HTTP;
- **authenticator**: suporta autenticação de usuários, que será explicado na seção [Autenticação](#);
- **rateLimiter**: suporta limitação de taxa, que será explicado na seção [Limitação de taxa](#).

Estes filtros são declarados no método `behaviors()`. Você pode sobrescrever esse método para configurar alguns filtros, desativar outros, ou adicionar seus próprios filtros. Por exemplo, se você precisar somente de autenticação básica de HTTP, poderá utilizar o seguinte código:

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::class,
    ];
    return $behaviors;
}
```

11.3.3 Estendendo

Se a sua classe controller estende de `yii\rest\ActiveController`, você deve configurar a propriedade `modelClass` para ser o nome da classe de recurso que você pretende servir através deste controller. A classe deve estender de `yii\db\ActiveRecord`.

Customizando Ações

Por padrão, o `yii\rest\ActiveController` fornece as seguintes ações:

- **index**: recursos de lista página por página;
- **view**: retorna os detalhes de um recurso especificado;
- **create**: cria um novo recurso;
- **update**: atualiza um recurso existente;
- **delete**: excluir o recurso especificado;
- **options**: retorna os métodos HTTP suportados.

Todas essas ações são declaradas através do método `actions()`. Você pode configurar essas ações ou desativar algumas delas, sobrescrevendo o método `actions()`, como mostrado a seguir:

```
public function actions()
{
    $actions = parent::actions();
```

```

        // desabilita as ações "delete" e "create"
        unset($actions['delete'], $actions['create']);

        // customiza a preparação do data provider com o método
        "prepareDataProvider()"
        $actions['index']['prepareDataProvider'] = [$this,
            'prepareDataProvider'];

        return $actions;
    }

    public function prepareDataProvider()
    {
        // preparar e retornar um data provider para a ação "index"
    }

```

Por favor, consulte as referências de classe para classes de ação individual para saber as opções de configuração que estão disponíveis.

Executando Verificação de Acesso

Ao disponibilizar recursos por meio de APIs RESTful, muitas vezes você precisa verificar se o usuário atual tem permissão para acessar e manipular o(s) recurso(s) solicitado(s). Com o `yii\rest\ActiveController`, isso pode ser feito sobrescrevendo o método `checkAccess()` conforme a seguir:

```

/**
 * Verifica os privilégios do usuário corrente.
 *
 * Este método deve ser sobrescrito para verificar se o usuário atual tem o
 * privilégio
 * para executar a ação especificada diante do modelo de dados especificado.
 * se o usuário não tiver acesso, uma [[ForbiddenHttpException]] deve ser
 * lançada.
 *
 * @param string $action o ID da ação a ser executada
 * @param \yii\base\Model $model o model a ser acessado. Se `null`, isso
 * significa que nenhum model específico está sendo acessado.
 * @param array $params parâmetros adicionais
 * @throws ForbiddenHttpException se o usuário não tiver acesso
 */
public function checkAccess($action, $model = null, $params = [])
{
    // verifica se o usuário pode acessar $action and $model
    // lança a ForbiddenHttpException se o acesso for negado
    if ($action === 'update' || $action === 'delete') {
        if ($model->author_id !== \Yii::$app->user->id)
            throw new \yii\web\ForbiddenHttpException(sprintf('You can only
                %s articles that you\'ve created.', $action));
    }
}

```

O método `checkAccess()` será chamado pelas ações padrões do `yii\rest\ActiveController`. Se você criar novas ações e também desejar executar a verificação de acesso, deve chamar esse método explicitamente nas novas ações.

Dica: Você pode implementar `checkAccess()` usando o [componente de Role-Based Access Control \(RBAC\)](#).

11.4 Roteamento

Com as classes de recurso e controller prontas, você pode acessar os recursos utilizando uma URL como `http://localhost/index.php?r=user/create`, semelhante ao que você pode fazer com aplicações Web normais.

Na prática, normalmente você desejará utilizar URLs amigáveis e tirar proveito dos métodos HTTP. Por exemplo, uma requisição `POST /users` seria o mesmo que a ação `user/create`. Isto pode ser feito facilmente através da configuração do [componente de aplicação `urlManager`](#) conforme mostrado a seguir:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

Em comparação com o gerenciamento de URL para aplicações Web, a principal novidade acima é o uso de `yii\rest\UrlRule` para rotear requisições API RESTful. Esta classe especial criará um conjunto de regras de URL filhas para dar suporte ao roteamento e a criação de URL para o controller especificado. Por exemplo, o código acima é mais ou menos equivalente às seguintes regras:

```
[
    'PUT,PATCH users/<id>' => 'user/update',
    'DELETE users/<id>' => 'user/delete',
    'GET,HEAD users/<id>' => 'user/view',
    'POST users' => 'user/create',
    'GET,HEAD users' => 'user/index',
    'users/<id>' => 'user/options',
    'users' => 'user/options',
]
```

E as seguintes URLs (também chamadas de *endpoints*) da API são suportados por esta regra:

- GET /users: lista todos os usuários página por página;
- HEAD /users: mostrar a informações gerais da listagem de usuários;
- POST /users: cria um novo usuário;
- GET /users/123: retorna detalhes do usuário 123;
- HEAD /users/123: mostrar a informações gerais do usuário 123;
- PATCH /users/123 and PUT /users/123: atualiza o usuário 123;
- DELETE /users/123: deleta o usuário 123;
- OPTIONS /users: exibe os métodos suportados pela URL /users;
- OPTIONS /users/123: exibe os métodos suportados pela URL /users/123.

Você pode configurar as opções `only` e `except` para listar explicitamente quais ações são suportadas ou quais ações devem ser desativadas, respectivamente. Por exemplo,

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'except' => ['delete', 'create', 'update'],
],
```

Você também pode configurar `patterns` ou `extraPatterns` para redefinir padrões existentes ou adicionar novos padrões suportados por esta regra. Por exemplo, para acessar a uma nova ação `search` pela URL GET /users/search, configure a opção `extraPatterns` como a seguir,

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'extraPatterns' => [
        'GET search' => 'search',
    ],
],
```

Você deve ter notado que o ID `user` de controller aparece no plural como `users` na extremidade das URLs. Isto acontece porque `yii\rest\UrlRule` pluraliza os IDs de controllers automaticamente na criação de regras de URLs filhas. Você pode desabilitar este comportamento configurando `yii\rest\UrlRule::$pluralize` para `false`.

Observação: A pluralização dos IDs de controllers são feitas pelo método `yii\helpers\Inflector::pluralize()`. O método respeita as regras especiais de pluralização. Por exemplo, a palavra `box` será pluralizada para `boxes` em vez de `boxs`.

Caso a pluralização automática não encontre uma opção para a palavra requerida, você pode configurar a propriedade `yii\rest\UrlRule::$controller` para especificar explicitamente como mapear um nome para ser usado como uma URL para um ID de controller. Por exemplo, o seguinte código mapeia o nome `u` para o ID `user` de controller.


```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['u' => 'user'],
]
```

11.5 Formatando Respostas

Ao manipular uma requisição da API RESTful, a aplicação normalmente realiza as seguintes etapas que estão relacionadas com a formatação da resposta:

1. Determinar diversos fatores que podem afetar o formato da resposta, tais como tipo de mídia, idioma, versão, etc. Este processo também é conhecido como negociação de conteúdo (*content negotiation*)³.
2. Converter objetos de recursos em arrays, como descrito na seção [Recursos](#). Isto é feito por `yii\rest\Serializer`.
3. Converte arrays em uma string no formato como determinado pela etapa de negociação de conteúdo. Isto é feito pelos **formatadores de respostas** registrados na propriedade `formatters` do **componente de aplicação** `response`.

11.5.1 Negociação de Conteúdo

O Yii suporta a negociação de conteúdo através do filtro `yii\filters\ContentNegotiator`. A classe base de controller API RESTful `yii\rest\Controller` está equipada com este filtro sob o nome de `contentNegotiator`. O filtro fornece negociação de formato de resposta, bem como negociação de idioma. Por exemplo, se uma requisição da API RESTful tiver o seguinte cabeçalho,

```
Accept: application/json; q=1.0, */*; q=0.1
```

ele obterá uma resposta em formato JSON, como o seguinte:

```
$ curl -i -H "Accept: application/json; q=1.0, */*; q=0.1"
"http://localhost/users"
```

```
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
X-Powered-By: PHP/5.4.20
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
```

³https://en.wikipedia.org/wiki/Content_negotiation

```

Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

```

```

[
  {
    "id": 1,
    ...
  },
  {
    "id": 2,
    ...
  },
  ...
]

```

Por baixo dos panos, antes de uma ação do controlador API RESTful ser executada, o filtro `yii\filters\ContentNegotiator` verificará o `Accept` do cabeçalho HTTP na requisição e definirá o `response format` para `'json'`. Após a ação ser executada e retornar o objeto resultante de recursos ou coleção, `yii\rest\Serializer` converterá o resultado em um array. E finalmente, `yii\web\JsonResponseFormatter` irá serializar o array em uma string JSON e incluí-la no corpo da resposta.

Por padrão, APIs RESTful suportam tanto os formatos JSON quanto XML. Para suportar um novo formato, você deve configurar a propriedade `formats` do filtro `contentNegotiator` como mostrado no exemplo a seguir em suas classes do controlador da API:

```

use yii\web\Response;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['contentNegotiator']['formats']['text/html'] =
        Response::FORMAT_HTML;
    return $behaviors;
}

```

As chaves da propriedade `formats` são os tipos MIME suportados, enquanto os valores são os nomes de formato de resposta correspondentes que devem ser suportados em `yii\web\Response::$formatters`.

11.5.2 Serializando Dados

Como foi descrito acima, `yii\rest\Serializer` é a peça central responsável pela conversão de objetos de recursos ou coleções em arrays. Ele reconhece objetos que implementam a interface `yii\base\ArrayableInterface` bem

como `yii\data\DataProviderInterface`. O primeiro é aplicado principalmente pelos objetos de recursos, enquanto o último se aplica mais a coleções de recursos.

Você pode configurar o serializador, definindo a propriedade `yii\rest\Controller::$serializer` com um array de configuração. Por exemplo, às vezes você pode querer ajudar a simplificar o trabalho de desenvolvimento do cliente, incluindo informações de paginação diretamente no corpo da resposta. Para fazê-lo, configure a propriedade `yii\rest\Serializer::$collectionEnvelope` como a seguir:

