Przewodnik po Yii 2.0

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

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

Polski translation provided by:
Paweł Brzozowski,
Tomek Romik,
Daniel Filipek

This tutorial is released under the Terms of Yii Documentation.

Copyright 2014 Yii Software LLC. All Rights Reserved.

Spis treści

1 Wstęp **1**
- 1.1 Czym jest Yii . . . 1
- 1.2 Aktualizacja z wersji 1.1 . . . 2

2 Pierwsze kroki **15**
- 2.1 Instalacja Yii . . . 17
- 2.2 Uruchamianie aplikacji . . . 26
- 2.3 Witaj świecie . . . 30
- 2.4 Praca z formularzami . . . 33
- 2.5 Praca z bazami danych . . . 39
- 2.6 Generowanie kodu za pomocą Gii . . . 45
- 2.7 Dalsze kroki . . . 51

3 Struktura aplikacji **53**
- 3.1 Struktura aplikacji . . . 53
- 3.2 Skrypty wejściowe . . . 54
- 3.3 Komponenty aplikacji . . . 58

4 Obsługa żądań **69**
- 4.1 Przegląd . . . 69
- 4.2 Bootstrapping . . . 70

5 Kluczowe koncepcje **79**
- 5.1 Komponenty . . . 79
- 5.2 Behaviory . . . 84
- 5.3 Aliasy . . . 93
- 5.4 Autoładowanie klas . . . 95

6 Praca z bazami danych **101**
- 6.1 Active Record . . . 104
- 6.2 Migracje bazy danych . . . 137

7 Odbieranie danych od użytkowników **161**
7.1 Tworzenie formularzy . . . 161
7.2 Walidacja danych wejściowych . . . 166
7.3 Wysyłanie plików . . . 184
7.4 Odczytywanie tablicowych danych wejściowych . . . 188
7.5 Pobieranie danych dla wielu modeli . . . 190
7.6 Rozszerzanie ActiveForm po stronie klienta . . . 192

8 Wyświetlanie danych **197**
8.1 Paginacja . . . 199
8.2 Praca ze skryptami . . . 204

9 Bezpieczeństwo **207**
9.1 Bezpieczeństwo . . . 207

10 Pamięć podręczna **213**
10.1 Pamięć podręczna . . . 213
10.2 Pamięć podręczna fragmentów . . . 215
10.3 Pamięć podręczna stron . . . 218
10.4 Pamięć podręczna HTTP . . . 219

11 Webserwisy z wykorzystaniem REST **223**
11.1 Routing . . . 227
11.2 Limit użycia . . . 231
11.3 Wersjonowanie . . . 232
11.4 Obsługa błędów . . . 234

12 Narzędzia wspomagające tworzenie aplikacji **237**

13 Testowanie **239**
13.1 Testowanie . . . 239
13.2 Przygotowanie środowiska testowego . . . 241
13.3 Testy jednostkowe . . . 242
13.4 Testy funkcjonalne . . . 243
13.5 Testy akceptacyjne . . . 243

14 Tematy specjalne **245**
14.1 Tworzenie własnej struktury aplikacji . . . 245
14.2 Wysyłanie poczty . . . 251
14.3 Współdzielone środowisko hostujące . . . 257
14.4 Silniki szablonów . . . 259
14.5 Używanie Yii jako mikroframeworka . . . 262

15 Widżety **267**

16 Klasy pomocnicze **269**
16.1 Klasy pomocnicze 269

17 Uwagi do polskiego tłumaczenia przewodnika **275**

Rozdział 1

Wstęp

1.1 Czym jest Yii

Yii jest wysoko wydajnym, opartym na komponentach frameworkiem PHP do szybkiego programowania nowoczesnych stron internetowych. Nazwa Yii (wymawiana `[ji:]`) oznacza w języku chińskim "prosto i ewolucyjnie". Może to być również rozumiane jako akronim dla **Yes It Is**!

1.1.1 Dla jakich zastosowań Yii jest najlepszy?

Yii jest frameworkiem ogólnego przeznaczenia, co oznacza, że może być wykorzystany do stworzenia każdego rodzaju aplikacji internetowych korzystających z PHP. Z uwagi na architekturę opartą na komponentach i zaawansowane wsparcie dla mechanizmów pamięci podręcznej jest on odpowiedni do tworzenia rozbudowanych aplikacji, takich jak: portale, fora, systemy zarządzania treścią (CMS), projekty komercyjne (e-sklepy), usługi sieciowe i inne.

1.1.2 Jak wygląda porównanie Yii z innymi frameworkami?

Jeśli korzystałeś już z innych frameworków, na pewno docenisz, jak Yii wypada na ich tle:

- Jak większość frameworków, Yii wykorzystuje architekturę MVC (Model-Widok-Kontroler) i wspiera organizację kodu zgodną z tym wzorcem.
- Yii opiera się na filozofii, która mówi, że kod powinien być napisany w prosty, ale jednocześnie elegancki sposób. Yii nigdy nie będzie upierać się przy przeprojektowaniu kodu jedynie w celu dokładnego trzymania się zasad wzorca projektowego.
- Yii jest w pełni rozwiniętym frameworkiem dostarczającym sprawdzonych i gotowych do użycia funkcjonalności: konstruktorów zapytań oraz ActiveRecord dla baz danych relacyjnych i NoSQL, wsparcia dla tworzenia RESTful API oraz wielopoziomowych mechanizmów pamięci podręcznej i wielu, wielu innych.

- Yii jest ekstremalnie rozszerzalny. Możesz dostosować lub wymienić praktycznie każdy fragment podstawowego kodu. Dodatkowo Yii wykorzystuje architekturę rozszerzeń, dzięki czemu możesz w prosty sposób stworzyć i opublikować swoje własne moduły i widżety.
- Podstawowym celem, do którego Yii zawsze dąży, jest wysoka wydajność.

Yii nie jest efektem pracy pojedynczego programisty - projekt wspiera zarówno grupa doświadczonych deweloperów[1], jak i ogromna społeczność programistyczna, nieustannie przyczyniając się do jego rozwoju. Deweloperzy trzymają rękę na pulsie najnowszych trendów Internetu, za pomocą prostych i eleganckich interfejsów wzbogacając Yii w najlepsze sprawdzone rozwiązania i funkcjonalności, dostępne w innych frameworkach i projektach.

1.1.3 Wersje Yii

Yii aktualnie dostępny jest w dwóch głównych wersjach: 1.1 i 2.0. Wersja 1.1 jest kodem starszej generacji, obecnie w fazie utrzymaniowej. Wersja 2.0 jest całkowicie przepisaną wersją Yii z uwzględnieniem najnowszych protokołów i technologii, takich jak Composer, PSR, przestrzenie nazw, traity i wiele innych. 2.0 reprezentuje aktualną generację frameworka i na niej skupi się głównie praca programistów w ciągu najbliższych lat.
Ten przewodnik opisuje wersję 2.0.

1.1.4 Wymagania i zależności

Yii 2.0 wymaga PHP w wersji 5.4.0 lub nowszej i pracuje najwydajniej na najnowszej wersji PHP. Aby otrzymać więcej informacji na temat wymagań i indywidualnych funkcjonalności, uruchom specjalny skrypt testujący system dołączony w każdym wydaniu Yii.

Używanie Yii wymaga podstawowej wiedzy o programowaniu obiektowym w PHP (OOP), ponieważ Yii jest frameworkiem czysto obiektowym. Yii 2.0 wykorzystuje ostatnie udoskonalenia w PHP, jak przestrzenie nazw[2] i traity[3]. Zrozumienie tych konstrukcji pomoże Ci szybciej i łatwiej rozpocząć pracę z Yii 2.0.

1.2 Aktualizacja z wersji 1.1

Pomiędzy wersjami 1.1 i 2.0 Yii jest ogrom różnic, ponieważ framework został całkowicie przepisany w 2.0. Z tego też powodu aktualizacja z wersji 1.1 nie jest tak trywialnym procesem, jak w przypadku aktualizacji pomiędzy

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

pomniejszymi wersjami. W tym przewodniku zapoznasz się z największymi różnicami dwóch głównych wersji.

Jeśli nie korzystałeś wcześniej z Yii 1.1, możesz pominąć tę sekcję i przejść bezpośrednio do "Pierwszych kroków".

Zwróć uwagę na to, że Yii 2.0 wprowadza znacznie więcej nowych funkcjonalności, niż wymienionych jest w tym podsumowaniu. Wskazane jest zapoznanie się z treścią całego przewodnika, aby poznać je wszystkie. Jest bardzo prawdopodobne, że niektóre z mechanizmów, które poprzednio musiałeś stworzyć samemu, teraz są częścią podstawowego kodu.

1.2.1 Instalacja

Yii 2.0 w pełni korzysta z udogodnień Composera[4], będącego de facto menadżerem projektów PHP. Z jego pomocą odbywa się zarówno instalacja podstawowego frameworka, jak i wszystkich rozszerzeń. Aby zapoznać się ze szczegółową instrukcją instalacji Yii 2.0, przejdź do sekcji Instalacja Yii. Jeśli chcesz stworzyć nowe rozszerzenie lub zmodyfikować istniejące w wersji 1.1, aby było kompatybilne z 2.0, przejdź do sekcji Tworzenie rozszerzeń.

1.2.2 Wymagania PHP

Yii 2.0 wymaga PHP w wersji 5.4 lub nowszej, która została znacząco ulepszona w stosunku do wersji 5.2 (wymaganej przez Yii 1.1). Z tego też powodu już na poziomie samego języka pojawiło się sporo różnic, na które należy zwrócić uwagę. Poniżej znajdziesz krótkie podsumowanie głównych różnic dotyczących PHP:

- Przestrzenie nazw[5].
- Funkcje anonimowe[6].
- Skrócona składnia zapisu tablic `[...elementy...]` używana zamiast `array(...elementy...)`.
- Krótkie tagi echo `<?=` używane w plikach widoków. Można ich używać bezpiecznie, począwszy od PHP 5.4.
- Klasy i interfejsy SPL[7].
- Opóźnione statyczne wiązania[8].
- Data i czas[9].
- Traity[10].
- Rozszerzenie intl[11]. Yii 2.0 korzysta z rozszerzenia PHP `intl` do wsparcia obsługi internacjonalizacji.

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

1.2.3 Przestrzeń nazw

Najbardziej oczywista zmiana w Yii 2.0 dotyczy używania przestrzeni nazw. Praktycznie każda z podstawowych klas je wykorzystuje, np. `yii\web\Request`. Prefiks "C" nie jest już używany w nazwach, a sam schemat nazewnictwa odpowiada teraz strukturze folderów - dla przykładu `yii\web\Request` wskazuje, że plik klasy to `web/Request.php` znajdujący się w folderze frameworka Yii.

Dzięki mechanizmowi ładowania klas Yii możesz użyć dowolnej podstawowej klasy frameworka bez konieczności bezpośredniego dołączania jej kodu.

1.2.4 Komponent i obiekt

Yii 2.0 rozdzielił klasę `CComponent` z 1.1 na dwie: `BaseObject` i `Component`. Lekka klasa `BaseObject` pozwala na zdefiniowanie właściwości obiektu poprzez gettery i settery. Klasa `Component` dziedziczy po `BaseObject` i dodatkowo wspiera obsługę zdarzeń oraz zachowań.

Jeśli Twoja klasa nie wymaga ww. wsparcia, rozważ użycie `BaseObject` jako klasy podstawowej. Tak jest zazwyczaj w przypadku klas reprezentujących najbardziej podstawową strukturę danych.

1.2.5 Konfiguracja obiektu

Klasa `BaseObject` wprowadza ujednoliconą formę konfigurowania obiektów. Każda klasa dziedzicząca po `BaseObject` powinna zadeklarować swój konstruktor (jeśli tego wymaga) w następujący sposób, dzięki czemu zostanie poprawnie skonfigurowana:

```
class MyClass extends \yii\base\BaseObject
{
    public function __construct($param1, $param2, $config = [])
    {
        // ... inicjalizacja przed skonfigurowaniem

        parent::__construct($config);
    }

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

        // ... inicjalizacja po skonfigurowaniu
    }
}
```

W powyższym przykładzie ostatnim parametrem konstruktora musi być tablica konfiguracyjna, zawierająca pary nazwa-wartość służące do zainicjowania właściwości na końcu konstruktora. Możesz nadpisać metodę `init()`, aby wykonać dodatkowy proces inicjalizacyjny po zaaplikowaniu konfiguracji.

Dzięki tej konwencji możesz tworzyć i konfigurować nowe obiekty, używając tablicy konfiguracyjnej:

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

Więcej szczegółów na temat konfiguracji znajdziesz w sekcji Konfiguracje.

1.2.6 Zdarzenia (Events)

W Yii 1 zdarzenia były tworzone poprzez definiowanie `on`-metody (np., `onBeforeSave`). W Yii 2 możesz użyć dowolnej nazwy. Uruchomienie zdarzenia następuje poprzez wywołanie metody `trigger()`:

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

Aby dołączyć uchwyt do zdarzenia, użyj metody `on()`:

```
$component->on($eventName, $handler);
// a aby od␣łączyć uchwyt użyj:
// $component->off($eventName, $handler);
```

Zdarzenia zostały wzbogacone w wiele udoskonaleń. Więcej szczegółów na ten temat znajdziesz w sekcji Zdarzenia (Events).

1.2.7 Aliasy ścieżek

Yii 2.0 rozszerza funkcjonalność aliasów ścieżek zarówno na ścieżki plików oraz folderów, jak i adresy URL. Yii 2.0 wymaga teraz też, aby nazwa aliasu zaczynała się znakiem `@` w celu odróżnienia jej od zwyczajnych ścieżek plików/folderów lub URLi. Dla przykładu: alias `@yii` odnosi się do folderu instalacji Yii. Aliasy ścieżek są wykorzystywane w większości miejsc w podstawowym kodzie Yii, choćby `cachePath` - można tu przekazać zarówno zwykłą ścieżkę, jak i alias.