```
use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
    public $serializer = [
        'class' => 'yii\rest\Serializer',
        'collectionEnvelope' => 'items',
    ];
}
```

Você pode, então, obter a seguinte resposta para a url `http://localhost/users`:

```
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
X-Powered-By: PHP/5.4.20
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8
```

```
{
    "items": [
        {
            "id": 1,
            ...
        },
        {
            "id": 2,
            ...
        },
        ...
    ],
    "_links": {
        "self": {
            "href": "http://localhost/users?page=1"
```

```

    },
    "next": {
      "href": "http://localhost/users?page=2"
    },
    "last": {
      "href": "http://localhost/users?page=50"
    }
  },
  "_meta": {
    "totalCount": 1000,
    "pageCount": 50,
    "currentPage": 1,
    "perPage": 20
  }
}

```

11.6 Autenticação

Ao contrário de aplicações Web, APIs RESTful são geralmente stateless, o que significa que as sessões ou os cookies não devem ser utilizados. Portanto, cada requisição deve vir com algum tipo de credencial de autenticação pois o estado de autenticação do usuário não pode ser mantido por sessões ou cookies. Uma prática comum é enviar um token de acesso secreto com cada solicitação para autenticar o usuário. Uma vez que um token de acesso pode ser utilizado para identificar de forma exclusiva e autenticar um usuário. **Solicitações de API devem sempre ser enviadas via HTTPS para evitar ataques man-in-the-middle (MitM).**

Existem diferentes maneiras de enviar um token de acesso:

- Autenticação Básica HTTP⁴: o token de acesso é enviado como um nome de usuário. Isso só deve ser usado quando um token de acesso puder ser armazenado com segurança no lado do consumidor da API. Por exemplo, o consumidor API é um programa executado em um servidor.
- Parâmetro de consulta da URL: o token de acesso é enviado como um parâmetro de consulta na URL da API, ex., `https://example.com/users?access-token=xxxxxxxx`. Como a maioria dos servidores Web manterão os parâmetros de consulta nos logs do servidor, esta abordagem deve ser utilizada principalmente para servir requisições JSONP que não pode usar cabeçalhos HTTP para enviar tokens de acesso.
- OAuth 2⁵: o token de acesso é obtido pelo consumidor a partir de um servidor de autorização e enviado para o servidor da API via [HTTP Bearer Tokens] (<https://datatracker.ietf.org/doc/html/rfc6750>), de acordo com o protocolo OAuth2.

⁴https://en.wikipedia.org/wiki/Basic_access_authentication

⁵<https://oauth.net/2/>

Yii suporta todos os métodos de autenticação descritos acima. Você também pode criar facilmente um novo método de autenticação.

Para ativar a autenticação nas suas APIs, siga os seguintes passos:

1. Configure o **componente de aplicação user**:
 - Defina a propriedade `enableSession` como `false`.
 - Defina a propriedade `loginUrl` como `null` para mostrar o erro HTTP 403 em vez de redirecionar para a página de login.
2. Especificar quais métodos de autenticação você planeja usar configurando o behavior `authenticator` na sua classe controller REST.
3. Implemente `yii\web\IdentityInterface::findIdentityByAccessToken()` na sua classe de identidade do usuário.

Passo 1 não é obrigatório, mas é recomendado para APIs RESTful stateless. Quando `enableSession` está marcado como falso, o status de autenticação de usuário NÃO será mantido entre as requisições usando sessões. Em lugar disso, autenticação será realizada para cada requisição, que é realizado no passo 2 e 3.

Dica: Você pode configurar `enableSession` do componente `user` nas configurações da aplicação se você estiver desenvolvendo APIs RESTful para sua aplicação. Se você desenvolver APIs RESTful como um módulo, você pode colocar a seguinte linha no método `init()` do módulo, conforme exemplo a seguir:

```
public function init()
{
    parent::init();
    \Yii::$app->user->enableSession = false;
}
```

Por exemplo, para usar autenticação HTTP básica, você pode configurar o behavior `authenticator` como o seguinte:

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::class,
    ];
    return $behaviors;
}
```

Se você quiser dar suporte a todos os três métodos de autenticação explicado acima, você pode utilizar o `CompositeAuth` conforme mostrado a seguir:

```

use yii\filters\auth\CompositeAuth;
use yii\filters\auth\HttpBasicAuth;
use yii\filters\auth\HttpBearerAuth;
use yii\filters\auth\QueryParamAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => CompositeAuth::class,
        'authMethods' => [
            HttpBasicAuth::class,
            HttpBearerAuth::class,
            QueryParamAuth::class,
        ],
    ];
    return $behaviors;
}

```

Cada elemento em `authMethods` deve ser o nome de uma classe de método de autenticação ou um array de configuração.

Implementação de `findIdentityByAccessToken()` é específico por aplicação. Por exemplo, em cenários simples quando cada usuário só pode ter um token de acesso, você pode armazenar o token de acesso em uma coluna `access_token` na tabela `user`. O método pode então ser facilmente implementado na classe `User` como o seguinte:

```

use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }
}

```

Após a autenticação ser ativada, conforme descrito acima, para todas as requisições da API, o controller requisitado irá tentar autenticar o usuário no passo `beforeAction()`.

Se a autenticação retornar com sucesso, o controller irá executar outras verificações (tais como limitação de taxa, autorização) e então executará a ação. As informações de identidade do usuário autenticado podem ser recuperadas através de `Yii::$app->user->identity`.

Se a autenticação falhar, uma resposta HTTP com status 401 será enviado de volta junto com outros cabeçalhos apropriados (tal como um `WWW-Authenticate` cabeçalho HTTP para Autenticação Básica).

11.6.1 Autorização

Após um usuário se autenticar, você provavelmente vai querer verificar se ele ou ela tem a permissão para executar a ação do recurso solicitado. Este processo é chamado de *autorização* que é tratada em pormenor na seção de [Autorização](#).

Se o seu controller estende de `yii\rest\ActiveController`, você pode sobrescrever o método `yii\rest\Controller::checkAccess()` para executar a verificação de autorização. O método será chamado pelas ações incorporadas fornecidas pelo `yii\rest\ActiveController`.

11.7 Limitador de Acesso

Para prevenir abusos, você pode considerar a utilização de um *limitador de acesso* nas suas APIs. Por exemplo, você pode querer limitar o uso da API para cada usuário em no máximo 100 chamadas a cada 10 minutos. Se o número de solicitações recebidas por usuário ultrapassar este limite, uma resposta com status 429 (significa “Muitas Requisições”) deve ser retornada.

Para habilitar o limitador de acesso, a classe de identidade do usuário deve implementar `yii\filters\RateLimitInterface`. Esta interface requer a implementação de três métodos:

- `getRateLimit()`: retorna o número máximo de pedidos permitidos e o período de tempo (ex., `[100, 600]` significa que pode haver, no máximo, 100 chamadas de API dentro de 600 segundo);
- `loadAllowance()`: retorna o número restante de pedidos permitidos e a hora da última verificação;
- `saveAllowance()`: salva tanto o número restante de requisições e a hora atual.

Você pode usar duas colunas na tabela de usuários para registrar estas informações. Com esses campos definidos, então `loadAllowance()` e `saveAllowance()` podem ser implementados para ler e guardar os valores das duas colunas correspondentes ao atual usuário autenticado. Para melhorar o desempenho, você também pode considerar armazenar essas informações em um cache ou armazenamento NoSQL.

Uma vez que a classe de identidade do usuário estiver com a interface necessária implementada, o Yii automaticamente usará a classe `yii\filters\RateLimiter` configurada como um filtro da ação para o `yii\rest\Controller` realizar a verificação da limitação do acesso. O limitador de acesso lançará uma exceção `yii\web\TooManyRequestsHttpException` quando o limite for excedido.

Você pode configurar o limitador de acesso da seguinte forma em suas classes controller REST:

```
public function behaviors()
{
```

```

$behaviors =
parent::behaviors();
$behaviors['rateLimiter']['enableRateLimitHeaders']
= false;
return $behaviors;
}

```

Quando o limitador de acesso está habilitado, por padrão a cada resposta será enviada com o seguinte cabeçalho HTTP contendo a informação da atual taxa de limitação:

- X-Rate-Limit-Limit, o número máximo permitido de pedidos em um período de tempo;
- X-Rate-Limit-Remaining, o número de pedidos restantes no período de tempo atual;
- X-Rate-Limit-Reset, o número de segundos de espera a fim de obter o número máximo de pedidos permitidos.

Você pode desativar esses cabeçalhos, configurando `yii\filters\RateLimiter::$enableRateLimitHeaders` para `false`, como mostrado no exemplo acima.

11.8 Versionamento

Uma boa API é *versionada*: Mudanças e novos recursos são implementados em novas versões da API em vez de alterar continuamente apenas uma versão. Diferente de aplicações Web, com a qual você tem total controle do código de ambos os lados cliente e servidor, APIs são destinadas a ser utilizadas por clientes além de seu controle. Por esta razão, a compatibilidade (BC) entre as APIs deve ser mantida sempre que possível. Se uma mudança que pode quebrar esta compatibilidade é necessária, você deve introduzi-la em uma nova versão de API e subir o número da versão. Os clientes existentes podem continuar a usar a versão antiga da API; e os clientes novos ou atualizados podem obter a nova funcionalidade na nova versão da API.

Dica: Consulte o artigo [Semantic Versioning](https://semver.org/)⁶ para obter mais informações sobre como projetar números de versão da API.

Uma maneira comum de implementar versionamento de API é incorporar o número da versão nas URLs da API. Por exemplo, `https://example.com/v1/users` representa o terminal `/users` da API versão 1.

Outro método de versionamento de API, que tem sido muito utilizado recentemente, é colocar o número da versão nos cabeçalhos das requisições HTTP. Isto é tipicamente feito através do cabeçalho `Accept`:

```

// Através de um parâmetro
Accept: application/json; version=v1

```

⁶<https://semver.org/>


```
// através de um vendor content type
Accept: application/vnd.company.myapp-v1+json
```

Ambos os métodos tem seus prós e contras, e há uma série de debates sobre cada abordagem. A seguir, você verá uma estratégia prática para o controle de versão de API que é uma mistura dos dois métodos:

- Coloque cada versão principal de implementação da API em um módulo separado cuja identificação é o número de versão principal (ex. v1, v2). Naturalmente, as URLs da API irão conter os números da versão principal.
- Dentro de cada versão principal (e, assim, dentro do módulo correspondente), utilize o cabeçalho `Accept` da requisição HTTP para determinar o número de versão secundária e escrever código condicional para responder às versões menores em conformidade.

Para cada módulo destinado a uma versão principal, deve incluir o recurso e a classe de controller destinados a esta versão específica. Para melhor separar a responsabilidade do código, você pode manter um conjunto comum de classes base de recursos e de controller e criar subclasses delas para cada versão individual do módulo. Dentro das subclasses, implementar o código concreto, tais como `Model::fields()`.

Seu código pode ser organizado da seguinte maneira:

```
api/
  common/
    controllers/
      UserController.php
      PostController.php
    models/
      User.php
      Post.php
  modules/
    v1/
      controllers/
        UserController.php
        PostController.php
      models/
        User.php
        Post.php
      Module.php
    v2/
      controllers/
        UserController.php
        PostController.php
      models/
        User.php
        Post.php
      Module.php
```

A configuração da sua aplicação seria algo como:

```

return [
    'modules' => [
        'v1' => [
            'class' => 'app\modules\v1\Module',
        ],
        'v2' => [
            'class' => 'app\modules\v2\Module',
        ],
    ],
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'enableStrictParsing' => true,
            'showScriptName' => false,
            'rules' => [
                ['class' => 'yii\rest\UrlRule', 'controller' => ['v1/user', 'v1/post']],
                ['class' => 'yii\rest\UrlRule', 'controller' => ['v2/user', 'v2/post']],
            ],
        ],
    ],
];

```

Como resultado do código acima, <https://example.com/v1/users> retornará a lista de usuários na versão 1, enquanto <https://example.com/v2/users> retornará a lista de usuários na versão 2.

Graças aos módulos, o código para diferentes versões principais pode ser bem isolado. Entretanto esta abordagem torna possível a reutilização de código entre os módulos através de classes bases comuns e outros recursos compartilhados.

Para lidar com números de subversões, você pode tirar proveito da negociação de conteúdo oferecida pelo behavior `contentNegotiator`. O behavior `contentNegotiator` irá configurar a propriedade `yii\web\Response::$acceptParams` que determinará qual versão é suportada.

Por exemplo, se uma requisição é enviada com o cabeçalho HTTP `Accept: application/json; version=v1`, Após a negociação de conteúdo, `yii\web\Response::$acceptParams` terá o valor `['version' => 'v1']`.

Com base na informação da versão em `acceptParams`, você pode escrever um código condicional em lugares tais como ações, classes de recursos, serializadores, etc. para fornecer a funcionalidade apropriada.

Uma vez que subversões por definição devem manter compatibilidades entre si, esperamos que não haja muitas verificações de versão em seu código. De outra forma, você pode precisar criar uma nova versão principal.

11.9 Tratamento de Erros

Ao manusear uma requisição da API RESTful, se existir um erro na requisição do usuário ou se alguma coisa inesperada acontecer no servidor, você pode simplesmente lançar uma exceção para notificar o usuário de que algo deu errado. Se você puder identificar a causa do erro (ex., o recurso requisitado não existe), você deve considerar lançar uma exceção juntamente com um código de status HTTP adequado (ex., `yii\web\NotFoundException` representa um código de status 404). O Yii enviará a resposta juntamente com o código e o texto do status HTTP correspondente. O Yii também incluirá a representação serializada da exceção no corpo da resposta. Por exemplo:

```
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
}
```

A lista a seguir descreve os códigos de status HTTP que são usados pelo framework REST do Yii:

- 200: OK. Tudo funcionou conforme o esperado;
- 201: Um recurso foi criado com êxito em resposta a uma requisição POST. O cabeçalho `location` contém a URL que aponta para o recurso recém-criado;
- 204: A requisição foi tratada com sucesso e a resposta não contém nenhum conteúdo no corpo (por exemplo uma requisição DELETE);
- 304: O recurso não foi modificado. Você pode usar a versão em cache;
- 400: Requisição malfeita. Isto pode ser causado por várias ações por parte do usuário, tais como o fornecimento de um JSON inválido no corpo da requisição, fornecendo parâmetros inválidos, etc;
- 401: Falha de autenticação;
- 403: O usuário autenticado não tem permissão para acessar o recurso da API solicitado;
- 404: O recurso requisitado não existe;
- 405: Método não permitido. Favor verificar o cabeçalho `Allow` para conhecer os métodos HTTP permitidos;
- 415: Tipo de mídia não suportada. O número de versão ou o content type requisitado são inválidos;

- 422: Falha na validação dos dados (na resposta a uma requisição POST, por exemplo). Por favor, verifique o corpo da resposta para visualizar a mensagem detalhada do erro;
- 429: Excesso de requisições. A requisição foi rejeitada devido a limitação de taxa;
- 500: Erro interno do servidor. Isto pode ser causado por erros internos do programa.

11.9.1 Customizando Resposta de Erro

Às vezes você pode querer personalizar o formato de resposta de erro padrão. Por exemplo, em vez de confiar em usar diferentes status HTTP para indicar os diversos erros, você pode querer usar sempre o status 200 como resposta e colocar o código de status real como parte da estrutura JSON da resposta, como mostrado abaixo,

```
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
  }
}
```

Para atingir este objetivo, você pode responder o evento `beforeSend` do componente `response` na configuração da aplicação:

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null &&
                    Yii::$app->request->get('suppress_response_code')) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            }
        ]
    ]
];
```

```
        },  
    ],  
    ],  
];
```

O código acima formatará a resposta (para ambas as respostas, bem-sucedidas e com falha) como explicado quando `suppress_response_code` é passado como um parâmetro GET.

Capítulo 12

Ferramentas de Desenvolvimento

Capítulo 13

Testes

13.1 Testes

O teste é uma parte importante do desenvolvimento de software. Se estamos conscientes disso ou não, Realizamos testes continuamente. Por exemplo, enquanto escrevemos uma classe PHP, podemos depurá-lo passo a passo ou simplesmente usar declarações `echo` ou `die` para verificar se a implantação está de acordo com nosso plano inicial. No caso de uma aplicação web, estamos entrando em alguns testes de dados em forma de assegurar que a página interage com a gente como esperado.

O processo de teste pode ser automatizado de modo que cada momento em que precisamos para verificar alguma coisa, só precisamos chamar o código que faz isso por nós. O código que verifica o resultado coincide com o que temos planejado é chamado de teste e o processo de sua criação e posterior execução é conhecido como teste automatizado, que é o principal tema destes capítulos de testes.

13.1.1 Desenvolvendo com testes

Test-Driven Development (TDD), and Behavior-Driven Development (BDD) são abordagens de desenvolvimento de software, descrevendo o comportamento de um trecho de código ou todo o recurso como um conjunto de cenários ou testes antes de escrever código real e só então criar a aplicação que permite que estes testes passem verificando se comportamento a que se destina é conseguido.

O processo de desenvolvimento de uma funcionalidade é a seguinte:

- Criar um novo teste que descreve uma funcionalidade a ser implementada.
- Execute o novo teste e verifique se ele falha. isto é esperado já que não há nenhuma implementação ainda.
- Escrever um código simples para fazer o novo teste passar.
- Executar todos os testes e garantir que todos eles passam.

- Melhorar código e certificar-se de testes ainda estão OK.

Depois feito o processo é repetido novamente para outras funcionalidades ou melhorias. Se uma funcionalidade existente deve ser alterada, os testes devem ser mudadas também.

Dica: Se você sentir que você está perdendo tempo fazendo um monte de pequenas e simples iterações, experimente cobrindo mais por você. Cenários de teste é para que você faça mais antes de executar testes novamente. Se você está depurando muito, tente fazer o oposto.

A razão para criar testes antes de fazer qualquer implementação é que ela nos permite focar no que queremos alcançar e totalmente mergulhar “como fazê-lo” depois. Normalmente, leva a melhores abstrações e manutenção de teste mais fácil quando se trata de ajustes na funcionalidade ou de menos componentes acoplados.

Assim, para resumir as vantagens de tal abordagem são as seguintes:

- Mantém-se focado em uma coisa de cada vez que resulta em uma melhor planejamento e implementação.
- Resultados cobertos por testes para mais funcionalidade em maior detalhe, ou seja, se os testes são OK provavelmente nada está quebrado.

No longo prazo, geralmente, dá-lhe um boa melhoria na produtividade.

Dica: Se você quiser saber mais sobre os princípios de levantamento de requisitos de software e modelagem do assunto esta é uma referência boa para aprender [Domain Driven Development (DDD)] (https://en.wikipedia.org/wiki/Domain-driven_design).

13.1.2 Quando e como testar

Enquanto a primeira abordagem de teste descrito acima faz sentido em longo prazo para projetos relativamente complexos e que seria um exagero para os mais simples. Existem alguns indicadores de quando é apropriado:

- Projeto já é grande e complexo.
- Requisitos do projeto estão começando a ficarem complexos. Projeto cresce constantemente.
- Projeto pretende ser a longo prazo.
- O custo da falha é muito alta.

Não há nada errado na criação de testes que abrangem o comportamento de implementação existente.

- Projeto é um legado para ser gradualmente renovada.
- Você tem um projeto para trabalhar e não tem testes.

Em alguns casos, qualquer forma de teste automatizado poderia ser um exagero:

- Projeto é simples e não está ficando mais complexo.

- É um projeto emporal eue deixarão de trabalhar nele.

Ainda assim, se você tiver tempo é bom automatizar testes nestes casos também.

13.1.3 Outras leituras

- Test Driven Development: By Example / Kent Beck. ISBN: 0321146530.

Error: not existing file: test-environment-setup.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

Capítulo 14

Tópicos Especiais

Error: not existing file: tutorial-start-from-scratch.md

Error: not existing file: tutorial-console.md

14.1 Validadores Nativos

O Yii fornece um conjunto de validadores nativos bastante utilizados, encontrados principalmente sob o namespace `yii\validators`. Em vez de usar nomes longos nas classes de validadores, você pode usar *alias*es para especificar o uso desses validadores. Por exemplo, você pode usar o alias `required` para referenciar a classe `yii\validators\RequiredValidator`:

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

A propriedade `yii\validators\Validator::$builtInValidators` declara todos os alias de validação suportados.

A seguir, descreveremos o uso principal e as propriedades de cada um desses validadores.

14.1.1 boolean