Alias ścieżki jest mocno powiązany z przestrzenią nazw klasy. Zalecane jest, aby zdefiniować alias dla każdej podstawowej przestrzeni nazw, dzięki czemu mechanizm automatycznego ładowania klas Yii nie będzie wymagał dodatkowej konfiguracji. Dla przykładu: dzięki temu, że `@yii` odwołuje się do folderu instalacji Yii, klasa taka jak `yii\web\Request` może być automatycznie załadowana. Jeśli używasz zewnętrznych bibliotek, jak np. Zend Framework, możesz zdefiniować alias `@Zend` odnoszący się do folderu instalacji tego frameworka. Od tej pory Yii będzie również w stanie automatycznie załadować każdą klasę z tej biblioteki.

Więcej o aliasach ścieżek dostępne jest w sekcji Aliasy.

1.2.8 Widoki

Najbardziej znaczącą zmianą dotyczącą widoków w Yii 2 jest użycie specjalnej zmiennej `$this`. W widoku nie odnosi się ona już do aktualnego kontrolera lub widżetu, lecz do obiektu *widoku*, nowej koncepcji przedstawionej w 2.0. Obiekt *widoku* jest klasą typu `View`, która reprezentuje część wzorca MVC. Jeśli potrzebujesz odwołać się do kontrolera lub widżetu w widoku, możesz użyć `$this->context`.

Aby zrenderować częściowy widok wewnątrz innego widoku, możesz użyć `$this->render()` zamiast dotychczasowego `$this->renderPartial()`. Wywołanie `render` musi teraz też być bezpośrednio wyechowane, ponieważ metoda `render()` zwraca rezultat renderowania zamiast od razu go wyświetlać.

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

Oprócz wykorzystania PHP jako podstawowego języka szablonów, Yii 2.0 oficjalnie wspiera dwa popularne silniki szablonów: Smarty i Twig (The Prado nie jest już wspierany). Aby użyć któregokolwiek z tych silników, musisz skonfigurować komponent aplikacji `view` poprzez ustawienie właściwości `$renderers`. Po więcej szczegółów przejdź do sekcji Silniki szablonów.

1.2.9 Modele

Yii 2.0 korzysta z `Model` jako bazowego modelu, podobnie jak `CModel` w 1.1. Klasa `CFormModel` została całkowicie usunięta, w Yii 2 należy rozszerzyć `Model`, aby stworzyć klasę modelu formularza.

Yii 2.0 wprowadza nową metodę `scenarios()`, służącą do deklarowania scenariuszy, jak i do oznaczania, w którym scenariuszu atrybut będzie wymagał walidacji, może być uznany za bezpieczny lub nie itp. Dla przykładu:

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

Widzimy tutaj dwa zadeklarowane scenariusze: `backend` i `frontend`. W scenariuszu `backend` obydwa atrybuty, `email` i `role`, są traktowane jako bezpieczne i mogą być przypisane zbiorczo. W przypadku scenariusza `frontend`, `email` może być przypisany zbiorczo, ale `role` już nie. Zarówno `email` jak i `role` powinny przejść proces walidacji.

Metoda `rules()` wciąż służy do zadeklarowania zasad walidacji. Zauważ, że z powodu wprowadzenia `scenarios()`, nie ma już walidatora `unsafe`.

Jeśli metoda `rules()` deklaruje użycie wszystkich możliwych scenariuszy i jeśli nie masz potrzeby deklarowania atrybutów `unsafe` (niebezpiecznych), w większości przypadków nie potrzebujesz nadpisywać metody `scenarios()`.

Aby dowiedzieć się więcej o modelach, przejdź do sekcji Modele.

1.2.10 Kontrolery

Yii 2.0 używa `Controller` jako bazowej klasy kontrolera, podobnie do `CController` w Yii 1.1. `Action` jest bazową klasą dla akcji.

Najbardziej oczywistą implikacją tych zmian jest to, że akcja kontrolera powinna zwracać zawartość, którą chcesz wyświetlić, zamiast wyświetlać ją bezpośrednio:

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

Przejdź do sekcji Kontrolery, aby poznać więcej szczegółów na ten temat.

1.2.11 Widżety

Yii 2.0 korzysta z `Widget` jako bazowej klasy widżetów, podobnie jak `CWidget` w Yii 1.1.

Dla lepszego wsparcia frameworka w aplikacjach IDE Yii 2.0 wprowadził nową składnię używania widżetów. Używane są teraz metody `begin()`, `end()` i `widget()` w następujący sposób:

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

// Zwróć uwagę na konieczność użycia "echo", aby wyświetlić rezultat
echo Menu::widget(['items' => $items]);

// Przekazujemy tablicę, aby zainicjalizować w␣łaściwości obiektu
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge']],
]);
... pola formularza w tym miejscu ...
ActiveForm::end();
```

Więcej szczegółów na ten temat znajdziesz w sekcji Widżety.

1.2.12 Skórki i motywy (Theming)

Skórki działają zupełnie inaczej w 2.0. Oparte są teraz na mechanizmie mapowania ścieżki, który przekształca źródłowy plik widoku w plik widoku

skórki. Dla przykładu, jeśli mapa ścieżki dla skórki to ['/web/views' => '/web/themes/basic'], to skórkowa wersja pliku widoku /web/views/site/index.php to /web/themes/basic/site/index.php. Dzięki temu skórki mogą być użyte dla dowolnego pliku widoku, nawet w przypadku widoku wyrenderowanego poza kontekstem kontrolera lub widżetu.

Nie ma również już komponentu CThemeManager. Zamiast tego theme jest konfigurowalną właściwością komponentu aplikacji view.

Sekcja Skórki i motywy (Theming) zawiera więcej szczegółów na ten temat.

1.2.13 Aplikacje konsolowe

Aplikacje konsolowe używają teraz kontrolerów tak jak aplikacje webowe. Kontrolery konsolowe powinny rozszerzać klasę yii\console\Controller, podobnie jak CConsoleCommand w 1.1.

Aby uruchomić polecenie konsoli, użyj yii <route>, gdzie <route> oznacza ścieżkę kontrolera (np. sitemap/index). Dodatkowe anonimowe argumenty są przekazywane jako parametry do odpowiedniej metody akcji kontrolera, natomiast nazwane argumenty są przetwarzane według deklaracji zawartych w options().

Yii 2.0 wspiera automatyczne generowanie informacji pomocy poprzez bloki komentarzy.

Aby dowiedzieć się więcej na ten temat, przejdź do sekcji Komendy konsolowe.

1.2.14 I18N

Yii 2.0 usunął wbudowany formater dat i liczb na rzecz modułu PECL intl PHP[12].

Tłumaczenia wiadomości odbywają się teraz poprzez komponent aplikacji i18n, w którym można ustalić zestaw źródeł treści, dzięki czemu możliwy jest ich wybór dla różnych kategorii wiadomości.

W sekcji Internacjonalizacja znajdziesz więcej szczegółów na ten temat.

1.2.15 Filtry akcji

Filtry akcji są implementowane od teraz za pomocą zachowań (behavior). Aby zdefiniować nowy filtr, należy rozszerzyć klasę ActionFilter. Użycie filtru odbywa się poprzez dołączenie go do kontrolera jako zachowanie. Dla przykładu: aby użyć filtra yii\filters\AccessControl, dodaj poniższy kod w kontrolerze:

```
public function behaviors()
{
```

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

```
    return [
        'access' => [
            'class' => 'yii\filters\AccessControl',
            'rules' => [
                ['allow' => true, 'actions' => ['admin'], 'roles' => ['@']],
            ],
        ],
    ];
}
```

Więcej informacji na ten temat znajdziesz w sekcji Filtry.

1.2.16 Zasoby (Assets)

Yii 2.0 wprowadza nowy mechanizm tzw. *pakietów zasobów*, który zastąpił koncepcję pakietów skryptowych z Yii 1.1.

Pakiet zasobów jest kolekcją plików zasobów (np. plików JavaScript, CSS, obrazków, itd.) zgromadzoną w folderze. Każdy pakiet jest reprezentowany przez klasę rozszerzającą `AssetBundle`. Zarejestrowanie pakietu poprzez metodę `register()` pozwala na udostępnienie go publicznie. W przeciwieństwie do rozwiązania z Yii 1, strona rejestrująca pakiet będzie automatycznie zawierać referencje do plików JavaScript i CSS wymienionych na jego liście.

Sekcja Zasoby (Assets) zawiera szczegółowe informacje na ten temat.

1.2.17 Klasy pomocnicze

Yii 2.0 zawiera wiele powszechnie używanych statycznych klas pomocniczych (helperów), takich jak:

- `Html`
- `ArrayHelper`
- `StringHelper`
- `FileHelper`
- `Json`

W sekcji Klasy pomocnicze znajdziesz więcej informacji na ten temat.

1.2.18 Formularze

Yii 2.0 wprowadza koncepcję *pola* do budowy formularzy, korzystając z klasy `ActiveForm`. Pole jest kontenerem składającym się z etykiety, pola wprowadzenia danych formularza, informacji o błędzie i/lub tekstu podpowiedzi, reprezentowanym przez obiekt klasy `ActiveField`. Używając pól, możesz stworzyć formularz w sposób o wiele prostszy i bardziej przejrzysty niż do tej pory:

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
```

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

Aby dowiedzieć się więcej na ten temat, przejdź do sekcji Tworzenie formularzy.

1.2.19 Konstruktor kwerend

W 1.1 budowanie kwerend było rozrzucone pomiędzy kilka klas, tj. `CDbCommand`, `CDbCriteria` i `CDbCommandBuilder`. Yii 2.0 reprezentuje kwerendę bazodanową w postaci obiektu `Query`, który może być zamieniony w komendę SQL za pomocą `QueryBuilder`. Przykładowo:

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

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

Co najlepsze, taki sposób tworzenia kwerend może być również wykorzystany przy pracy z Active Record.

Po więcej szczegółów udaj się do sekcji Konstruktor kwerend.

1.2.20 Active Record

Yii 2.0 wprowadza sporo zmian w mechanizmie Active Record. Dwie najbardziej znaczące to konstruowanie kwerend i obsługa relacji.

Klasa `CDbCriteria` z 1.1 została zastąpiona przez `ActiveQuery` w Yii 2. Klasa ta rozszerza `Query`, dzięki czemu dziedziczy wszystkie metody konstruowania kwerend. Aby rozpocząć budowanie kwerendy, wywołaj metodę `find()`:

```
// Pobranie wszystkich *aktywnych* klientów i posortowanie po ich ID:
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

Deklaracja relacji polega na prostym zdefiniowaniu metody gettera, który zwróci obiekt `ActiveQuery`. Nazwa właściwości określonej przez tego gettera reprezentuje nazwę stworzonej relacji. Dla przykładu: w poniższym kodzie deklarujemy relację `orders` (w 1.1 konieczne było zadeklarowanie relacji wewnątrz wydzielonej specjalnie metody `relations()`):

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

Od tej pory można posługiwać się `$customer->orders`, aby uzyskać dostęp do tabeli zamówień klientów poprzez relację. Dodatkowo można również posłużyć się następującym kodem, aby wywołać relacyjną kwerendę dla zadanych warunków:

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

Przy "gorliwym" pobieraniu relacji (ęager", w przyciwieństwie do leniwego pobierania "lazy") Yii 2.0 działa inaczej niż w wersji 1.1. W 1.1 tworzono kwerendę JOIN, aby pobrać zarówno główne, jak i relacyjne rekordy. W Yii 2.0 wywoływane są dwie komendy SQL bez użycia JOIN - pierwsza pobiera główne rekordy, a druga relacyjne, filtrując je przy użyciu kluczy głównych rekordów.

Aby zmniejszyć zużycie CPU i pamięci, zamiast zwracać obiekty `ActiveRecord`, do kwerendy pobierającej dużą ilość rekordów możesz podpiąć metodę `asArray()`, dzięki czemu zostaną one pobrane jako tablice. Przykładowo:

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

Inną istotną zmianą jest to, że nie można już definiować domyślnych wartości atrybutów poprzez publiczne właściwości. Jeśli potrzebujesz takich definicji, powinieneś przypisać je wewnątrz metody `init` w klasie rekordu.

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

Nadpisywanie konstruktora klasy ActiveRecord w 1.1 wiązało się z pewnymi problemami, co nie występuje już w wersji 2.0. Zwróć jednak uwagę na to, że przy dodawaniu parametrów do konstruktora możesz potrzebować nadpisać metodę `instantiate()`.

W nowym rekordzie aktywnym znajdziesz wiele innych zmian i udogodnień. Aby zapoznać się z nimi, przejdź do sekcji Rekord aktywny.

1.2.21 Zachowania Active Record

W 2.0 zrezygnowaliśmy z bazowej klasy zachowania `CActiveRecordBehavior`. Jeśli chcesz stworzyć zachowanie dla rekordu aktywnego, musisz rozszerzyć bezpośrednio klasę `Behavior`. Jeśli klasa zachowania ma reagować na zdarzenia, powinna nadpisywać metodę `events()`, jak zaprezentowano poniżej:

```
namespace app\components;

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

class MyBehavior extends Behavior
{
    // ...

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

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

1.2.22 Klasa User i IdentityInterface

Klasa `CWebUser` z 1.1 została zastąpiona przez `User` i nie ma już klasy `CUserIdentity`. Zamiast tego należy zaimplementować interfejs `IdentityInterface`, który jest znacznie bardziej wygodny i oczywisty w użyciu. Szablon zaawansowanego projektu zawiera przykład takiego właśnie użycia.

Po więcej szczegółów zajrzyj do sekcji Uwierzytelnianie, Autoryzacja i Szablon zaawansowanego projektu[13].

1.2.23 Zarządzanie adresami URL

Zarządzanie adresami URL w Yii 2 jest bardzo podobne do tego znanego z 1.1. Głównym ulepszeniem tego mechanizmu jest teraz wsparcie dla parametrów opcjonalnych. Dla przykładu: warunek dla adresu zadeklarowany poniżej obejmie zarówno `post/popular` jak i `post/1/popular`. W 1.1 konieczne byłoby napisanie dwóch warunków, aby osiągnąć to samo.

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

Przejdź do sekcji Zarządzania adresami URL po więcej informacji.

Istotną zmianą konwencji nazw dla adresów jest to, że nazwy kontrolerów i akcji typu "camel case" są teraz konwertowane do małych liter, z każdym

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

słowem oddzielonym za pomocą myślnika, np. ID kontrolera `CamelCaseController` zostanie przekształcone w `camel-case`.

Zapoznaj się z sekcją dotyczącą ID kontrolerów i ID akcji.