```
[
    // Verifica se "selected" é 0 ou 1, independentemente do tipo de dados
    ['selected', 'boolean'],

    // verifica se "deleted" é um tipo boolean, e se é verdadeiro ou falso
    ['deleted', 'boolean', 'trueValue' => true, 'falseValue' => false,
     'strict' => true],
]
```

Este validador verifica se o valor de entrada é um booleano.

- `trueValue`: o valor representando *true*. O padrão é `'1'`.
- `falseValue`: o valor representando *false*. O padrão é `'0'`.
- `strict`: se o tipo do valor de entrada deve corresponder ao `trueValue` e `falseValue`. O padrão é `false`.

Observação: Como a entrada de dados enviados através de formulários HTML são todos strings, normalmente deverá deixar a propriedade `strict` como `false`.

14.1.2 captcha

```
[
    ['verificationCode', 'captcha'],
]
```

Este validador é geralmente usado junto com `yii\captcha\CaptchaAction` e `yii\captcha\Captcha` para garantir que a entrada de dados seja igual ao código de verificação exibido pelo widget `CAPTCHA`.

- **caseSensitive**: se a comparação da verificação de código for case sensível. O padrão é `false`.
- **captchaAction**: a [rota](#) correspondente à ação CAPTCHA que renderiza as imagens. O padrão é `'site/captcha'`.
- **skipOnEmpty**: se a validação pode ser ignorada se a entrada estiver vazia. O padrão é `false`, o que significa que a entrada é obrigatória.

14.1.3 compare

```
[
  // valida se o valor do atributo "password" é igual a "password_repeat"
  ['password', 'compare'],

  // valida se a idade é maior do que ou igual a 30
  ['age', 'compare', 'compareValue' => 30, 'operator' => '>='],
]
```

Este validador compara o valor de entrada especificado com um outro e certifica se a sua relação está como especificado pela propriedade **operator**.

- **compareAttribute**: o nome do atributo cujo valor deve ser comparado. Quando o validador está sendo usado para validar um atributo, o valor padrão dessa propriedade seria o nome do atributo com o sufixo `_repeat`. Por exemplo, se o atributo que está sendo validado é `password`, então esta propriedade será por padrão `password_repeat`.
- **compareValue**: um valor constante com o qual o valor de entrada deve ser comparado. Quando esta propriedade e a propriedade **compareAttribute** forem especificadas, a propriedade **compareValue** terá precedência.
- **operator**: o operador de comparação. O padrão é `==`, ou seja, verificar se o valor de entrada é igual ao do **compareAttribute** ou **compareValue**. Os seguintes operadores são suportados:
 - `==`: verifica se dois valores são iguais. A comparação é feita no modo `non-strict`.
 - `===`: verifica se dois valores são iguais. A comparação é feita no modo `strict`.
 - `!=`: verifica se dois valores NÃO são iguais. A comparação é feita no modo `non-strict`.
 - `!==`: verifica se dois valores NÃO são iguais. A comparação é feita no modo `strict`.
 - `>`: verifica se o valor que está sendo validado é maior do que o valor que está sendo comparado.
 - `>=`: verifica se o valor que está sendo validado é maior ou igual ao valor que está sendo comparado.
 - `<`: verifica se o valor que está sendo validado é menor do que o valor que está sendo comparado.
 - `<=`: verifica se o valor que está sendo validado menor ou igual ao valor que está sendo comparado.

14.1.4 date

```
[
    [['from_date', 'to_date'], 'date'],
]
```

Este validador verifica se o valor de entrada é uma data, hora ou data e hora em um formato adequado. Opcionalmente, pode converter o valor de entrada para um UNIX timestamp ou outro formato legível e armazená-lo em um atributo especificado via `timestampAttribute`.

- **format**: o formato date/time que o valor que está sendo validado deve ter. Este pode ser um padrão de data e hora conforme descrito no [ICU manual] (https://unicode-org.github.io/icu/userguide/format_parse/datetime/#datetime-format-syntax). Alternativamente esta pode ser uma string com o prefixo `php`: representando um formato que pode ser reconhecido pela classe PHP `Datetime`. Por favor, consulte https://www.php.net/manual/pt_BR/datetime.createfromformat.php para formatos suportados. Se isso não for definido, ele terá o valor de `Yii::$app->formatter->dateFormat`. Consulte a documentação da API para mais detalhes.
- **timestampAttribute**: o nome do atributo para que este validador possa atribuir o UNIX timestamp convertido a partir da entrada de data / hora. Este pode ser o mesmo atributo que está sendo validado. Se este for o caso, valor original será substituído pelo valor timestamp após a validação. Veja a seção ["Manipulando Datas com DatePicker"] (<https://github.com/yiisoft/yii2-jui/blob/master/docs/guide/topics-date-picker.md>) para exemplos de uso.

Desde a versão 2.0.4, um formato e um fuso horário podem ser especificados utilizando os atributos `$timestampAttributeFormat` e `$timestampAttributeTimeZone`, respectivamente.

- Desde a versão 2.0.4 também é possível definir um timestamp **minimum** ou **maximum**.

Caso a entrada de dados seja opcional (preenchimento não obrigatório), você também pode querer adicionar um filtro chamado `default` para o validador de data garantir que entradas vazias sejam armazenadas com `NULL`. De outra forma você pode terminar com datas como 0000-00-00 no seu banco de dados ou 1970-01-01 no campo de entrada de um *date picker*.

```
[
    [['from_date', 'to_date'], 'default', 'value' => null],
    [['from_date', 'to_date'], 'date'],
],
```

14.1.5 default

```
[
    // configura "age" para ser null se este for vazio
```

```

    ['age', 'default', 'value' => null],

    // configura "country" para ser "USA" se este for vazio
    ['country', 'default', 'value' => 'USA'],

    // atribui "from" e "to" com uma data de 3 dias e 6 dias a partir de hoje,
    // se estiverem vazias
    [['from', 'to'], 'default', 'value' => function ($model, $attribute) {
        return date('Y-m-d', strtotime($attribute === 'to' ? '+3 days' : '+6
            days'));
    }],
]

```

Este validador não valida dados. Em vez disso, atribui um valor padrão para os atributos que estão sendo validados caso estejam vazios.

- **value:** o valor padrão ou um PHP callable que retorna o valor padrão que será atribuído aos atributos que estão sendo validados caso estejam vazios. A assinatura do PHP callable deve ser como a seguir,

```

function foo($model, $attribute) {
    // ... computar $value ...
    return $value;
}

```

Observação: Como determinar se um valor está vazio ou não é um tópico separado descrito na seção Valores Vazios.

14.1.6 double

```

[
    // verifica se o "salary" é um double
    ['salary', 'double'],
]

```

Este validador verifica se o valor de entrada é um double. É equivalente ao validador number.

- **max:** o limite superior do valor (inclusive). Se não configurado, significa que o validador não verifica o limite superior.
- **min:** o limite inferior do valor (inclusive). Se não configurado, significa que o validador não verifica o limite inferior.

14.1.7 each

Observação: Este validador está disponível desde a versão 2.0.4.

```

[
    // verifica se todas as categorias 'categoryIDs' são 'integer'
    ['categoryIDs', 'each', 'rule' => ['integer']],
]

```

Este validador só funciona com um atributo array. Ele valida *todos* os elementos do array com uma regra de validação especificada. No exemplo acima, o atributo `categoryIDs` deve ter um array e cada elemento do array será validado pela regra de validação `integer`.

- **rule:** um array especificando as regras de validação. O primeiro elemento do array determina o nome da classe ou o alias do validador. O restante dos pares nome-valor no array são utilizados para configurar o objeto do validador.
- **allowMessageFromRule:** se pretende usar a mensagem de erro retornada pela regra de validação incorporada. Padrão é `true`. Se for `false`, ele usará `message` como a mensagem de erro.

Observação: Se o valor do atributo não for um array, a validação será considerada como falha e a `mensagem` será retornada como erro.

14.1.8 email

```
[
    // verifica se o "email" é um endereço de e-mail válido
    ['email', 'email'],
]
```

Este validador verifica se o valor de entrada é um endereço de email válido.

- **allowName:** permitir nome no endereço de email (ex. John Smith <john.smith@example.com>). O padrão é `false`;
- **checkDNS,** para verificar se o domínio do e-mail existe e tem tanto um A ou registro MX. Esteja ciente de que esta verificação pode falhar devido a problemas de DNS temporários, mesmo se o endereço de e-mail for realmente válido. O padrão é `false`;
- **enableIDN,** se o processo de validação deve verificar uma conta IDN (internationalized domain names). O padrão é `false`. Observe que para usar a validação IDN você deve instalar e habilitar a extensão PHP `intl`, caso contrário uma exceção será lançada.

14.1.9 exist

```
[
    // a1 precisa existir na coluna representada pelo atributo "a1"
    ['a1', 'exist'],