1.2.24 Korzystanie z Yii 1.1 i 2.x jednocześnie

Jeśli chciałbyś skorzystać z kodu napisanego dla Yii 1.1 w aplikacji Yii 2.0, prosimy o zapoznanie się z sekcją Używanie Yii 1.1 i 2.0 razem.

Rozdział 2

Pierwsze kroki

Error: not existing file: start-prerequisites.md

2.1 Instalacja Yii

Yii możesz zainstalować na dwa sposoby, korzystając z Composera[1] lub pobierając plik archiwum.
Preferowanym sposobem jest ten pierwszy, ponieważ pozwala na instalację i aktualizację dodatkowych rozszerzeń oraz samego Yii przy użyciu zaledwie jednej komendy.

Standardowa instalacja Yii skutkuje pobraniem i wstępnym skonfigurowaniem frameworka wraz z szablonem projektu.
Szablon projektu jest aplikacją Yii zawierającą podstawowe funkcjonalności, takie jak logowanie, formularz kontaktowy itp.
Struktura jego kodu została stworzona w oparciu o zalecany sposób pisania aplikacji opartych na Yii, dlatego może służyć jako dobry punkt wyjściowy dla stworzenia Twojego bardziej zaawansowanego projektu.

W tej oraz kilku kolejnych sekcjach opiszemy jak zainstalować Yii z tak zwanym "podstawowym szablonem projektu" oraz jak zaimplementować w nim nowe funkcjonalności. Oprócz podstawowego, Yii dostarcza również drugi, zaawansowany szablon projektu[2], przystosowany dla programistów tworzących wielowarstwowe aplikacje.

> **Informacja:** Podstawowy szablon projektu jest odpowiedni dla 90zaawansowanego szablonu projektu, jest organizacja kodu. Jeśli dopiero zaczynasz swoją przygodę z Yii, zalecamy zapoznać się z podstawowym szablonem, ze względu na jego prostotę oraz funkcjonalność.

2.1.1 Instalacja z użyciem Composera

Instalacja Composera

Jeśli nie posiadasz jeszcze Composera, to możesz go zainstalować korzystając z instrukcji zamieszczonej na stronie getcomposer.org[3]. W systemach operacyjnych Linux i Mac OS X należy wywołać następujące komendy:

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

W systemie Windows należy pobrać i uruchomić Composer-Setup.exe[4].

W przypadku napotkania jakichkolwiek problemów należy zapoznać się z sekcją Rozwiązywania problemów w dokumentacji Composera[5].

[1] https://getcomposer.org/
[2] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md
[3] https://getcomposer.org/download/
[4] https://getcomposer.org/Composer-Setup.exe
[5] https://getcomposer.org/doc/articles/troubleshooting.md

Jeśli dopiero rozpoczynasz przygodę z Composerem, zalecamy przeczytanie przynajmniej sekcji Podstaw użycia[6] w dokumentacji Composera.

W tym przewodniku zakładamy, że Composer został zainstalowany globalnie[7], dzięki czemu jest dostępny z użyciem komendy `composer`. Jeśli jednak zamiast tego używasz pliku `composer.phar` w lokalnym folderze, pamiętaj, żeby odpowiednio zmodyfikować podane tu przykładowe komendy.

Jeśli jesteś już posiadaczem Composera, upewnij się, że jest on zaktualizowany do najnowszej wersji (komenda `composer self-update`).

> **Uwaga:** Podczas instalacji Yii, Composer będzie potrzebował pobrać sporo informacji z API serwisu Github. Ilość zapytań zależy od liczby powiązanych wtyczek, rozszerzeń i modułów, których wymaga Twoja aplikacja, i może być większa niż **limit zapytań API GitHuba**. Jeśli faktycznie tak będzie, Composer może poprosić o Twoje dane logowania w serwisie Github, aby uzyskać token dostępowy API Githuba. Przy szybkim łączu napotkanie limitu może nastąpić szybciej niż Composer jest w stanie obsłużyć zapytania, zatem zalecane jest skonfigurowanie tokenu dostępowego przed instalacją Yii.
> Instrukcja opisująca jak tego dokonać znajduje się w dokumentacji Composera dotyczącej tokenów API Githuba[8].

Installing Yii

Teraz możesz przejść już do instalacji samego Yii, wywołując poniższe komendy w katalogu dostępnym z poziomu sieci web:

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

Komenda ta zainstaluje najnowszą stabilną wersję szablonu aplikacji Yii w katalogu `basic`. Możesz oczywiście wybrać inną nazwę.

> **Informacja:** Jeśli komenda `composer create-project` zwróci błąd, sprawdź, czy przypadkiem nie jest on już opisany w dokumentacji Composera w sekcji Rozwiązywania problemów[9]. Kiedy uporasz się już z błędem, możesz wznowić przerwaną instalację uruchamiając komendę `composer update` w folderze `basic`.

> **Wskazówka:** Jeśli chcesz zainstalować najnowszą wersję deweloperską Yii, użyj poniższej komendy, która dodaje opcję stabilności[10]:

[6] https://getcomposer.org/doc/01-basic-usage.md
[7] https://getcomposer.org/doc/00-intro.md#globally
[8] https://getcomposer.org/doc/articles/troubleshooting.md#api-rate-limit-and-oauth-tokens
[9] https://getcomposer.org/doc/articles/troubleshooting.md
[10] https://getcomposer.org/doc/04-schema.md#minimum-stability

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

Pamiętaj, że wersja deweloperska Yii nie powinna być używana w wersjach produkcyjnych Twojej aplikacji, ponieważ mogą wystąpić w niej niespodziewane błędy.

2.1.2 Instalacja z pliku archiwum

Instalacja Yii z pliku archiwum składa się z trzech kroków:

1. Pobranie pliku archiwum z yiiframework.com[11].
2. Rozpakowanie pliku archiwum do katalogu dostępnego z poziomu sieci web.
3. Zmodyfikowanie pliku `config/web.php` przez dodanie sekretnego klucza do elementu konfiguracji `cookieValidationKey` (jest to wykonywane automatycznie, jeśli instalujesz Yii używając Composera):

```
// !!! wprowadź sekretny klucz tutaj - jest to wymagane do walidacji
ciasteczek
'cookieValidationKey' => 'enter your secret key here',
```

2.1.3 Inne opcje instalacji

Powyższe instrukcje pokazują, jak zainstalować Yii oraz utworzyć podstawową, gotową do uruchomienia aplikację web. To podejście jest dobrym punktem startowym dla większości projektów, zarówno małych jak i dużych. Jest to szczególnie korzystne, gdy zaczynasz naukę Yii.

Dostępne są również inne opcje instalacji:

- Jeśli chcesz zainstalować wyłącznie framework i samemu zbudować aplikację, zapoznaj się z rozdziałem Tworzenie aplikacji od podstaw.
- Jeśli chcesz utworzyć bardziej zaawansowaną aplikację, przystosowaną do programowania dla wielu środowisk, powinieneś rozważyć instalację zaawansowanego szablonu aplikacji[12].

2.1.4 Instalowanie zasobów

Yii używa menadżerów pakietów Bower[13] i/lub NPM[14] do instalacji bibliotek zasobów (CSS i JavaScript). Proces pobierania tych bibliotek korzysta z Composera, pozwalając na rozwiązywanie zależności pakietów PHP i

[11] https://www.yiiframework.com/download/
[12] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md
[13] https://bower.io/
[14] https://www.npmjs.com/

CSS/JavaScript w tym samym czasie, za pomocą serwisu asset-packagist.org[15] lub wtyczki composer asset plugin[16].
Po więcej informacji sięgnij do sekcji dokumentacji Zasobów.

Możesz, rzecz jasna, również zarządzać swoimi zasobami za pomocą natywnego klienta Bower/NPM, korzystać z CDN, albo też całkowicie zrezygnować z instalacji zasobów. Aby zablokować automatyczne pobieranie zasobów podczas używania Composera, dodaj poniższe linie w pliku '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"
},
```

> **Uwaga:** w przypadku zablokowania instalacji zasobów przez Composera, odpowiedzialność za ich instalację i rozwiązywanie zależności spada na Ciebie. Przygotuj się na potencjalne niezgodności w plikach zasobów pochodzących z różnych rozszerzeń.

2.1.5 Weryfikacja instalacji

Po zakończeniu instalacji, skonfiguruj swój serwer (zobacz następną sekcję) lub użyj wbudowanego serwera PHP[17], uruchamiając poniższą komendę w konsoli z poziomu folderu `web` w projekcie:

```
php yii serve
```

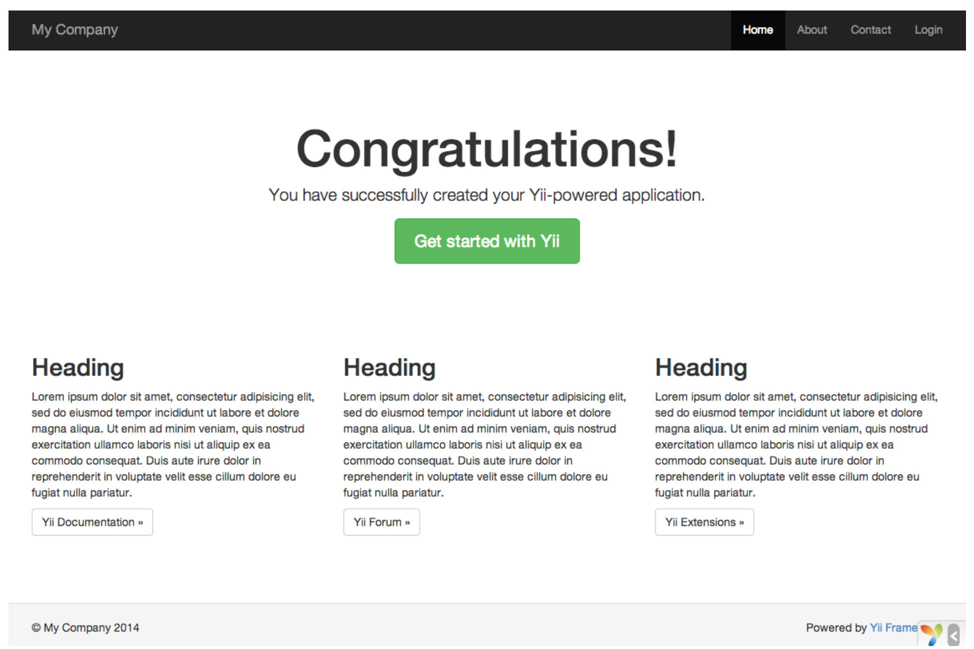
> **Uwaga:** Domyślnym portem, na którym serwer HTTP nasłuchuje, jest 8080. Jeśli jednak ten port jest już w użyciu lub też chcesz obsłużyć wiele aplikacji w ten sposób, możesz podać inny numer portu, dodając argument –port:

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

Możesz teraz użyć swojej przeglądarki, aby uzyskać dostęp do zainstalowanej aplikacji Yii przechodząc pod adres:

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

[15] https://asset-packagist.org
[16] https://github.com/fxpio/composer-asset-plugin
[17] https://www.php.net/manual/en/features.commandline.webserver.php



Powinieneś zobaczyć stronę z napisem "Congratulations!" ("Gratulacje!"). Jeśli nie, sprawdź, czy zainstalowane elementy środowiska spełniają wymagania Yii. Możesz sprawdzić minimalne wymagania na dwa sposoby:

- Skopiuj plik `/requirements.php` do `/web/requirements.php`, a następnie przejdź do przeglądarki i uruchom go przechodząc pod adres `http://localhost/requirements.php`
- Lub też uruchom następujące komendy:

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

Powinieneś skonfigurować swoją instalację PHP tak, aby spełniała minimalne wymogi Yii. Najważniejszym z nich jest posiadanie PHP w wersji 5.4 lub wyższej. Powinieneś również zainstalować rozszerzenie PDO[18] oraz odpowiedni sterownik bazy danych (np. `pdo_mysql` dla bazy danych MySQL), jeśli Twoja aplikacja potrzebuje bazy danych.

2.1.6 Konfigurowanie serwerów WWW

> **Informacja:** Możesz pominąć tą sekcję, jeśli tylko testujesz Yii, bez zamiaru zamieszczania aplikacji na serwerze produkcyjnym.

Aplikacja zainstalowana według powyższych instrukcji powinna działać bezproblemowo zarówno na serwerze HTTP Apache[19] jak i serwerze HTTP Nginx[20], na systemie operacyjnym Windows, Mac OS X oraz Linux, posiadającym

[18] https://www.php.net/manual/en/pdo.installation.php
[19] https://httpd.apache.org
[20] https://nginx.org

zainstalowane PHP 5.4 lub nowsze. Yii 2.0 jest również kompatybilne z facebookowym HHVM[21], są jednak przypadki, gdzie Yii zachowuje się inaczej w HHVM niż w natywnym PHP, dlatego powinieneś zachować szczególną ostrożność używając HHVM.

Na serwerze produkcyjnym możesz skonfigurować swój host tak, aby aplikacja była dostępna pod adresem `https://www.example.com/index.php` zamiast `https://www.example.com/basic/web/index.php`. Taka konfiguracja wymaga wskazania głównego katalogu serwera jako katalogu `basic/web`. Jeśli chcesz ukryć `index.php` w adresie URL, skorzystaj z informacji opisanych w dziale routing i tworzenie adresów URL.
W tej sekcji dowiesz się, jak skonfigurować Twój serwer Apache lub Nginx, aby osiągnąć te cele.

> **Informacja:** Ustawiając `basic/web` jako główny katalog serwera, unikasz niechcianego dostępu użytkowników końcowych do prywatnego kodu oraz wrażliwych plików aplikacji, które są przechowywane w katalogu `basic`. Zablokowanie dostępu do tych folderów jest jednym z wymogów bezpieczeństwa aplikacji.

> **Informacja:** W przypadku, gdy Twoja aplikacja działa na wspólnym środowisku hostingowym, gdzie nie masz dostępu do modyfikowania konfiguracji serwera, nadal możesz zmienić strukturę aplikacji dla lepszej ochrony. Po więcej informacji zajrzyj do działu Współdzielone środowisko hostingowe.

Zalecane ustawienia Apache

Użyj następującej konfiguracji serwera Apache w pliku `httpd.conf` lub w konfiguracji wirtualnego hosta. Pamiętaj, że musisz zamienić ścieżkę `path/to/basic/web` na aktualną ścieżkę do `basic/web` Twojej aplikacji.

```
# Ustaw g␣łówny katalog na "basic/web"
DocumentRoot "path/to/basic/web"

<Directory "path/to/basic/web">
    # użyj mod_rewrite do wsparcia "␣ladnych URLi"
    RewriteEngine on

    # jeśli $showScriptName jest ustawione na false w UrlManager, nie
    pozwalaj na dostęp do URLi za pomocą nazwy skryptu
    RewriteRule ^index.php/ - [L,R=404]

    # Jeśli katalog lub plik istnieje, użyj go bezpośrednio
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
```

[21] `https://hhvm.com`

```
    # W innym przypadku przekieruj żądanie na index.php
    RewriteRule . index.php

    # ...inne ustawienia...
</Directory>
```

Zalecane ustawienia Nginx

Aby użyć Nginx[22] powinieneś zainstalować PHP jako FPM SAPI[23]. Możesz użyć przedstawionej poniżej konfiguracji Nginx, zastępując jedynie ścieżkę `path/to/basic/web` aktualną ścieżką do `basic/web` Twojej aplikacji oraz `mysite.test` aktualną nazwą hosta.

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

    listen 80; ## nas␣luchuj ipv4
    #listen [::]:80 default_server ipv6only=on; ## nas␣luchuj ipv6

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

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

    location / {
        # Przekieruj wszystko co nie jest prawdziwym plikiem na index.php
        try_files $uri $uri/ /index.php$is_args$args;
    }

    # odkomentuj poniższe aby uniknąć przetwarzania żądań do nieistniejących
    plików przez Yii
    #location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.html;

    # zablokuj dostęp do plików php w folderze /assets
    location ~ ^/assets/.*\.php$ {
        deny all;
    }

    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;
```

[22] https://wiki.nginx.org/

[23] https://www.php.net/install.fpm

```
    }

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

W przypadku użycia tej konfiguracji, powinieneś ustawić również `cgi.fix_pathinfo=0` w pliku `php.ini`, aby zapobiec wielu zbędnym wywołaniom `stat()`.

Należy również pamiętać, że podczas pracy na serwerze HTTPS musisz dodać `fastcgi_param HTTPS on;`, aby Yii prawidłowo wykrywało, że połączenie jest bezpieczne.

Zalecane ustawienia NGINX Unit

Możesz uruchomić aplikacje oparte na Yii korzystając z NGINX Unit[24] z modułem języka PHP. Poniżej znajdziesz przykładową konfigurację.

```
{
    "listeners": {
        "*:80": {
            "pass": "routes/yii"
        }
    },

    "routes": {
        "yii": [
            {
                "match": {
                    "uri": [
                        "!/assets/*",
                        "*.php",
                        "*.php/*"
                    ]
                },

                "action": {
                    "pass": "applications/yii/direct"
                }
            },
            {
                "action": {
                    "share": "/path/to/app/web/",
                    "fallback": {
                        "pass": "applications/yii/index"
                    }
                }
            }
        ]
    },
```

[24] https://unit.nginx.org/

```
    "applications": {
        "yii": {
            "type": "php",
            "user": "www-data",
            "targets": {
                "direct": {
                    "root": "/path/to/app/web/"
                },

                "index": {
                    "root": "/path/to/app/web/",
                    "script": "index.php"
                }
            }
        }
    }
}
```

Możesz również skonfigurować[25] swoje środowisko PHP lub przygotować spersonalizowany plik `php.ini` w tej samej konfiguracji.

Konfiguracja IIS

Zalecane jest hostowanie aplikacji na wirtualnym hoście (strona Web), gdzie podstawowa ścieżka dokumentów wskazuje na folder `path/to/app/web` i strona Web jest skonfigurowana do uruchamiania PHP. W folderze `web` musisz umieścić plik `web.config` (`path/to/app/web/web.config`). Zawartość tego pliku powinna wyglądać jak poniżej:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<system.webServer>
<directoryBrowse enabled="false" />
  <rewrite>
    <rules>
      <rule name="Hide Yii Index" stopProcessing="true">
        <match url="." ignoreCase="false" />
        <conditions>
        <add input="{REQUEST_FILENAME}" matchType="IsFile"
              ignoreCase="false" negate="true" />
        <add input="{REQUEST_FILENAME}" matchType="IsDirectory"
              ignoreCase="false" negate="true" />
        </conditions>
        <action type="Rewrite" url="index.php" appendQueryString="true" />
      </rule>
    </rules>
  </rewrite>
</system.webServer>
</configuration>
```

[25] https://unit.nginx.org/configuration/#php

Sprawdź również poniższe oficjalne poradniki firmy Microsoft, opisujące jak poprawnie skonfigurować PHP dla IIS:

1. Jak uruchomić swoją pierwszą stronę Web na IIS[26]
2. Konfiguracja strony Web PHP dla IIS[27]

2.2 Uruchamianie aplikacji

Po zainstalowaniu Yii posiadasz działającą aplikację Yii dostępną pod adresem `https://hostname/basic/web/index.php` lub `https://hostname/index.php`, zależnie od Twojej konfiguracji. Ta sekcja wprowadzi Cię do wbudowanych funkcjonalności aplikacji, pokaże jak zorganizowany jest jej kod oraz jak aplikacja obsługuje żądania.

> **Informacja:** Dla uproszczenia zakładamy, że ustawiłeś główny katalog serwera na `basic/web`, według poradnika "Instalacja Yii", oraz skonfigurowałeś adres URL tak, aby Twoja aplikacja była dostępna pod adresem `https://hostname/index.php`. Dla Twoich potrzeb dostosuj odpowiednio adres URL w naszych opisach.

Należy pamiętać, że w przeciwieństwie do samego frameworka, po zainstalowaniu szablonu projektu należy on w całości do Ciebie. Możesz dowolnie dodawać, modyfikować lub usuwać kod, zależnie od Twoich potrzeb.

2.2.1 Funkcjonalność

Zainstalowana podstawowa aplikacja posiada cztery strony:
- stronę główną, która jest wyświetlana przy wywołaniu adresu `https://hostname/index.php`,
- strona informacyjna `About`,
- strona kontaktowa `Contact`, gdzie wyświetlany jest formularz kontaktowy, pozwalający użytkownikowi skontaktować się z Tobą przez email,
- strona logowania `Login`, gdzie wyświetlany jest formularz logowania, który może być użyty do uwierzytelniania użytkowników. Zaloguj się danymi "admin/admin", przez co pozycja `Login` z menu zamieni się na `Logout`.

Wszystkie te strony posiadają wspólne nagłówek i stopkę. Nagłówek zawiera główne menu pozwalające na nawigację po innych stronach.

Powinieneś również widzieć pasek narzędzi na dole okna przeglądarki. Jest to użyteczne narzędzie do debugowania[28] dostarczone przez Yii, zapisujące

[26] https://docs.microsoft.com/en-us/iis/manage/creating-websites/scenario-build-a-static-website-on-iis
[27] https://docs.microsoft.com/en-us/iis/application-frameworks/scenario-build-a-php-website-on-iis/configure-a-php-website-on-iis
[28] https://github.com/yiisoft/yii2-debug/blob/master/docs/guide/README.md

i wyświetlające wiele informacji, takich jak wiadomości logów, statusy odpowiedzi, zapytania do baz danych i wiele innych.

Dodatkowo do aplikacji Web dostarczono skrypt konsolowy nazwany `yii`, który jest ulokowany w głównym katalogu aplikacji. Skrypt może być użyty do uruchomienia w tle zadań dla aplikacji, które są opisane w sekcji Komendy konsolowe.

2.2.2 Struktura aplikacji

Najważniejsze katalogi oraz pliki w Twojej aplikacji to (zakładając, że główny katalog aplikacji to `basic`):

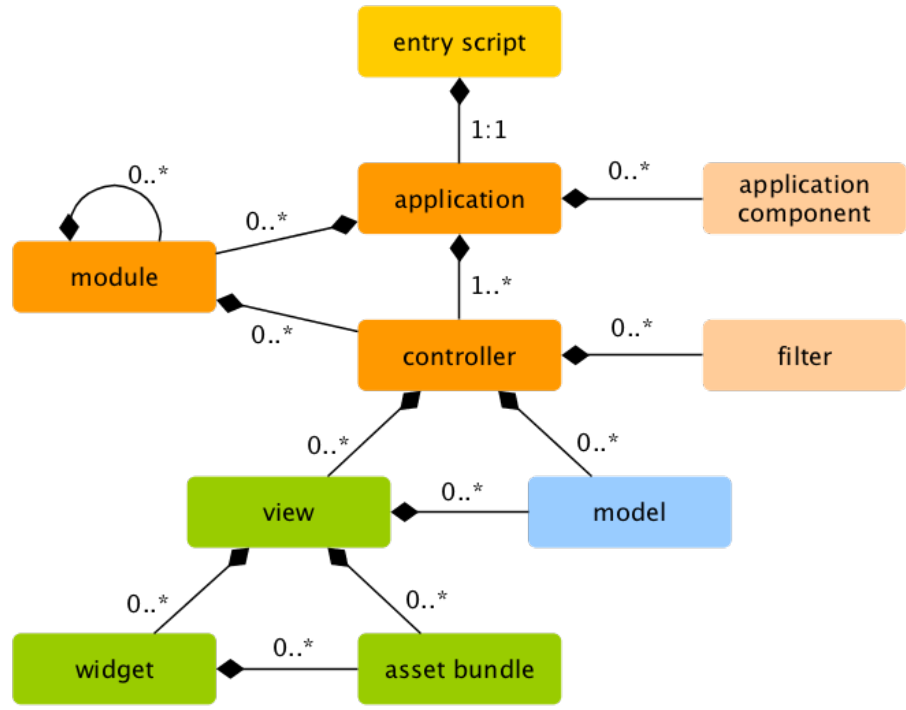
```
basic/                  bazowa ścieżka aplikacji
    composer.json       plik używany przez Composer, opisuje informacje
    paczek
    config/             zawiera wszystkie konfiguracje, w tym aplikacji
        console.php     konfiguracja konsoli aplikacji
        web.php         konfiguracja aplikacji Web
    commands/           zawiera klasy komend konsoli
    controllers/        zawiera klasy kontrolerów
    models/             zawiera klasy modeli
    runtime/            zawiera pliki wygenerowane przez Yii podczas pracy,
    takie jak logi i pliki cache
    vendor/             zawiera zainstalowane paczki Composer'a, w tym
    framework Yii
    views/              zawiera pliki widoków
    web/                ścieżka aplikacji Web, zawiera dostępne publicznie
    pliki
        assets/         zawiera opublikowane przez Yii pliki zasobów
        (javascript oraz css)
        index.php       skrypt wejściowy dla aplikacji
    yii                 skrypt wykonujący komendy konsolowe Yii
```

Ogólnie pliki aplikacji mogą zostać podzielone na dwa typy: pliki w katalogu `basic/web` oraz pliki w innych katalogach. Dostęp do pierwszego typu można uzyskać przez HTTP (np. przez przeglądarkę), podczas gdy reszta nie może, i nie powinna być, dostępna publicznie.

Yii implementuje wzór architektoniczny model-widok-kontroler (MVC)[29], który jest odzwierciedleniem przedstawionej wyżej organizacji katalogów. Katalog `models` zawiera wszystkie klasy modeli, katalog `views` zawiera wszystkie skrypty widoków oraz katalog `controllers` zawiera wszystkie klasy kontrolerów.

Poniższy schemat pokazuje statyczną strukturę aplikacji.

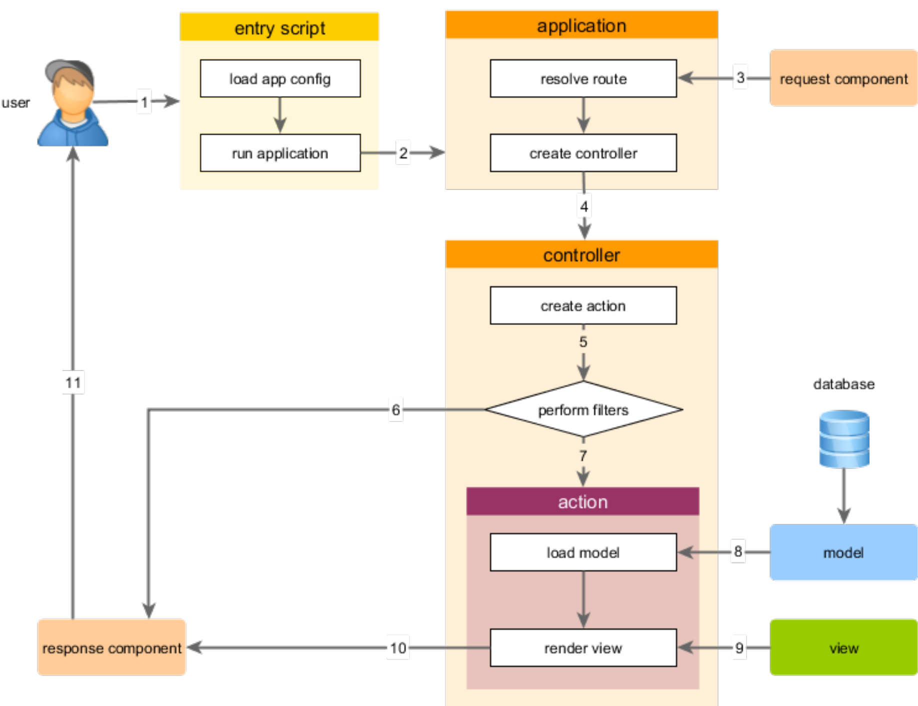
[29] https://wikipedia.org/wiki/Model-view-controller



Każda aplikacja zawiera skrypt wejściowy `web/index.php`, który jest jedynym publicznie dostępnym skryptem PHP w aplikacji. Skrypt wejściowy pobiera przychodzące żądanie i tworzy instancję aplikacji do przetworzenia tego żądania. Aplikacja obsługuje żądanie z pomocą komponentów, po czym wysyła żądanie do elementów MVC. Widżety są używane w widokach, aby pomóc zbudować złożone i dynamiczne elementy interfejsu użytkownika.