    // a1 precisa existir, mas seu valor usará a2 para verificar a existência
    ['a1', 'exist', 'targetAttribute' => 'a2'],

    // a1 e a2 precisam existir juntos, e ambos receberão mensagem de erro
    [['a1', 'a2'], 'exist', 'targetAttribute' => ['a1', 'a2']],

    // a1 e a2 precisam existir juntos, somente a1 receberá mensagem de erro
```



```

['a1', 'exist', 'targetAttribute' => ['a1', 'a2']],

// a1 precisa existir, verificando a existência de ambos A2 e A3 (usando o
// valor de a1)
['a1', 'exist', 'targetAttribute' => ['a2', 'a1' => 'a3']],

// a1 precisa existir. Se a1 for um array, então todos os seus elementos
// devem existir.
['a1', 'exist', 'allowArray' => true],
]

```

Este validador verifica se o valor de entrada pode ser encontrado em uma coluna representada por um atributo [Active Record](#). Você pode usar `targetAttribute` para especificar o atributo [Active Record](#) e `targetClass` a classe [Active Record](#) correspondente. Se você não especificá-los, eles receberão os valores do atributo e a classe model (modelo) que está sendo validada.

Você pode usar este validador para validar uma ou várias colunas (ex., a combinação de múltiplos valores de atributos devem existir).

- **targetClass:** o nome da classe [Active Record](#) que deve ser usada para procurar o valor de entrada que está sendo validado. Se não for configurada, a atual classe do model (modelo) que está sendo validado será usada.
- **targetAttribute:** o nome do atributo em `targetClass` que deve ser utilizado para validar a existência do valor de entrada. Se não for configurado, será usado o nome do atual atributo que está sendo validado. Você pode utilizar um array para validar a existência de múltiplas colunas ao mesmo tempo. Os valores do array são os atributos que serão utilizados para validar a existência, enquanto as chaves são os atributos cujos valores devem ser validados. Se a chave e o valor forem os mesmos, você pode especificar apenas o valor.
- **filter:** filtro adicional para ser aplicado na consulta do banco de dados utilizada para verificar a existência do valor de entrada. Pode ser uma string ou um array representando a condição da consulta adicional (consulte o formato de condição `yii\db\Query::where()`), ou uma função anônima com a assinatura `function ($query)`, onde `$query` é o objeto `Query` que você pode modificar.
- **allowArray:** se permitir que o valor de entrada seja um array. Padrão é `false`. Se esta propriedade for definida como `true` e a entrada for um array, então, cada elemento do array deve existir na coluna destinada. Observe que essa propriedade não pode ser definida como `true` se você estiver validando várias colunas configurando `targetAttribute` como um array.

14.1.10 file

```
[
    // verifica se "primaryImage" é um arquivo de imagem carregado no formato
    // PNG, JPG ou GIF.
    // o tamanho do arquivo deve ser inferior a 1MB
    ['primaryImage', 'file', 'extensions' => ['png', 'jpg', 'gif'], 'maxSize'
    => 1024*1024],
]
```

Este validador verifica se o dados de entrada é um arquivo válido.

- **extensions**: uma lista de extensões de arquivos que são permitidos para upload. Pode ser utilizado tanto um array quanto uma string constituída de extensões de arquivos separados por espaços ou por vírgulas (Ex. “gif, jpg”). Os nomes das extensões são case-insensitive. O padrão é `null`, significa que todas as extensões são permitidas.
- **mimeType**: uma lista de tipos de arquivos MIME que são permitidos no upload. Pode ser utilizado tanto um array quanto uma string constituída de tipos MIME separados por espaços ou por vírgulas (ex. “image/jpeg, image/png”). Os nomes dos tipos MIME são case-insensitive. O padrão é `null`, significa que todos os tipos MIME são permitidos. Para mais detalhes, consulte o artigo [common media types](#)¹.
- **minSize**: o número mínimo de bytes exigido para o arquivo carregado. O padrão é `null`, significa não ter limite mínimo.
- **maxSize**: o número máximo de bytes exigido para o arquivo carregado. O padrão é `null`, significa não ter limite máximo.
- **maxFiles**: o número máximo de arquivos que o atributo pode receber. O padrão é 1, ou seja, a entrada de dados deve ser composto de um único arquivo. Se o **maxFiles** for maior que 1, então a entrada de dados deve ser composto por um array constituído de no máximo **maxFiles** arquivos.
- **checkExtensionByMimeType**: verificação da extensão do arquivo por tipo MIME do arquivo. Se a extensão produzido pela verificação do tipo MIME difere da extensão do arquivo carregado, o arquivo será considerado inválido. O padrão é `true`, o que significa realizar tal verificação.

`FileValidator` é usado junto com `yii\web\UploadedFile`. Consulte a seção [Upload de Arquivos](#) para mais informações sobre o upload de arquivos e de uma validação sobre os arquivos carregados.

14.1.11 filter

```
[
    // trima as entradas "username" e "email"
    [['username', 'email'], 'filter', 'filter' => 'trim', 'skipOnArray' =>
    true],
]
```

¹https://pt.wikipedia.org/wiki/Tipo_de_m%C3%ADdia_da_Internet

```
// normaliza a entrada "phone"
['phone', 'filter', 'filter' => function ($value) {
    // normaliza a entrada phone aqui
    return $value;
}],
]
```

Este validador não valida dados. Em vez disso, aplica um filtro no valor de entrada e retorna para o atributo que está sendo validado.

- **filter**: um PHP callback que define um filtro. Pode ser um nome de função global, uma função anônima, etc. A assinatura da função deve ser `function ($value) { return $newValue; }`. Esta propriedade deve ser definida.
- **skipOnArray**: para ignorar o filtro se o valor de entrada for um array. O padrão é `false`. Observe que se o filtro não puder manipular a entrada de array, você deve configurar esta propriedade como `true`. De outra forma algum erro do PHP deve ocorrer.

Dica: Se você quiser trinar valores de entrada, você deve utilizar o validador `trim`.

Dica: Existem várias funções PHP que tem a assinatura esperada para o callback do `filter`. Por exemplo, para aplicar a conversão de tipos (usando por exemplo `intval`², `boolval`³, ...) para garantir um tipo específico para um atributo, você pode simplesmente especificar os nomes das funções do filtro sem a necessidade de envolvê-los em um closure:

```
['property', 'filter', 'filter' => 'boolval'],
['property', 'filter', 'filter' => 'intval'],
```

14.1.12 image

```
[
    // verifica se "primaryImage" é uma imagem válida com as proporções
    // adequadas
    ['primaryImage', 'image', 'extensions' => 'png, jpg',
     'minWidth' => 100, 'maxWidth' => 1000,
     'minHeight' => 100, 'maxHeight' => 1000,
    ],
]
```

Este validador verifica se o valor de entrada representa um arquivo de imagem válido. Por estender o validador `file`, ele herda todas as suas propriedades. Além disso, suporta as seguintes propriedades adicionais específicas para fins de validação de imagem:

²https://www.php.net/manual/pt_BR/function.intval.php

³https://www.php.net/manual/pt_BR/function.boolval.php

- **minWidth**: a largura mínima da imagem. O padrão é `null`, significa não ter limite mínimo.
- **maxWidth**: a largura máxima da imagem. O padrão é `null`, significa não ter limite máximo.
- **minHeight**: a altura mínima da imagem. O padrão é `null`, significa não ter limite mínimo.
- **maxHeight**: a altura máxima da imagem. O padrão é `null`, significa não ter limite máximo.

14.1.13 in

```
[  
  // verifica se o "level" é 1, 2 ou 3  
  ['level', 'in', 'range' => [1, 2, 3]],  
]
```

Este validador verifica se o valor de entrada pode ser encontrado entre os valores da lista fornecida.

- **range**: uma lista de determinados valores dentro da qual o valor de entrada deve ser procurado.
- **strict**: se a comparação entre o valor de entrada e os valores dados devem ser strict (o tipo e o valor devem ser idênticos). O padrão é `false`.
- **not**: se o resultado de validação deve ser invertido. O padrão é `false`. Quando esta propriedade é definida como `true`, o validador verifica se o valor de entrada NÃO está entre os valores da lista fornecida.
- **allowArray**: para permitir que o valor de entrada seja um array. Quando esta propriedade é marcada como `true` e o valor de entrada é um array, todos os elementos neste array devem ser encontrados na lista de valores fornecida, caso contrário a validação falhará.

14.1.14 integer

```
[  
  // verifica se "age" é um inteiro  
  ['age', 'integer'],  
]
```

Este validador verifica se o valor de entrada é um inteiro.

- **max**: limite máximo (inclusive) do valor. Se não for configurado, significa que não tem verificação de limite máximo.
- **min**: o limite mínimo (inclusive) do valor. Se não for configurado, significa que não tem verificação de limite mínimo.

14.1.15 match

```
[  
  // verifica se "username" começa com uma letra e contém somente  
  // caracteres  
  ['username', 'match', 'pattern' => '/^[a-z]\w*$/i']  
]
```

Este validador verifica se o valor de entrada atende a expressão regular especificada.

- **pattern**: a expressão regular que o valor de entrada deve corresponder. Esta propriedade deve ser configurada, caso contrário uma exceção será lançada.
- **not**: para inverter o resultado da validação. O padrão é `false`, significa que a validação terá sucesso apenas se o valor de entrada corresponder ao padrão definido. Se for configurado como `true` a validação terá sucesso apenas se o valor de entrada NÃO corresponder ao padrão definido.

14.1.16 number

```
[  
  // verifica se "salary" é um number  
  ['salary', 'number'],  
]
```

Este validador verifica se o valor de entrada é um `number`. É equivalente ao validador `double`.

- **max**: limite máximo (inclusive) do valor. Se não for configurado, significa que não tem verificação de limite máximo.
- **min**: o limite mínimo (inclusive) do valor. Se não for configurado, significa que não tem verificação de limite mínimo.

14.1.17 required