2.2.3 Cykl życia żądania

Poniższy schemat pokazuje jak aplikacja przetwarza żądania.



1. Użytkownik tworzy zapytanie do skryptu wejściowego `web/index.php`.
2. Skrypy wejściowy ładuje konfigurację aplikacji oraz tworzy instancję aplikacji w celu przetworzenia żądania.
3. Aplikacja obsługuje żądanie route'a z pomocą komponentu żądania aplikacji.
4. Aplikacja tworzy instancję kontrolera do obsługi żądania.
5. Kontroler tworzy instancję akcji i wykonuje filtrowanie dla akcji.
6. Jeśli warunek dowolnego z filtrów nie jest spełniony, akcja jest zatrzymana.
7. W przeciwnym wypadku wywoływana jest akcja.
8. Akcja wczytuje model danych, prawdopodobnie z bazy danych.
9. Akcja renderuje widok, dostarczając mu model danych.
10. Wynik zwracany jest do komponentu odpowiedzi aplikacji.
11. Komponent odpowiedzi wysyła wynik do przeglądarki użytkownika.

2.3 Witaj świecie

Ta sekcja opisuje jak utworzyć nową stronę "Witaj" w Twojej aplikacji. Aby to osiągnąć, musisz utworzyć akcję i widok:

- Aplikacja wyśle żądanie strony web do akcji
- Następnie akcja włączy widok, który pokazuje użytkownikowi słowo "Witaj".

Podczas tego poradnika nauczysz się trzech rzeczy:

1. Jak utworzyć akcję, która będzie odpowiadać na żądania,
2. Jak utworzyć widok, aby wyeksponować treść odpowiedzi,
3. Jak aplikacja wysyła żądania do akcji.

2.3.1 Tworzenie akcji

Do zadania "Witaj" utworzysz akcję `say`, która odczytuje parametr `message` z żądania oraz wyświetla tą wiadomość użytkownikowi. Jeśli żądanie nie dostarczy parametru `message`, akcja wyświetli domyślnie wiadomość "Witaj".

> **Informacja:** Akcje są obiektami, do których użytkownik może bezpośrednio odnieść się, aby je wywołać. Akcje są pogrupowane w kontrolery. Wynikiem użycia akcji jest odpowiedź, którą otrzyma końcowy użytkownik.

Akcje muszą być deklarowane w kontrolerach. Dla uproszczenia, możesz zdeklarować akcję `say` w już istniejącym kontrolerze `SiteController`. Kontroler jest zdefiniowany w klasie `controllers/SiteController.php`. Oto początek nowej akcji:

```php
<?php

namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    // ...obecny kod...

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

W powyższym kodzie, akcja `say` jest zdefiniowana jako metoda o nazwie `actionSay` w klasie `SiteController`. Yii używa prefixu `action` do rozróżnienia

metod akcji od zwykłych metod w klasie kontrolera. Nazwa po prefixie `action` kieruje do ID akcji.

Podczas nazywania Twoich akcji powinieneś zrozumieć jak Yii traktuje ID akcji. Odwołanie do ID akcji zawsze występuje z małych liter. Jeśli ID akcji potrzebuje wielu słów, będą one łączone myślnikami (np. `create-comment`). Nazwy metod akcji są przypisywane do ID akcji przez usunięcie myślników z ID, przekształcenie piewszej litery w słowie na dużą literę oraz dodanie prefixu `action`. Dla przykładu akcja o ID `create-comment` odpowiada metodzie akcji o nazwie `actionCreateComment`.

Metoda akcji w naszym przykładzie przyjmuje parametr `$message`, którego wartość domyślna to `"Hello"` (w ten sam sposób ustawiasz domyślną wartość dla każdego argumentu funkcji lub metody w PHP). Kiedy aplikacja otrzymuje żądanie i określa, że akcja `say` jest odpowiedzialna za jego obsługę, aplikacja uzupełni parametr znaleziony w żądaniu. Innymi słowy, jeśli żądanie zawiera parametr `message` z wartością `"Goodbye"` to do zmiennej `$message` w akcji będzie przypisana ta wartość.

W metodzie akcji wywołana jest funkcja `render()`, która renderuje nam widok pliku o nazwie `say`. Parametr `message` jest również przekazywany do widoku, co sprawia, że może być w nim użyty. Metoda akcji zwraca wynik renderowania. Wynik ten będzie odebrany przez aplikację oraz wyświetlony końcowemu użytownikowi w przeglądarce (jako część kompletnej strony HTML).

2.3.2 Tworzenie widoku

Widoki są skryptami, które tworzysz w celu wyświetlenia treści odpowiedzi. Do zadania "Hello" utworzysz widok `say`, który wypisuje parametr `message` otrzymany z metody akcji.

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

Widok `say` powinien być zapisany w pliku `views/site/say.php`. Kiedy wywołana jest metoda `render()` w akcji, będzie ona szukała pliku PHP nazwanego wg schematu `views/ControllerID/ViewName.php`.

Zauważ, że w powyższym kodzie parametr `message` jest przetworzony za pomocą metody `encode()` przed wyświetleniem go. Jest to konieczne w przypadku parametrów pochodzących od użytkownika, wrażliwych na ataki XSS[30] przez podanie złośliwego kodu JavaScript.

Naturalnie możesz umieścić więcej zawartości w widoku `say`. Zawartość może zawierać tagi HTML, czysty tekst, a nawet kod PHP. Tak naprawdę, widok `say` jest tylko skryptem PHP, który jest wywoływany przez metodę `render()`. Zawartość wyświetlana przez skrypt widoku będzie zwrócona do

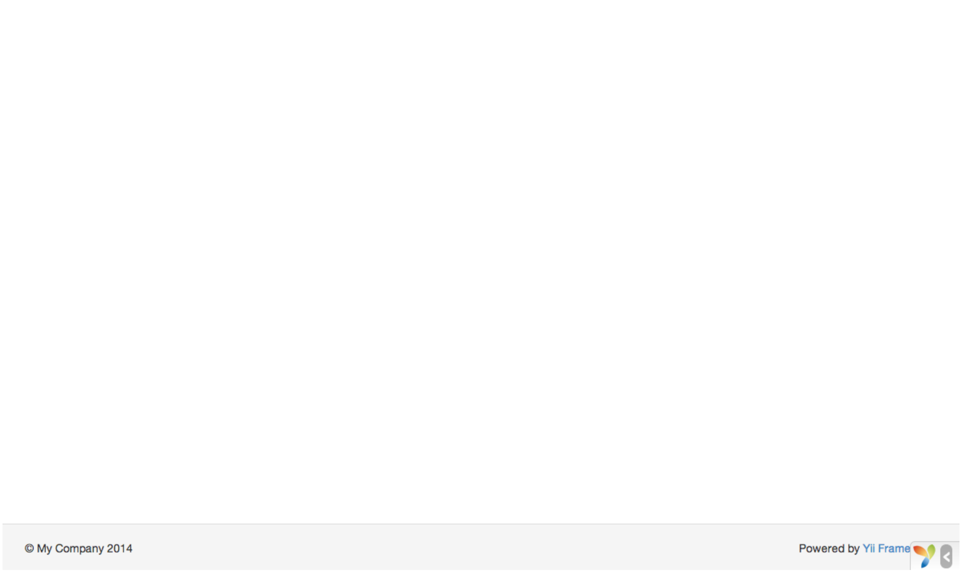
[30] https://en.wikipedia.org/wiki/Cross-site_scripting

aplikacji jako wynik odpowiedzi. Aplikacja z kolei przedstawi ten wynik końcowemu użytkownikowi.

2.3.3 Próba

Po utworzeniu akcji oraz widoku możesz uzyskać dostęp do nowej strony przez przejście pod podany adres URL:

```
https://hostname/index.php?r=site%2Fsay&message=Hello+World
```



Wynikiem wywołania tego adresu jest wyświetlenie napisu "Hello World". Strona dzieli ten sam nagłówek i stopkę z innymi stronami aplikacji.

Jeśli pominiesz parametr `message` w adresie URL, zobaczysz na stronie tylko "Hello". `message` jest przekazywane jako parametr do metody `actionSay` i, jeśli zostanie pominięty, zostanie użyta domyślna wartość `"Hello"`.

> **Informacja:** Nowa strona dzieli ten sam nagłówek i stopkę z innymi stronami, ponieważ metoda `render()` automatycznie osadza wynik widoku `say` w tak zwanym układzie strony, który, w tym przypadku, znajduje się w `views/layouts/main.php`.

Parametr `r` w powyższym adresie URL wymaga głębszego objaśnienia. Oznacza on route'a, identyfikator akcji unikatowy w obrębie aplikacji. Format route'a to `ControllerID/ActionID`. Kiedy aplikacja otrzymuje żądanie, sprawdza ten parametr, a następnie używa części `ControllerID`, aby ustalić, która klasa kontrolera powinna zostać zainstancjowana dla przetworzenia tego żądania. Następnie, kontroler używa części `ActionID` do ustalenia, która akcja powinna

zostać użyta. W tym przykładzie, route `site/say` będzie odczytany jako klasa kontrolera `SiteController` oraz akcja `say`. W rezultacie zostanie wywołana metoda `SiteController::actionSay()`.

> **Informacja:** Tak jak i akcje, kontrolery również posiadają swoje ID, które jednoznacznie identyfikuje je w aplikacji. ID kontrolerów używają tych samych zasad nazewnictwa, co ID akcji. Nazwy klas kontrolerów uzyskiwane są z ID kontrolerów przez usunięcie myślników z ID, zamianę pierwszej litery na wielką w każdym słowie oraz dodanie przyrostka `Controller`. Dla przykładu ID kontrolera `post-comment` odpowiada nazwie klasy kontrolera `PostCommentController`.

2.3.4 Podsumowanie

W tej sekcji zobaczyłeś część kontrolerów oraz widoków wzorca architektonicznego MVC. Utworzyłeś akcję jako część kontrolera do obsługi specyficznego żądania. Utworzyłeś też widok, który prezentuje zawartość odpowiedzi. W tym prostym przykładzie nie został zaangażowany żaden model, ponieważ dane jakimi się posługiwaliśmy były zawarte w parametrze `message`.

Nauczyłeś się też czegoś o routingu w Yii, który działa jak most pomiędzy żądaniami użytkownika a akcjami kontrolerów.

W następnej sekcji nauczysz się jak utworzyć model oraz dodać nową stronę zawierającą formularz HTML.

2.4 Praca z formularzami

Ta sekcja opisuje jak utworzyć nową stronę z formularzem pobierającym dane od użytkownika. Strona będzie wyświetlała formularz z dwoma polami do uzupełnienia: `nazwa` oraz `email`. Po otrzymaniu tych dwóch danych od użytkownika, wyświetlimy z powrotem wprowadzone wartości w celu ich potwierdzenia.

Aby to osiągnąć, oprócz utworzenia akcji i dwóch widoków, będziesz musiał utworzyć model.

W tym poradniku nauczysz się jak:

- utworzyć model reprezentujący dane wprowadzone przez użytkownika przez formularz,
- zadeklarować zasady do sprawdzenia wprowadzonych danych,
- zbudować formularz HTML w widoku.

2.4.1 Tworzenie modelu

Dane które pobierzemy od użytkownika będą reprezentowane przez klasę modelu `EntryForm`, która pokazana jest poniżej. Jest ona zapisana w pliku `models/EntryForm.php`. Po więcej szczegółów odnośnie konwencji nazewnictwa plików zajrzyj do sekcjiautoładowania klas

```php
<?php

namespace app\models;

use Yii;
use yii\base\Model;

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

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

Klasa `EntryForm` rozszerza `Model`, podstawową klasę dostarczoną przez Yii, głównie używaną do reprezentowania danych z formularzy.

> **Informacja:** `Model` jest używane jako rodzic dla klasy modeli *NIE* powiązanych z tabelą bazy danych. `ActiveRecord` jest rodzicem dla klas modeli powiązanych z tabelami bazy danych.

Klasa `EntryForm` zawiera dwa elementy publiczne, `name` oraz `email`, które są używane do przechowania danych wprowadzonych przez użytkownika. Zawiera również metodę nazwaną `rules()`, która zwraca zestaw zasad do walidacji wprowadzonych danych. Zadeklarowane zasady oznaczają:

- wartości w polach `name` oraz ‘email są wymagane
- wartość pola `email` musi być prawidłowym adresem email

Jeśli posiadasz uzupełniony obiekt `EntryForm` danymi wprowadzonymi przez użytkownika, możesz wywołać jego funkcję `validate()`, aby uruchomić procedurę sprawdzania poprawności danych. W przypadku wystąpienia błędów w walidacji, wartość `hasErrors` zostanie ustawiona na `true`. Możesz zobaczyć jakie błędy wystąpiły za pomocą metody `getErrors()`.

```php
<?php
$model = new EntryForm();
$model->name = 'Qiang';
$model->email = 'bad';
if ($model->validate()) {
    // Dobrze!
} else {
    // Źle!
    // Use $model->getErrors()
}
```

2.4.2 Tworzenie akcji

Następnie musisz utworzyć akcję o nazwie `entry` w kontrolerze `site`, która użyje Twojego nowego modelu. Proces tworzenia i używania akcji był wytłumaczony w sekcji Witaj świecie.

```
<?php

namespace app\controllers;

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

class SiteController extends Controller
{
    // ...obecny kod...

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

        if ($model->load(Yii::$app->request->post()) && $model->validate())
        {
            // walidacja otrzymanych danych w modelu

            // zrób coś sensownego z modelem

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // w przypadku b␣łędów walidacji wyświetlana jest strona z
            b␣łędami
            return $this->render('entry', ['model' => $model]);
        }
    }
}
```

Akcja tworzy na początku obiekt `EntryForm`. Następnie próbuje uzupełnić model danymi ze zmiennej `$_POST`, dostarczanymi w Yii przez metodę `post()`. Jeśli model został prawidłowo uzupełniony(np. jeśli użytkownik wysłał formularz HTML), akcja wywoła metodę `validate()`, aby upewnić się, że wprowadzone dane są prawidłowe.

> **Informacja:** Wyrażenie `Yii::$app` reprezentuje instancję aplikacji, która jest globalnie dostępnym singletonem. Jest również lokatorem usług, który dostarcza komponenty takie jak `request`, `response` lub `db` do wsparcia specyficznej funkcjonalności. W powyższym kodzie użyty jest komponent `request` aby uzyskać dostęp do danych w zmiennej `$_POST`.

Jeśli wszystko jest w porządku, akcja wyrenderuje widok o nazwie `entry-confirm` w celu potwierdzenia prawidłowego przesłania danych przez użytkownika.

Jeśli nie zostały wysłane żadne dane lub dane zawierają błędy, zostanie wyrenderowany widok `entry`, w którym będzie pokazany formularz HTML wraz z wiadomościami błędów walidacji.

> **Uwaga:** W tym prostym przykładzie po prostu renderujemy stronę z potwierdzeniem prawidłowego przesłania danych. W praktyce powinieneś rozważyć użycie `refresh()` lub `redirect()`, aby uniknąć problemów z ponownym przesłaniem formularza[31].

2.4.3 Tworzenie widoku

Na koniec utwórz dwa pliki o nazwach `entry-confirm` oraz `entry`. Będą one renderowane przez akcję `entry`, tak jak to przed chwilą opisaliśmy.

Widok `entry-confirm` wyświetla po prostu dane `name` oraz `email`. Powinien być zapisany w pliku `views/site/entry-confirm.php`.

```
<?php
use yii\helpers\Html;
?>
<p>Wpisa␣leś następujące informacje:</p>

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

Widok `entry` wyświetla formularz HTML. Powinien być zapisany w pliku `views/site/entry.php`.

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

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

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

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

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

Widok używa potężnego widżetu nazwanego `ActiveForm` do budowania formularza HTML. Metody `begin()` oraz `end()` renderują odpowiednio otwierające i zamykające tagi formularza. Pomiędzy wywołaniami tych dwóch metod,

[31] https://en.wikipedia.org/wiki/Post/Redirect/Get

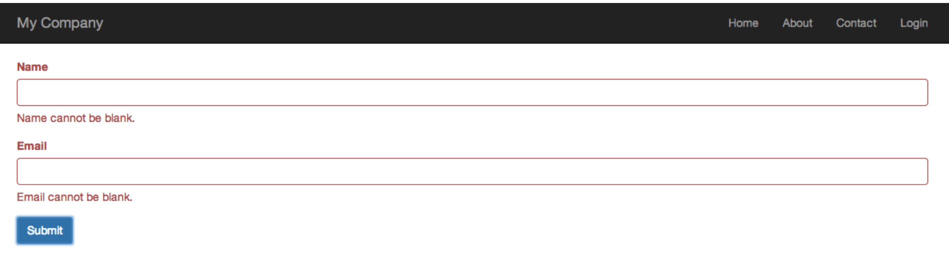
pola do wprowadzania są tworzone przez metodę `field()`. Następnie, po polach do wprowadzania danych, wywoływana jest metoda `submitButton()` do wygenerowania przycisku "Wyślij", który wysyła formularz.

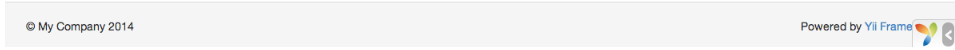
2.4.4 Próba

Aby zobaczyć jak to działa, użyj przeglądarki i przejdź pod dany adres:

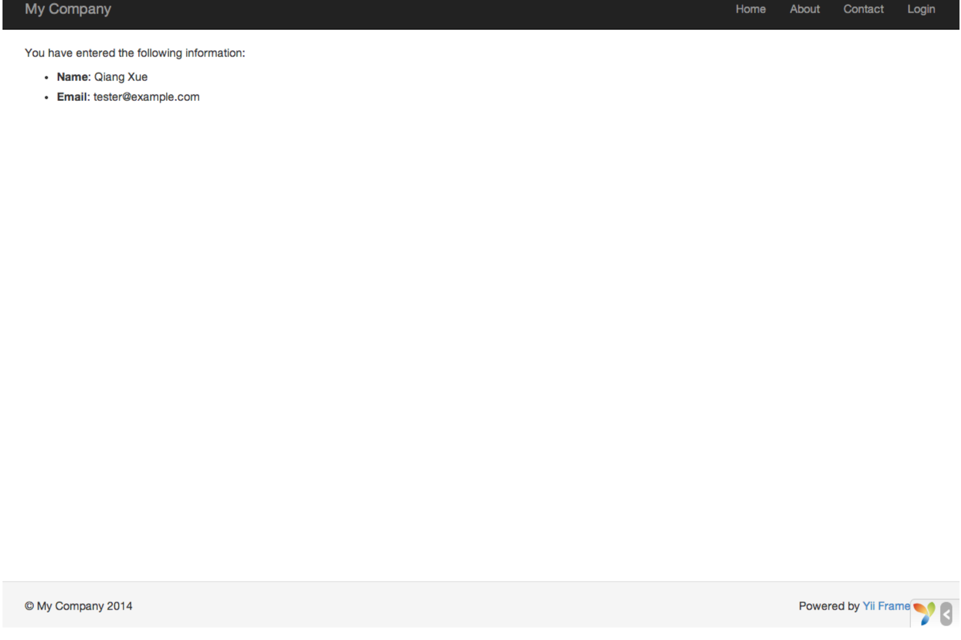
```
https://hostname/index.php?r=site%2Fentry
```

Zobaczysz stronę wyświetlającą formularz z dwoma polami. Przed każdym polem znajduje się etykieta opisująca to pole. Jeśli klikniesz przycisk "Wyślij" nie wpisując żadnych danych, lub jeśli nie wprowadzisz prawidłowego adresu email, zobaczysz wiadomość błędu wyświelona pod polem którego ona dotyczy.





Po wpisaniu prawidłowej nazwy, adresu email oraz kliknięciu przycisku "Wyślij", zobaczysz nową stronę wyświetlającą dane, które właśnie wprowadziłes.



"Wyjaśnienie magii"

Możesz się zastanawiać jak działa ten formularz HTML, ponieważ wydaje się prawie magicznym to, że wyświetla etykietę do każdego pola oraz wyświetla komunikat błędu, jeśli wprowadzisz błędne dane, bez przeładowania strony.

Wstępna walidacja jest wykonywana po stronie klienta używając JavaScriptu, kolejnie dopiero po stronie serwera przez PHP. `ActiveForm` potrafi wyodrębnić zasady walidacji, które zadeklarowałeś w `EntryForm`, przekształcić je w wykonywalny kod JavaScript oraz użyć go do walidacji danych. Jeśli zablokowałeś JavaScript w swojej przeglądarce, walidacja wciąż będzie wykonywana po stronie serwera, jak pokazano w metodzie `actionEntry()`. Gwarantuje to poprawność danych w każdych okolicznościach.

> **Ostrzeżenie:** Walidacja po stronie klienta jest opcją, która zapewnia wygodniejszą współpracę aplikacji z użytkownikiem. Walidacja po stronie serwera jest zawsze wymagana, niezależnie, czy walidacja po stronie klienta jest włączona, czy też nie.

Etykiety dla pól w formularzu generowane są przy pomocy metody `field()`, używającej nazw właściwości z modelu. Dla przykładu, etykieta `Name` zostanie wygenerowana dla właściwości `name`.

Możesz dostosowywać etykiety w widoku, używając poniższego kodu:

```
<?= $form->field($model, 'name')->label('Your Name') ?>
<?= $form->field($model, 'email')->label('Your Email') ?>
```

> **Informacja:** Yii dostarcza wiele podobnych widżetów, które pomogą Ci szybko tworzyć złożone i dynamiczne widoki. Tak jak nauczysz się później, pisanie nowego widżetu jest ekstremalnie łatwe. Będziesz mógł przekształcić Twój kod widoku na widżet do wielokrotnego użytku, aby uprościć rozwój swoich widoków w przyszłości.

2.4.5 Podsumowanie

W tej sekcji poradnika dotknęliśmy każdej części struktury MVC. Nauczyłeś się jak utworzyć klasę modelu, aby reprezentowała dane użytkownika oraz je walidowała.

Nauczyłeś się także, jak pobierać dane od użytkowników oraz jak wyświetlać pobrane dane w przeglądarce. To zadanie mogłoby zabrać Ci wiele czasu podczas pisania aplikacji, jednak Yii dostarcza wiele widżetów, które bardzo je ułatwiają.

W następnej sekcji nauczysz się pracy z bazą danych, która jest wymagana w niemalże każdej aplikacji.

2.5 Praca z bazami danych

Ta sekcja opisuje jak utworzyć nową stronę, która będzie wyświetlała dane krajów pobrane z tabeli bazy danych o nazwie `country`. Aby to osiągnąć, musisz skonfigurować swoje połączenie z bazą danych, utworzyć klasę Active Record, zdefiniować akcję oraz utworzyć widok.

W tej sekcji nauczysz się:

- konfigurowania połączenia z bazą danych,
- tworzenia klasy Active Record,
- tworzenia zapytań o dane przy użyciu klasy Active Record,
- wyświetlania danych w widoku wraz ze stronicowaniem.

Zauważ, że w celu przejścia tej sekcji należy mieć już podstawową wiedzę o bazach danych. W szczególności powinieneś wiedzieć, jak utworzyć bazę dancyh oraz jak wywołać komendę SQL używając klienta bazy danych.

2.5.1 Przygotowanie bazy danych

Aby rozpocząć, musisz utworzyć bazę danych o nazwie `yii2basic`, z której będziesz pobierał dane do swojej aplikacji. Możesz utworzyć bazę SQLite, MySQL, PostgreSQL, MSSQL lub Oracle, ponieważ Yii posiada wbudowane wsparcie dla wielu aplikacji bazodanowych. Dla uproszczenia w naszym przykładzie wykorzystamy MySQL.

Następnie, utwórz tabelę o nazwie `country` i wstaw przykładowe dane. Możesz użyc poniższej komendy:

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

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

W tym miejscu masz już utworzoną bazę danych o nazwie `yii2basic`, posiadającą tabelę `country` z trzema kolumnami. Tabela zawiera 10 wierszy danych.

2.5.2 Konfiguracja połączenia z bazą danych

Przed przystąpieniem do tej części, upewnij się, że masz zainstalowane rozszerzenie PDO[32] oraz sterownik PDO dla bazy danych której używasz (np. `pdo_mysql` dla MySQL). Jest to podstawowe wymaganie, jeśli Twoja aplikacja używa relacyjnej bazy danych.

Jeśli posiadasz zainstalowane powyższe rozszerzenia, otwórz plik `config/db.php` i zmień parametry na odpowiednie do Twojej bazy danych. Domyślnie plik zawiera poniższy fragment:

```php
<?php

return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

Plik `config/db.php` jest typowym narzędziem konfiguracyjnym opartym na plikach. Ten szczególny plik konfiguracyjny określa parametry potrzebne do utworzenia oraz zainicjalizowania instancji `Connection`, dzięki czemu będziesz mógł wywoływać komendy SQL do swojej bazy przez aplikację.

Powyższa konfiguracja może być dostępna z poziomu kodu aplikacji używając wyrażenia `Yii::$app->db`.

[32] https://www.php.net/manual/en/book.pdo.php

> **Informacja:** Plik `config/db.php` będzie załączony do głównej konfiguracji aplikacji `config/web.php`, która określa jak instancja aplikacji powinna zostać zainicjalizowana. Po więcej informacji zajrzyj do sekcji Konfiguracje.

2.5.3 Tworzenie klasy Active Record

Do pobrania i reprezentowania danych z tabeli `country` utwórz pochodną klasę Active Record o nazwie `Country`, kolejnie zapisz ją w pliku `models/Country.php`.

```
<?php

namespace app\models;

use yii\db\ActiveRecord;

class Country extends ActiveRecord
{
}
```

Klasa `Country` rozszerza klasę `ActiveRecord`. Nie musisz pisać w niej żadnego kodu! Posiadając tylko powyżej podany kod, Yii odgadnie nazwę powiązanej tabeli z nazwy klasy.

> **Informacja:** Jeśli nie można dopasować tabeli do nazwy klasy, możesz nadpisać metodę `tableName()`, aby wskazywała na konkretną powiązaną tabelę.

Używając klasy `Country` możesz w łatwy sposób manipulować danymi z tabeli `country`, tak jak pokazano w poniższych przykładach:

```
use app\models\Country;

// pobiera wszystkie wiersze tabeli `country` i porządkuje je wed␣lug
kolumny "name"
$countries = Country::find()->orderBy('name')->all();

// pobiera wiersz, którego kluczem g␣lównym jest "US"
$country = Country::findOne('US');

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

// modyfikuje nazwę kraju na "U.S.A." i zapisuje go do bazy danych
$country->name = 'U.S.A.';
$country->save();
```

> **Informacja:** Active Record jest potężnym narzędziem do dostępu i manipulacji danymi w bazie danych w sposób zorientowany

obiektowo. Więcej szczegółowych informacji znajdziesz w sekcji Active Record. Alternatywnie, do łączenia się z bazą danych możesz użyć niskopoziomowej metody dostępu do danych nazwanej Data Access Objects.