```
[  
  // verifica se ambos "username" e "password" não estão vazios  
  [['username', 'password'], 'required'],  
]
```

Este validador verifica se o valor de entrada foi fornecido e não está vazio.

- **requiredValue**: o valor desejado que a entrada deve ter. Se não configurado, significa que o valor de entrada apenas não deve estar vazio.
- **strict**: para verificar os tipos de dados ao validar um valor. O padrão é `false`. Quando **requiredValue** não é configurado, se esta propriedade for `true`, o validador verificará se o valor de entrada não é estritamente nulo; Se esta propriedade for `false`, o validador usará uma regra solta para determinar se o valor está vazio ou não. Quando **requiredValue**

está configurado, a comparação entre o valor de entrada e `requiredValue` também verificará os tipos de dados se esta propriedade for `true`.

Observação: Como determinar se um valor está vazio ou não é um tópico separado descrito na seção Valores Vazios.

14.1.18 `safe`

```
[  
  // marca o "description" como um atributo seguro  
  ['description', 'safe'],  
]
```

Este validador não executa validação de dados. Em vez disso, ele é usado para marcar um atributo para ser um [atributo seguro](#).

14.1.19 `string`

```
[  
  // verifica se "username" é uma string cujo tamanho está entre 4 e 24  
  ['username', 'string', 'length' => [4, 24]],  
]
```

Este validador verifica se o valor de entrada é uma string válida com um determinado tamanho.

- `length`: especifica o limite do comprimento da string de entrada que está sendo validada. Este pode ser especificado em uma das seguintes formas:
 - um inteiro: o comprimento exato que a string deverá ter;
 - um array de um elemento: o comprimento mínimo da string de entrada (ex. `[8]`). Isso substituirá `min`.
 - um array de dois elementos: o comprimento mínimo e máximo da string de entrada (ex. `[8, 128]`). Isso substituirá ambos `min` e `max`.
- `min`: o comprimento mínimo da string de entrada. Se não configurado, significa não ter limite para o comprimento mínimo.
- `max`: o comprimento máximo da string de entrada. Se não configurado, significa não ter limite para o comprimento máximo.
- `encoding`: a codificação da string de entrada a ser validada. se não configurado, será usado o valor de `charset` da aplicação que por padrão é UTF-8.

14.1.20 `trim`

```
[  
  // trima os espaços em branco ao redor de "username" e "email"  
  [['username', 'email'], 'trim'],  
]
```

Este validador não executa validação de dados. Em vez disso, ele vai retirar os espaços em branco ao redor do valor de entrada. Observe que se o valor de entrada for um array, ele será ignorado pelo validador.

14.1.21 unique

```
[
    // a1 precisa ser único na coluna representada pelo atributo "a1"
    ['a1', 'unique'],

    // a1 precisa ser único, mas a coluna a2 será usada para verificar a
    // singularidade do valor de a1
    ['a1', 'unique', 'targetAttribute' => 'a2'],

    // a1 e a2 precisam ser únicos, e ambos receberão mensagem de erro
    [['a1', 'a2'], 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a1 e a2 precisam ser únicos, mas somente 'a1' receberá mensagem de
    // erro
    ['a1', 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a1 precisa ser único verificando a singularidade de ambos a2 e a3
    // (usando o valor de a1)
    ['a1', 'unique', 'targetAttribute' => ['a2', 'a1' => 'a3']],
]
```

Este validador verifica se o valor de entrada é único na coluna da tabela. Ele só trabalha com atributos dos modelos (modelos) [Active Record](#). Suporta a validação de uma única coluna ou de várias.

- **targetClass**: o nome da classe [Active Record](#) que deve ser usada para procurar o valor de input que está sendo validado. Se não for configurado, a classe model atual que está sendo validado será usada.
- **targetAttribute**: o nome do atributo em **targetClass** que deve ser usado para validar a singularidade do valor de entrada. Se não for configurado, este usará o nome do atributo atual que está sendo validado. Você pode usar um array para validar a singularidade de várias colunas ao mesmo tempo. Os valores do array são os atributos que serão utilizados para validar a singularidade, enquanto as chaves do array são os atributos cujos valores serão validados. Se a chave e o valor forem os mesmos, você pode apenas especificar o valor.
- **filter**: filtro adicional para ser aplicado na query do banco de dados para validar a singularidade do valor de entrada. Este pode ser uma string ou um array representando a condição adicional da query (consulte o formato de condição `yii\db\Query::where()`) ou uma função anônima com a assinatura `function ($query)`, onde `$query` é o objeto `Query` que você pode modificar na função.

14.1.22 url

```
[  
    // verifica se "website" é uma URL válida. Coloca "http://" no atributo  
    "website"  
    // e ele não tiver um esquema da URI  
    ['website', 'url', 'defaultScheme' => 'http'],  
]
```

Este validador verifica se o valor de entrada é uma URL válida.

- **validSchemes**: um array especificando o esquema da URI que deve ser considerada válida. O padrão é ['http', 'https'], significa que ambas URLs http e https são considerados como válidos.
- **defaultScheme**: o esquema padrão da URI para ser anexado à entrada, se a parte do esquema não for informada na entrada. O padrão é null, significa que o valor de entrada não será modificado.
- **enableIDN**: se o validador deve ter uma conta IDN (internationalized domain names). O padrão é false. Observe que para usar a validação IDN você tem que instalar e ativar a extensão PHP intl, caso contrário uma exceção será lançada.

14.2 Yii e Docker

Para o desenvolvimento e implantação de aplicativos Yii, eles podem ser executados como contêineres Docker. Um contêiner é como uma máquina virtual isolada e leve que mapeia seus serviços para as portas do host, ou seja, um servidor da web em um contêiner na porta 80 está disponível na porta 8888 do seu (local) host.

Os contêineres podem resolver muitos problemas, como ter versões idênticas de software no computador do desenvolvedor e no servidor, implantações rápidas ou simulação de arquitetura multi-servidor durante o desenvolvimento.

Você pode ler mais sobre contêineres Docker em [docker.com](https://www.docker.com)⁴.

14.2.1 Requisitos

- docker
- docker-compose

Visite a página de download⁵ para obter as ferramentas do Docker.

14.2.2 Instalação

Após a instalação, você deve ser capaz de executar o comando `docker ps` e ver uma saída semelhante a esta:

⁴<https://www.docker.com/why-docker>

⁵<https://www.docker.com/products/container-runtime>

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
--------------	-------	---------	---------	--------	-------

Isso significa que o seu daemon Docker está em execução.

Além disso, execute o comando `docker-compose version`, a saída deve ser semelhante a esta:

```
docker-compose version 1.20.0, build unknown
docker-py version: 3.1.3
CPython version: 3.6.4
OpenSSL version: OpenSSL 1.1.0g  2 Nov 2017
```

Com o Compose, você pode configurar e gerenciar todos os serviços necessários para a sua aplicação, como bancos de dados e cache.

14.2.3 Recursos

- As imagens base do PHP para Yii podem ser encontradas em `yii2-docker`⁶
- Suporte do Docker para `yii2-app-basic`⁷
- Suporte do Docker para `yii2-app-advanced`⁸ está em desenvolvimento

14.2.4 Uso

Os comandos básicos do Docker são

```
docker-compose up -d
```

para iniciar todos os serviços em sua pilha, em segundo plano

```
docker-compose ps
```

para listar os serviços em execução

```
docker-compose logs -f
```

para visualizar os logs de todos os serviços continuamente

```
docker-compose stop
```

para interromper todos os serviços em sua pilha de forma elegante

```
docker-compose kill
```

para interromper todos os serviços em sua pilha imediatamente

```
docker-compose down -v
```

⁶<https://github.com/yiisoft/yii2-docker>

⁷<https://github.com/yiisoft/yii2-app-basic#install-with-docker>

⁸<https://github.com/yiisoft/yii2-app-advanced/pull/347>

para parar e remover todos os serviços, **atenção à perda de dados ao não usar volumes do host**

Para executar comandos em um contêiner:

```
docker-compose run --rm php composer install
```

executa a instalação do Composer em um novo contêiner

```
docker-compose exec php bash
```

executa um shell bash em um serviço php que está em *execução*.

14.2.5 Tópicos avançados

Testes do framework Yii

Você pode executar os testes do framework Yii em um contêiner Docker, conforme descrito aqui⁹.

Database administration tools

Ao executar o MySQL como (`mysql`), você pode adicionar um contêiner do phpMyAdmin à sua pilha, como mostrado abaixo:

```
phpmyadmin:
  image: phpmyadmin/phpmyadmin
  ports:
    - '8888:80'
  environment:
    - PMA_ARBITRARY=1
    - PMA_HOST=mysql
  depends_on:
    - mysql
```

⁹<https://github.com/yiisoft/yii2/blob/master/tests/README.md#dockerized-testing>

Error: not existing file: tutorial-i18n.md

Error: not existing file: tutorial-mailing.md

Error: not existing file: tutorial-performance-tuning.md

14.3 Ambiente de Hospedagem Compartilhada

Ambientes de hospedagem compartilhada geralmente são muito limitados com relação a configuração e estrutura de diretórios. Ainda assim, na maioria dos casos, você pode executar Yii 2.0 em um ambiente de hospedagem compartilhada com poucos ajustes.

14.3.1 Implantação do Template Básico

Uma vez que em um ambiente de hospedagem compartilhada geralmente não há apenas um webroot, use o template básico, se puder. Consulte o [Capítulo Instalando o Yii](#) e instale o modelo de projeto básico localmente. Depois de ter sua aplicação funcionando localmente, vamos fazer alguns ajustes para que possa ser hospedado em seu servidor de hospedagem compartilhada.

Renomear webroot

Ao conectar no seu servidor compartilhado através de FTP ou outros meios, você provavelmente verá algo como a seguir:

```
config
logs
www
```

No exemplo acima, `www` é seu diretório raiz do servidor web e pode possuir nomes diferente. Nomes comuns são: `www`, `htdocs`, e `public_html`.

O diretório raiz de nosso template básico é chamado `web`. Antes de enviar a aplicação para seu servidor renomeie seu diretório raiz local de acordo com o do seu servidor, ou seja, de `web` para `www`, `public_html` ou qualquer que seja o nome do seu diretório raiz na hospedagem.

Diretório raiz FTP precisa ser gravável

Se você possui permissão de escrita no diretório raiz, (onde estão `config`, `logs` e `www`), então faça upload de `assets`, `commands` etc.

Recursos extras para servidor web

Se o seu servidor web é Apache você precisará adicionar um arquivo `.htaccess` com o seguinte conteúdo em `web` (ou `public_html` ou qualquer outro) (onde o arquivo `index.php` está localizado):

```
Options +FollowSymLinks
IndexIgnore */*

RewriteEngine on
```

```
# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

Se o seu servidor web é Nginx, você não precisa de nenhuma configuração extra.