2.5.4 Tworzenie akcji

Aby przedstawić kraje użytkownikowi musisz utworzyć nową akcję. Zamiast umieszczać nową akcję w kontrolerze `site`, tak jak w poprzednich sekcjach, bardziej sensownym rozwiązaniem jest utworzenie nowego kontrolera odpowiedzialnego za wszystkie akcje dotyczące danych z tabeli `country`. Nazwij nowy kontroler `CountryController`, a następnie utwórz w nim akcję `index`, tak jak na poniższym przykładzie:

```
<?php

namespace app\controllers;

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

class CountryController extends Controller
{
    public function actionIndex()
    {
        $query = Country::find();

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

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

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

Zapisz powyższy kod w pliku `controllers/CountryController.php`.

Akcja `index` wywołuje metodę `Country::find()`, pochodzącą z klasy Active Record, która buduje zapytanie bazodanowe i wyszukuje wszystkich danych z tabeli `country`. Aby ograniczyć liczbę zwracanych krajów w każdym żądaniu, zapytanie jest stronicowane przy pomocy obiektu `Pagination`. Obiekt `Pagination` służy dwóm celom:

- Ustawia klauzule `offset` i `limit` do komend SQL reprezentowanych przez zapytanie tak, aby zwracały tylko jedną stronę na raz (maksymalnie 5 wierszy na stronę),
- Jest używany w widoku do wyświetlania stronicowania jako listy przycisków z numerami stron, co będzie wyjaśnione w kolejnej sekcji.

Na końcu akcja `index` renderuje widok o nazwie `index`, do którego przekazuje dane krajów oraz informacje o ich stronicowaniu.

2.5.5 Tworzenie widoku

W katalogu `views` utwórz nowy katalog o nazwie `country`. Będzie on używany do przechowywania wszystkich plików widoków renderowanych przez kontroler `country`. W katalogu `views/country` utwórz plik o nazwie `index.php` zawierający poniższy kod:

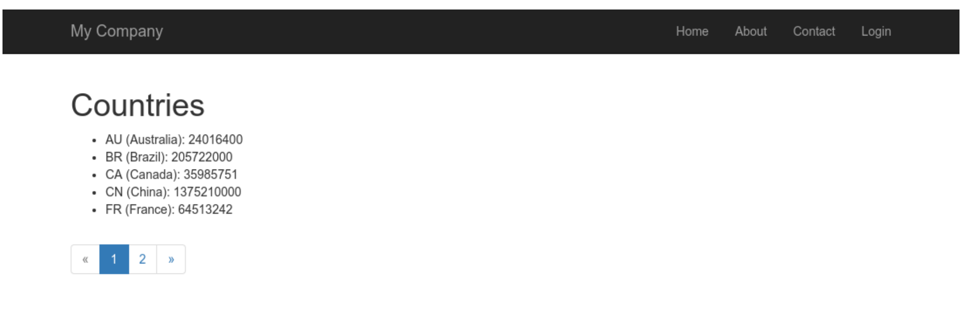
```
<?php
use yii\helpers\Html;
use yii\widgets\LinkPager;
?>
<h1>Kraje</h1>
<ul>
<?php foreach ($countries as $country): ?>
    <li>
        <?= Html::encode("{$country->name} ({$country->code})") ?>:
        <?= $country->population ?>
    </li>
<?php endforeach; ?>
</ul>

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

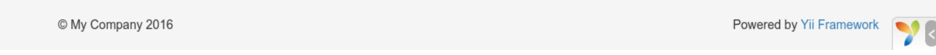
Widok posiada dwie części. Pierwsza część odpowiada za wyświetlenie danych krajów jako nieuporządkowana lista HTML, natomiast druga część renderuje widżet `LinkPager` na podstawie dostarczonych mu informacji z akcji. Widżet `LinkPager` wyświetla listę przycisków z numerami stron. Kliknięcie w którykolwiek z nich zmienia dane z listy na dane odpowiadające wybranej stronie.

2.5.6 Sprawdź jak to działa

Aby zobaczyć jak działa powyższy kod, użyj przeglądarki i przejdź pod podany adres URL:

```
https://hostname/index.php?r=country%2Findex
```





Na początku zobaczysz stronę pokazującą pięć krajów. Poniżej listy znajduje się paginacja z czterema przyciskami. Jeśli klikniesz przycisk "2", zobaczysz stronę wyświetlającą pięć innych krajów z bazy danych: druga strona wierszy. Zauważ, że adres URL w przeglądarce również się zmienił na

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

Za kulisami, `Pagination` dostarcza wszystkich niezbędnych funkcjonalności do stronicowania zbioru danych:

- Początkowo `Pagination` prezentuje pierwszą stronę, która odzwierciedla zapytanie "SELECT" tabeli `country` z klauzulą `LIMIT 5 OFFSET 0`. W rezultacie pobieranych i wyświetlanych jest pięć pierwszych krajów.
- Widżet `LinkPager` renderuje przyciski stron używając adresów URL tworzonych przez metodę `createUrl()`. Adresy zawierają parametr zapytania `page`, który reprezentuje różne numery stron.
- Jeśli klikniesz przycisk "2", zostanie uruchomione i przetworzone nowe żądanie dla route'a `country/index`. `Pagination` odczytuje parametr `query` z adresu URL, a następnie ustawia aktualny numer strony na 2. Nowe zapytanie o kraje będzie zawierało klauzulę `LIMIT 5 OFFSET 5` i zwróci pięć kolejnych krajów do wyświetlenia.

2.5.7 Podsumowanie

W tej sekcji nauczyłeś się jak pracować z bazą danych. Nauczyłeś się również jak pobierać i wyświetlać dane ze stronicowaniem przy pomocy `Pagination` oraz `LinkPager`.

W następnej sekcji nauczysz się jak używać potężnego narzędzie do generowania kodu nazwanego Gii[33], aby pomóc Ci w szybki sposób implementować niektóre powszechnie wymagane funkcjonalności, takie jak operacje CRUD dla zadań z danymi w bazie danych. Kod, który właśnie napisaliśmy, może być w całości automatycznie wygenerowany w Yii przy użyciu narzędzia Gii.

2.6 Generowanie kodu za pomocą Gii

Ta sekcja opisuje jak używać Gii[34] do automatycznego generowania kodu, który implementuje podstawowe funkcjonalności do aplikacji Web. Używanie Gii do automatycznego generowania kodu jest po prostu kwestią wprowadzenia odpowiednich informacji w formularzach zgodnie z instrukcjami widocznymi na podstronach Gii.

W tym poradniku nauczysz się:

- aktywować Gii w Twojej aplikacji,
- używać Gii do generowania klas Active Record,
- używać Gii do generowania kodu implementującego operacje CRUD dla tabel bazy danych,
- dostosować kod wygenerowany przez Gii,

2.6.1 Początki z Gii

Gii[35] jest modułem dostarczonym przez Yii. Możesz aktywować Gii konfigurując właściwość `modules` aplikacji. Zależnie od tego, jak utworzyłeś swoją aplikację, może się okazać, że poniższy kod jest już zawarty w pliku konfiguracyjnym `config/web.php`:

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

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

Powyższa konfiguracja sprawdza, czy aplikacja jest w środowisku rozwojowym - jeśli tak, dołącza moduł `gii` określając klasę modułu `yii\gii\Module`.

Jeśli sprawdzisz skrypt wejściowy `web/index.php` Twojej aplikacji, zauważysz linię kodu, która ustawia `YII_ENV` na wartość `dev`.