Verifique os requisitos

Para executar Yii, o seu servidor web deve atender alguns requisitos. O requisito mínimo é PHP 5.4. Para verificar os requisitos copie o arquivo `requirements.php` da raiz da aplicação para o diretório raiz do servidor web e execute-o através do navegador usando o endereço `https://example.com/requirements.php`. Não se esqueça de apagar o arquivo depois.

14.3.2 Implantação do Template Avançado

A implantação do Template Avançado para a hospedagem compartilhada é um pouco mais complicada do que o Template Básico, porque ele tem duas webroots, que servidores web da hospedagem compartilhada não suportam. Vamos precisar de ajustar a estrutura de diretórios.

Mova os scripts de entrada para única raiz

Primeiro de tudo, precisamos de um diretório raiz. Crie um novo diretório e nomeie-o para corresponder à raiz de sua hospedagem, conforme descrito no Renomear webroot, por exemplo `www` ou `public_html` ou semelhante. Em seguida, crie a seguinte estrutura onde `www` é o diretório raiz de sua hospedagem que você acabou de criar:

```
www
  admin
  backend
  common
  console
  environments
  frontend
  ...
```

`www` será nosso diretório frontend, então mova o conteúdo de `frontend/web` para ele. Mova o conteúdo de `backend/web` para `www/admin`. Em ambos os casos você precisará ajustar os caminhos em `index.php` and `index-test.php`.

Sessões e cookies separados

Originalmente, o backend e frontend destinam-se a ser executado em diferentes domínios. Quando movemos tudo para o mesmo domínio, tanto frontend quanto backend estarão partilhando os mesmos cookies, criando um problema. Para corrigi-lo, ajuste sua configuração backend `backend/config/main.php` da seguinte forma:

```
'components' => [
    'request' => [
        'csrfParam' => '_backendCSRF',
        'csrfCookie' => [
            'httpOnly' => true,
            'path' => '/admin',
        ],
    ],
    'user' => [
        'identityCookie' => [
            'name' => '_backendIdentity',
            'path' => '/admin',
            'httpOnly' => true,
        ],
    ],
    'session' => [
        'name' => 'BACKENDSESSID',
        'cookieParams' => [
            'path' => '/admin',
        ],
    ],
],
```


Error: not existing file: tutorial-template-engines.md

14.4 Trabalhando com Códigos de Terceiros

De tempos em tempos, você pode precisar usar algum código de terceiro na sua aplicação Yii. Ou você pode querer utilizar o Yii como uma biblioteca em alguns sistemas de terceiros. Nesta seção, vamos mostrar como fazer isto.

14.4.1 Usando Bibliotecas de Terceiros no Yii

Para utilizar bibliotecas de terceiros em uma aplicação Yii, você precisa primeiramente garantir que as classes na biblioteca estão devidamente incluídas ou se podem ser carregadas por demanda.

Usando Pacotes Composer

Muitas bibliotecas de terceiros gerenciam suas versões através de pacotes do Composer¹⁰. Você pode instalar tais bibliotecas realizando os dois seguintes passos:

1. Modifique o arquivo `composer.json` da sua aplicação e informe quais pacotes Composer você deseja instalar.
2. Execute `composer install` para instalar os pacotes especificados.

As classes nos pacotes Composer instalados podem ser carregadas automaticamente usando o autoloader do Composer. Certifique-se que o `script de entrada` da sua aplicação contém as seguintes linhas para instalar o autoloader do Composer:

```
// Instala o Composer autoloader
require __DIR__ . '/../vendor/autoload.php';

// faz o include da classe Yii
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';
```

Usando Bibliotecas baixadas

Se a biblioteca não foi lançada como um pacote Composer, você deve seguir as instruções de instalação para instalá-la. Na maioria dos casos, você precisará baixar manualmente o arquivo de liberação da biblioteca e descompactá-lo no diretório `BasePath/vendor`, onde `BasePath` representa o `caminho base` da sua aplicação.

Se uma biblioteca possui o seu próprio carregador automático, você pode instalá-la no `script de entrada` de sua aplicação. Recomenda-se que a instalação seja feita antes de incluir o arquivo `Yii.php`. Isto porque a classe autoloader Yii pode ter precedência nas classes de carregamento automático da biblioteca a ser instalada.

¹⁰<https://getcomposer.org/>

Se uma biblioteca não oferece um carregador automático de classe, mas seus nomes seguem o padrão PSR-4¹¹, você pode usar a classe de autoloader do Yii para carregar as classes. Tudo que você precisa fazer é apenas declarar um *alias* para cada namespace raiz utilizados em suas classes. Por exemplo, suponha que você tenha instalado uma biblioteca no diretório `vendor/foo/bar` e as classes de bibliotecas estão sob o namespace raiz `xyz`. Você pode incluir o seguinte código na configuração da sua aplicação:

```
[
    'aliases' => [
        '@xyz' => '@vendor/foo/bar',
    ],
]
```

Se não for nenhuma das opções acima, é provável que a biblioteca necessite fazer um `include PHP` com algum caminho específico para localizar corretamente os arquivos das classes. Basta seguir as instruções de como configurar o *include path* do PHP.

No pior dos casos, quando a biblioteca exige explicitamente a inclusão de cada arquivo de classe, você pode usar o seguinte método para incluir as classes por demanda:

- Identificar quais as classes da biblioteca contém.
- Liste as classes e os caminhos dos arquivos correspondentes em `Yii::$classMap` no script de entrada da aplicação. Por exemplo:

```
Yii::$classMap['Class1'] = 'path/to/Class1.php';
Yii::$classMap['Class2'] = 'path/to/Class2.php';
```

14.4.2 Usando o Yii em Sistemas de Terceiros

Como o Yii fornece muitas características excelentes, algumas vezes você pode querer utilizar algumas destas características como suporte ao desenvolvimento ou melhorias em sistemas de terceiros, tais como WordPress, Joomla ou aplicações desenvolvidas utilizando outros frameworks PHP. Por exemplo, você pode querer utilizar a classe `yii\helpers\ArrayHelper` ou usar o recurso de *Active Record* em um sistema de terceiros. Para alcançar este objetivo, você primeiramente precisa realizar dois passos: instale o Yii, e o bootstrap Yii.

Se o sistema em questão utilizar o Composer para gerenciar suas dependências, você pode simplesmente executar o seguinte comando para instalar o Yii:

```
composer global require "fxp/composer-asset-plugin:~1.4.1"
composer require yiisoft/yii2
composer install
```

¹¹<https://www.php-fig.org/psr/psr-4/>

O primeiro comando instala o Composer asset plugin¹² que permite gerenciar o bower e dependências de pacotes npm através do Composer. Mesmo que você apenas queira utilizar a camada de banco de dados ou outros recursos não-ativos relacionados do Yii, isto é necessário para instalar o pacote Composer do Yii. Veja também a seção [sobre a instalação do Yii](#) para obter mais informações do Composer e solução para os possíveis problemas que podem surgir durante a instalação.

Caso contrário, você pode fazer o download¹³ do Yii e descompactá-lo no diretório `BasePath/vendor`.

Em seguida, você deve modificar o script de entrada do sistema de terceiros incluindo o seguinte código no início:

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

$yiiConfig = require __DIR__ . '/../config/yii/web.php';
new yii\web\Application($yiiConfig); // NÃO execute o método run() aqui
```

Como você pode ver, o código acima é muito semelhante ao que existe no [script de entrada](#) de uma aplicação típica do Yii. A única diferença é que depois que a instância da aplicação é criada, o método `run()` não é chamado. Isto porque chamando `run()`, Yii vai assumir o controle do fluxo das requisições que não é necessário neste caso e já é tratado pela aplicação existente.

Como na aplicação Yii, você deve configurar a instância da aplicação com base no ambiente de execução do sistema em questão. Por exemplo, para usar o recurso de [Active Record](#), você precisa configurar o [componente da aplicação](#) `db` com a configuração de conexão do banco de dados utilizada pelo sistema.

Agora você pode usar a maioria dos recursos fornecidos pelo Yii. Por exemplo, você pode criar classes Active Record e usá-las para trabalhar com o banco de dados.

14.4.3 Usando Yii 2 com Yii 1

Se você estava usando o Yii 1 anteriormente, é provável que você tenha uma aplicação rodando com Yii 1. Em vez de reescrever toda a aplicação em Yii 2, você pode apenas querer melhorá-lo usando apenas alguns dos recursos disponíveis no Yii 2. Isto pode ser alcançado seguindo as instruções a seguir.

Observação: Yii 2 requer PHP 5.4 ou superior. Você deve certificar-se que o seu servidor e a sua aplicação suportem estes requisitos.

Primeiro, instale o Yii 2 na sua aplicação existente seguindo as instruções dadas na última subseção.

¹²<https://github.com/fxpio/composer-asset-plugin>

¹³<https://www.yiiframework.com/download/>

Segundo, altere o script de entrada da sua aplicação como a seguir,

```
// incluir a classe Yii personalizado descrito abaixo
require __DIR__ . '/../components/Yii.php';

// configuração para aplicação Yii 2
$yii2Config = require __DIR__ . '/../config/yii2/web.php';
new yii\web\Application($yii2Config); // NÃO execute o método run() aqui

// configuração para aplicação Yii 1
$yii1Config = require __DIR__ . '/../config/yii1/main.php';
Yii::createWebApplication($yii1Config)->run();
```

Uma vez que ambos Yii 1 e Yii 2 possuem a classe Yii, você deve criar uma versão personalizada para combiná-los. O código acima inclui o arquivo de classe personalizado Yii, que pode ser criado conforme o exemplo abaixo.

```
$yii2path = '/path/to/yii2';
require $yii2path . '/BaseYii.php'; // Yii 2.x

$yii1path = '/path/to/yii1';
require $yii1path . '/YiiBase.php'; // Yii 1.x

class Yii extends \yii\BaseYii
{
    // copie e cole o código de YiiBase (1.x) aqui
}

Yii::$classMap = include($yii2path . '/classes.php');
// registrar o autoloader do Yii 2 através do Yii 1
Yii::registerAutoloader(['Yii', 'autoload']);
// criar o contêiner de injeção de dependência
Yii::$container = new yii\di\Container;
```

Isto é tudo! Agora, em qualquer parte do seu código, você pode usar `Yii::$app` para acessar a instância da aplicação Yii 2, enquanto `Yii::app()` lhe dará a instância da aplicação Yii 1:

```
echo get_class(Yii::app()); // retorna 'CWebApplication'
echo get_class(Yii::$app);  // retorna 'yii\web\Application'
```

Error: not existing file: tutorial-yii-as-micro-framework.md

Capítulo 15

Widgets

Capítulo 16

Helpers - Funções Auxiliares

16.1 Helpers

Observação: Esta seção está em desenvolvimento.

O Yii oferece muitas classes que ajudam a simplificar as tarefas comuns de codificação, como manipulação de string ou de array, geração de código HTML, e assim por diante. Essas classes helpers (auxiliares) são organizadas no namespace `yii\helpers` e são todas classes estáticas (o que significa que contêm apenas propriedades e métodos estáticos e não devem ser instanciadas).

Você usa uma classe helper chamando diretamente um de seus métodos estáticos, como o seguinte:

```
use yii\helpers\Html;

echo Html::encode('Test > test');
```

Observação: Para oferecer suporte à personalização de classes helper, o Yii divide cada classe helper principal em duas classes: uma classe base (ex. `BaseArrayHelper`) e uma classe concreta (ex. `ArrayHelper`). Ao usar um helper, você deve usar apenas a versão concreta e nunca a classe base.

16.1.1 Principais Classes Helper

As seguintes classes helper são fornecidas nas versões Yii:

- [ArrayHelper](#)
- Console
- FileHelper
- FormatConverter
- [Html](#)
- HtmlPurifier
- Imagine (fornecido pela extensão yii2-imagine)

- Inflector
- Json
- Markdown
- StringHelper
- [Url](#)
- VarDumper

16.1.2 Personalização de Classes Helper

Para personalizar uma classe helper principal (ex. `yii\helpers\ArrayHelper`), você deve criar uma nova classe que estende da classe base correspondente ao helper (ex. `yii\helpers\BaseArrayHelper`) e nomear a sua classe da mesma forma que a classe concreta correspondente (ex. `yii\helpers\ArrayHelper`). Essa classe será então configurada para substituir a implementação original da estrutura.

O exemplo a seguir mostra como personalizar o método `merge()` da classe `yii\helpers\ArrayHelper`:

```
<?php

namespace yii\helpers;

class ArrayHelper extends BaseArrayHelper
{
    public static function merge($a, $b)
    {
        // sua implementação personalizada
    }
}
```

Salve sua classe em um arquivo chamado `ArrayHelper.php`. O arquivo pode estar em qualquer diretório, por exemplo `@app/components`.

Em seguida, no [script de entrada](#) da sua aplicação, adicione a seguinte linha de código depois do `include` do arquivo `yii.php` para dizer ao [autoloader de classes do Yii](#) para carregar sua classe personalizada em vez da classe Helper original do framework:

```
Yii::$classMap['yii\helpers\ArrayHelper'] =
    '@app/components/ArrayHelper.php';
```

Observe que personalizar as classes Helper só é útil se você quiser mudar o comportamento de uma função existente dos Helpers. Se você deseja adicionar outras funções para usar em sua aplicação, é melhor criar um Helper para isso.

Error: not existing file: helper-array.md

Error: not existing file: helper-html.md

16.2 URL Helper

URL helper fornece um conjunto de métodos estáticos para o gerenciamento de URLs.

16.2.1 Obtendo URLs comuns

Há dois métodos que você pode usar para obter URLs comuns: URL home e URL base para a requisição corrente. Para obter a URL home, use o seguinte:

```
$relativeHomeUrl = Url::home();  
$absoluteHomeUrl = Url::home(true);  
$httpsAbsoluteHomeUrl = Url::home('https');
```

Se nenhum parâmetro for passado, a URL gerada é relativa. Você pode passar `true` para obter uma URL absoluta para a o esquema corrente ou especificar um esquema explicitamente (`https`, `http`).

Para obter a URL base da requisição corrente, use o seguinte :

```
$relativeBaseUrl = Url::base();  
$absoluteBaseUrl = Url::base(true);  
$httpsAbsoluteBaseUrl = Url::base('https');
```

O único parâmetro do método funciona exatamente da mesma que em `Url::home()`.

16.2.2 Criando URLs

Afim de criar ma URL para uma rota utilize o método `Url::toRoute()`. O método usa `yii\web\UrlManager` para criar a URL:

```
$url = Url::toRoute(['product/view', 'id' => 42]);
```

Você pode especificar uma URL como string, ou seja, `site/index`. Você pode também usar um array se você precisa especificar parâmetros adicionais para a URL a ser criada. O formato do array deve ser:

```
// generates: /index.php?r=site/index&param1=value1&param2=value2  
['site/index', 'param1' => 'value1', 'param2' => 'value2']
```

Se você quiser criar uma URL com uma âncora (anchor), você pode usar no array o parâmetro `#`. Por exemplo,

```
// generates: /index.php?r=site/index&param1=value1#name  
['site/index', 'param1' => 'value1', '#' => 'name']
```

Uma rota pode ser absoluta ou relativa. Uma rota absoluta tem uma barra inicial (e.g. `/site/index`) enquanto uma rota relativa não (e.g. `site/index` or `index`). Uma rota relativa pode ser convertida em absoluta seguindo as seguintes regras:

- Se a rota é uma string vazia, será usada a corrente rota `route`;
- Se a rota não contém nenhuma barra (e.g. `index`), considera-se ser o ID da ação corrente do controlador; e serão precedidas por `yii\web\Controller::$uniqueId`;
- Se a rota não tem uma barra inicial (e.g. `site/index`), isto é considerado um URL relativa para para o modulo corrente e será precedido por `uniqueId`.

A partir da versão 2.0.2, você pode especificar uma rota como um `alias`. Se esse é o caso, o alias será primeiro convertido para rota atual que irá então ser transformado em uma rota absoluta de acordo às regras acima .

Abaixo estão alguns exemplos de como usar este método:

```
// /index.php?r=site/index
echo Url::toRoute('site/index');

// /index.php?r=site/index&src=ref1#name
echo Url::toRoute(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post/edit&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/index
echo Url::toRoute('site/index', true);

// https://www.example.com/index.php?r=site/index
echo Url::toRoute('site/index', 'https');
```

Há um outro método `Url::to()` que é muito semelhante a `toRoute()`. A única diferença é que este método requer uma rota a ser especificado como apenas como array. Se for dado uma string, ela será tratada como um URL.

O primeiro argumento pode ser:

- um array: `toRoute()` irá ser chamado para gerar a URL. Por exemplo: `['site/index', ['post/index', 'page' => 2]]`. Por favor consulte `toRoute()` para mais detalhes de como especificar uma rota.
- uma string com início `@`: ele é tratado como um alias, e as strings correspondentes ao alias serão devolvidos.
- uma string vazia: A URL da requisição corrente será retornado;
- uma string normal: será devolvida como ele foi passada (como ela é).

Quando `$scheme` é especificado (como uma string ou `true`), uma URL absoluta com informações do host (obtida de `yii\web\UrlManager::$hostInfo`) será retornada. Se `$url` já é uma URL absoluta, seu scheme irá ser substituído pelo o especificado.

Abaixo estão alguns exemplos de uso:

```
// /index.php?r=site/index
echo Url::to(['site/index']);
```

```
// /index.php?r=site/index&src=ref1#name
echo Url::to(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post/edit&id=100    assume the alias "postEdit" is defined
as "post/edit"
echo Url::to(['@postEdit', 'id' => 100]);

// the currently requested URL
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');
```

A partir da versão 2.0.3, você pode usar `yii\helpers\Url::current()` para criar uma URL base para rota solicitada e parâmetros GET. Você pode modificar ou remover alguns dos parâmetros GET ou adicionar novos por passando o parâmetro `$ params` para o método. Por exemplo,

```
// assume $_GET = ['id' => 123, 'src' => 'google'], current route is
"post/view"

// /index.php?r=post/view&id=123&src=google
echo Url::current();

// /index.php?r=post/view&id=123
echo Url::current(['src' => null]);
// /index.php?r=post/view&id=100&src=google
echo Url::current(['id' => 100]);
```

16.2.3 Relembrar URLs

Há casos em que você precisa se lembrar URL e depois usá-lo durante o processamento de uma das requisições sequenciais. Pode ser conseguida da seguinte forma:

```
// Remember current URL
Url::remember();

// Remember URL specified. See Url::to() for argument format.
Url::remember(['product/view', 'id' => 42]);

// Remember URL specified with a name given
Url::remember(['product/view', 'id' => 42], 'product');
```

Na próxima requisição, podemos obter URL lembrada da seguinte forma:

```
$url = Url::previous();  
$productUrl = Url::previous('product');
```

16.2.4 Verificar URLs relativas

Para saber se a URL é relativa, ou seja, ele não tem informações do host, você pode usar o seguinte código:

```
$isRelative = Url::isRelative('test/it');
```