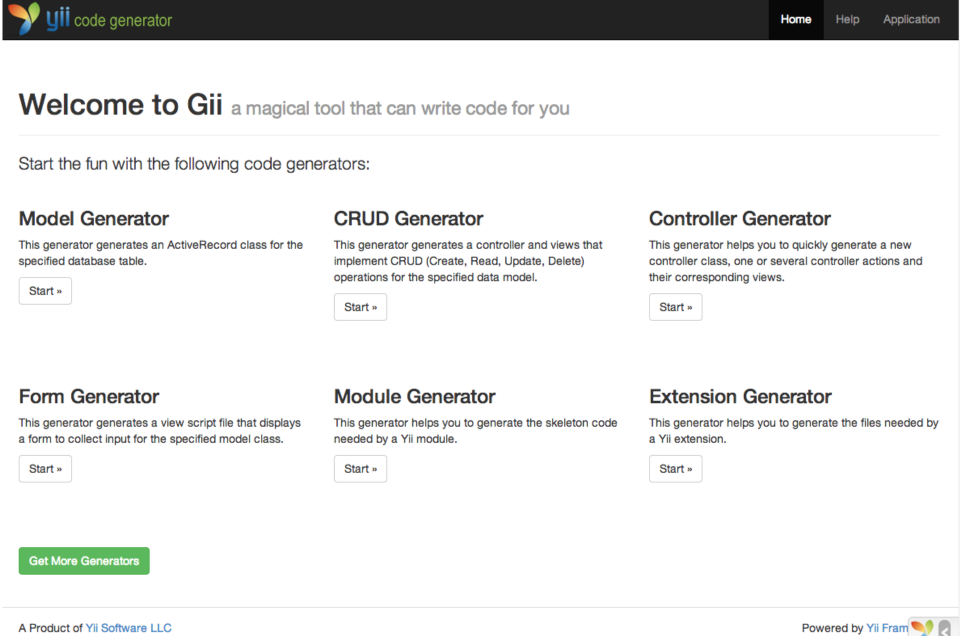
```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

[33] https://github.com/yiisoft/yii2-gii/blob/master/docs/guide/README.md
[34] https://github.com/yiisoft/yii2-gii/blob/master/docs/guide/README.md
[35] https://github.com/yiisoft/yii2-gii/blob/master/docs/guide/README.md

Dzięki temu Twoja aplikacja ustawiana jest w tryb rozwojowy, co uaktywnia moduł Gii. Możesz teraz uzyskać dostęp do Gii przez przejście pod podany adres URL:

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

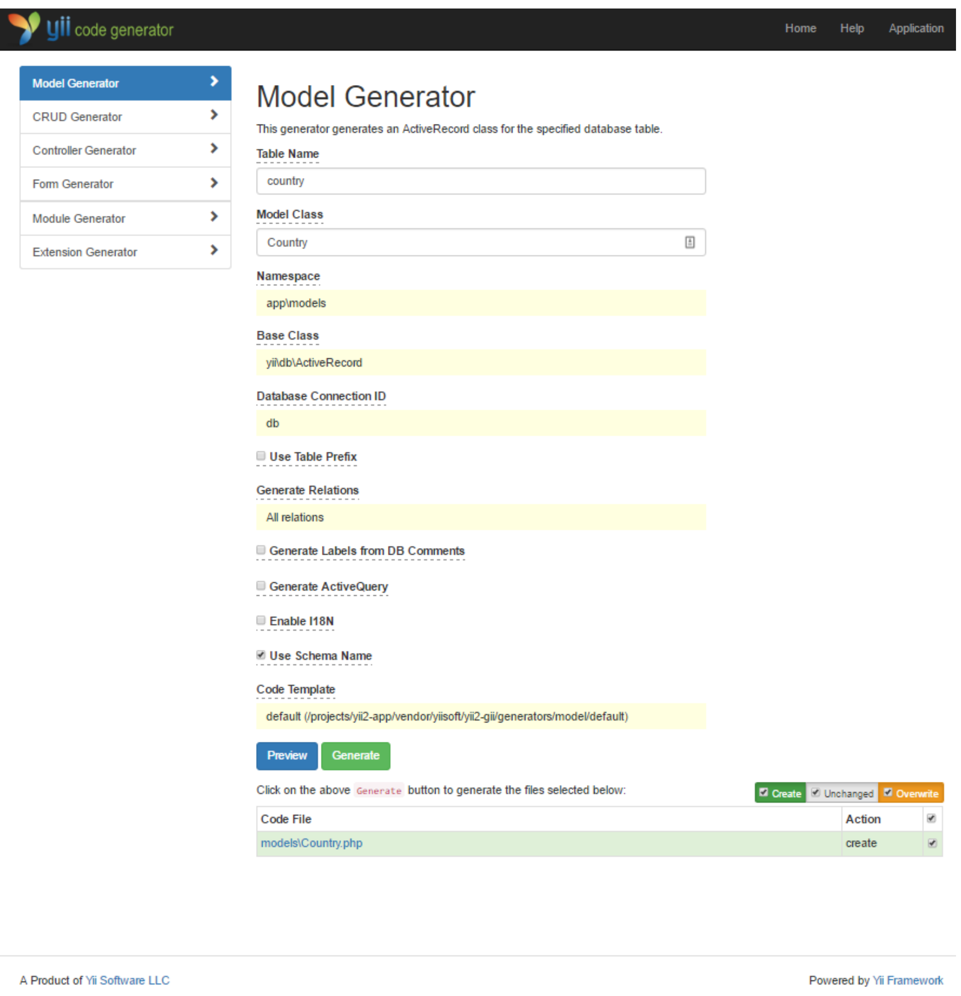
> **Uwaga:** Jeśli próbujesz dostać się do Gii z maszyny innej niż localhost, dostęp domyślnie będzie zablokowany ze względów bezpieczeństwa. Możesz dodać dozwolone adresy IP następująco:
>
> ```
> 'gii' => [
>     'class' => 'yii\gii\Module',
>     'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*',
>     '192.168.178.20'] // adjust this to your needs
> ],
> ```



2.6.2 Generowanie klasy Active Record

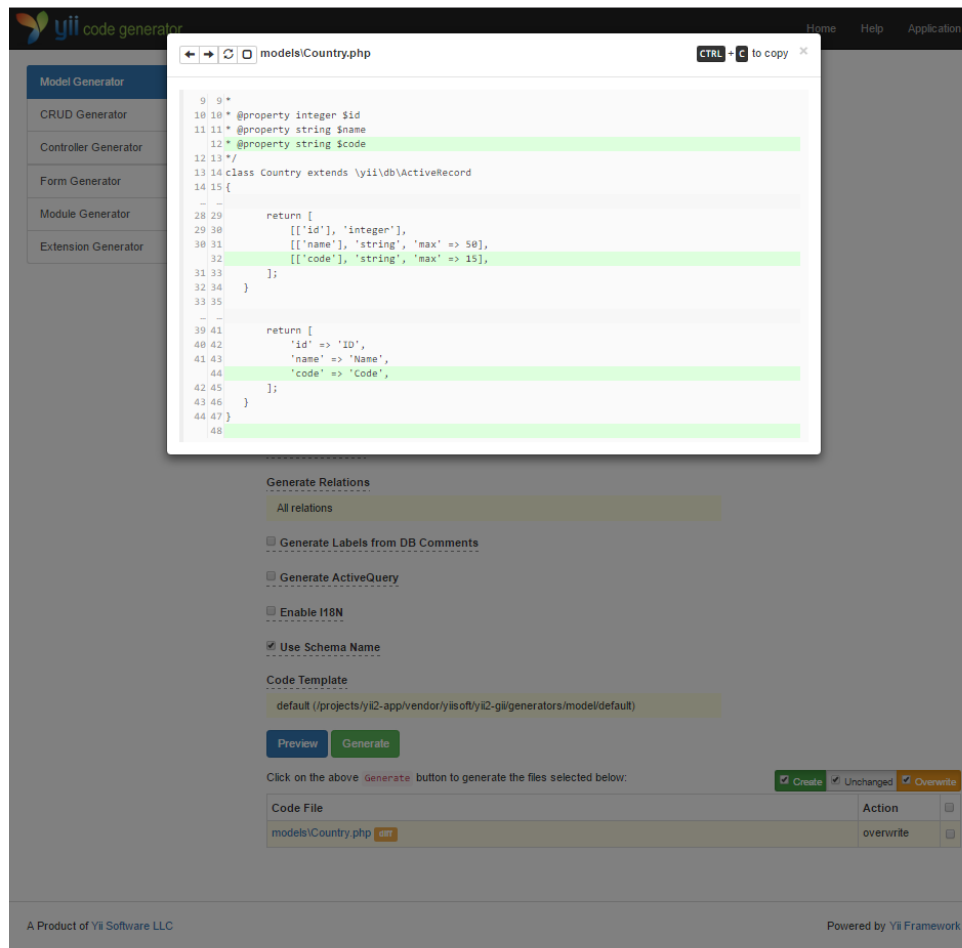
Aby użyć Gii do wygenerowania klasy Active Record, wybierz "Model Generator" z odnośników na stronie głównej Gii. Następnie uzupełnij formularz następująco:

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



Następnie kliknij przycisk "Preview". Powinieneś zauważyć na liście plik `models/Country.php`, który zostanie utworzony. Możesz kliknąć w nazwę pliku aby zobaczyć podgląd jego zawartości.

Podczas używania Gii, jeśli posiadałeś już utworzony plik o tej samej nazwie i będziesz chciał go nadpisać, kliknij w przycisk `diff` obok nazwy pliku, aby zobaczyć różnice w kodzie pomiędzy wygenerowanym, a już istniejącym plikiem.



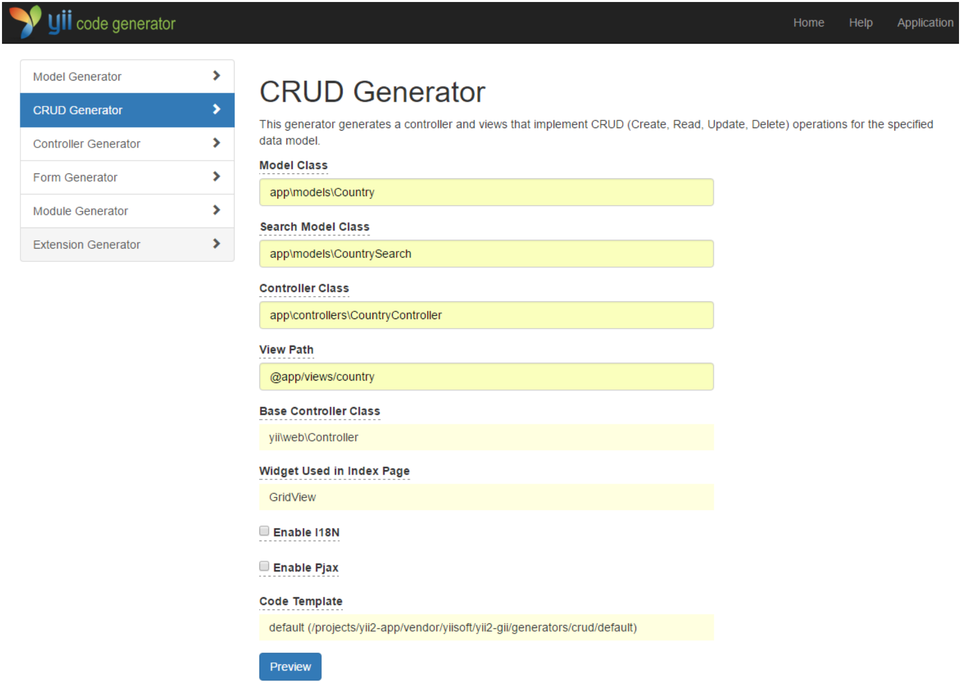
Podczas nadpisywania istniejącego już pliku musisz zaznaczyć opcję "overwrite", a następnie kliknąć przycisk "Generate". Jeśli tworzysz nowy plik, wystarczy kliknięcie "Generate".

Następnie zobaczysz stronę potwierdzającą, że kod został pomyślnie wygenerowany. Jeśli nadpisywałeś istniejący już plik, zobaczysz również wiadomość o nadpisaniu go nowo wygenerowanym kodem.

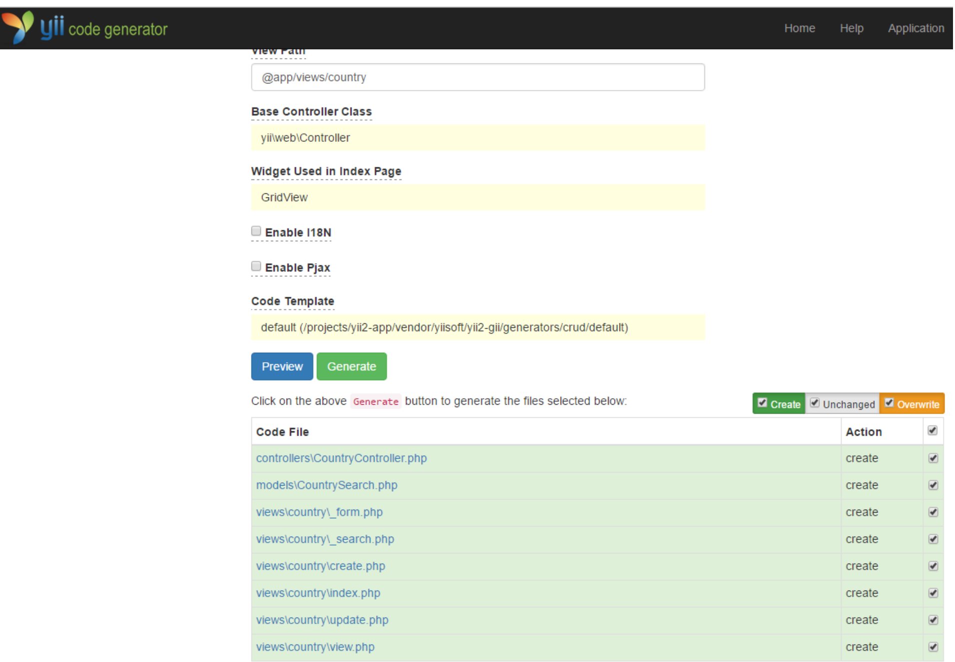
2.6.3 Generowanie kodu CRUD

Akronim CRUD pochodzi od słów *tworzenie*, *odczytywanie*, *aktualizowanie* oraz *usuwanie* (Create-Read-Update-Delete) i reprezentuje cztery podstawowe zadania dotyczące obsługi danych w większości serwisów internetowych. Aby utworzyć funkcjonalność CRUD używając Gii, wybierz na stronie głównej Gii "CRUD Generator". Do przykładu "country" uzupełnij formularz następująco:

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



Następnie kliknij przycisk "Preview". Zobaczysz listę plików, które zostaną wygenerowane, tak jak pokazano niżej.



Jeśli wcześniej utworzyłeś pliki `controllers/CountryController.php` oraz `views/country/index.php` (w sekcji baz danych tego poradnika), zaznacz opcję "overwrite", aby je zastąpić (poprzednia wersja nie posiada pełnego wsparcia CRUD).

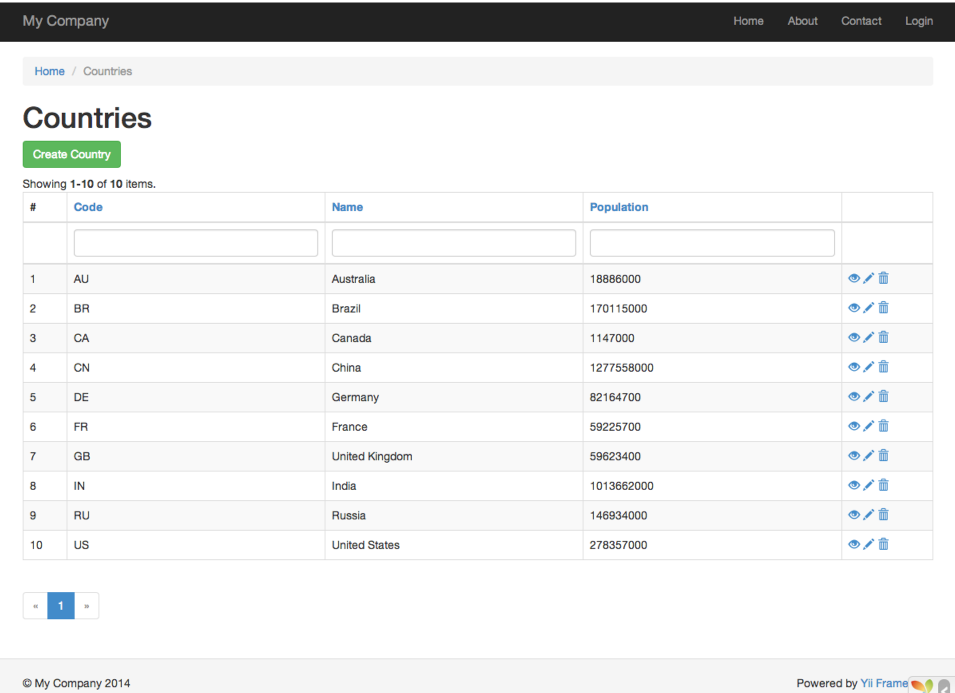
2.6.4 Sprawdzenie w działaniu

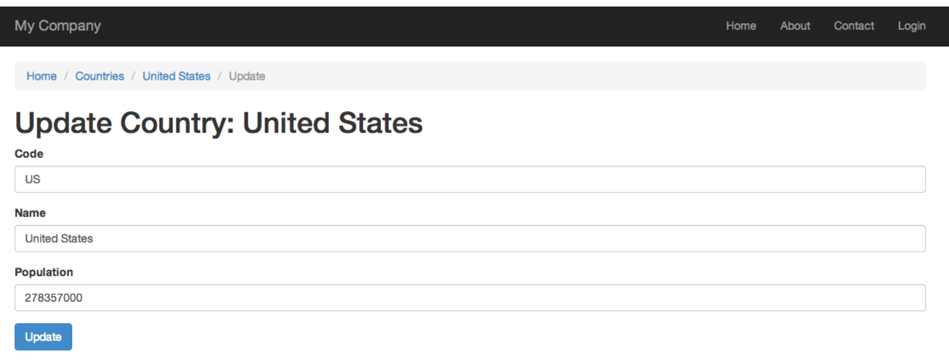
Aby zobaczyć, jak działa nowo wygenerowany kod, użyj przeglądarki, aby uzyskać dostęp do podanego adresu URL:

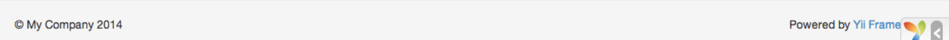
```
https://hostname/index.php?r=country%2Findex
```

Zobaczysz tabelę prezentującą kraje z bazy danych. Możesz sortować tabelę, lub filtrować ją przez wpisanie odpowiednich warunków w nagłówkach kolumn.

Dla każdego wyświetlanego kraju możesz zobaczyć jego szczegóły, zaktualizować go lub usunąć. Możesz również kliknąć przycisk "Create Country", aby przejść do formularza tworzenia nowego państwa.







Poniżej przedstawiamy listę plików wygenerowanych przez Gii, gdybyś chciał zbadać jak zostały zaimplementowane powyższe funkcjonalności (lub je edytować):

- Kontroler: `controllers/CountryController.php`
- Model: `models/Country.php` and `models/CountrySearch.php`
- Widok: `views/country/*.php`

> **Informacja:** Gii zostało zaprojektowane jako wysoce konfigurowalne i rozszerzalne narzędzie przeznaczone do generowania kodu. Prawidłowe używane może znacznie przyspieszyć szybkość tworzenia Twojej aplikacji. Po więcej szczegółów zajrzyj do sekcji Gii[36].

2.6.5 Podsumowanie

W tej sekcji nauczyłeś się jak używać Gii do wygenerowania kodu, który implementuje pełną funkcjonalność CRUD dla treści zawartch w tabelach bazy danych.

2.7 Dalsze kroki

Jeśli przebyłeś już cały dział "Pierwsze kroki", powinieneś mieć utworzoną kompletną aplikację Yii. Podczas tego procesu, nauczyłeś się jak zaimple-

[36] https://github.com/yiisoft/yii2-gii/blob/master/docs/guide/README.md

mentować zwykle niezbędne funkcjonalności, takie jak zbieranie danych od użytkownika wykorzystując formularz HTML, odczytywanie danych z bazy danych oraz wyświetlanie ich wraz ze stronicowaniem. Nauczyłeś się również jak korzystać z Gii[37] i generować kod automatycznie. Używanie Gii zmienia proces rozwoju Twojej aplikacji w tak proste zadanie, jak uzupełnienie formularza.

Ta sekcja podsumuje dostępne zasoby Yii, które pomogą Ci być bardziej produktywnym podczas używania frameworka.

- Dokumentacja
  - Przewodnik po Yii[38]: Jak sama nazwa wskazuje, jest to przewodnik, który opisuje jak działa Yii oraz dostarcza generalnych porad o użyciu Yii. Jest jednym z najważniejszych poradników, które powinieneś przeczytać przed napisaniem kodu w Yii.
  - Dokumentacja klas[39]: Określa korzystanie z każdej klasy dostarczonej przez Yii. Powinna być stosowana głównie przy pisaniu kodu oraz chęci zrozumienia, jak działa poszczególna klasa, metoda lub właściwość. Najlepiej używać jej po zrozumieniu działania całego frameworka.
  - Artykuły Wiki[40]: Artykuły w Wiki są pisane przez użytkowników Yii i są oparte na ich doświadczeniu. Większość z nich jest pisana w stylu przepisów kucharskich, pokazując jak rozwiązać poszczególne problemy używając Yii. Chociaż jakość tych artykułów może nie być tak dobra jak przewodnik, są one bardzo użyteczne oraz mogą dostarczać gotowych do użycia rozwiązań.
  - Książki[41]
- Rozszerzenia[42]: Yii może pochwalić się biblioteką tysięcy rozszerzeń użytkowników, które mogą być łatwo zainstalowane w Twojej aplikacji, dzięki czemu tworzenie Twojej aplikacji jest jeszcze prostsze i szybsze.
- Społeczność
  - Forum: `https://forum.yiiframework.com/`
  - IRC chat: Kanał #yii w sieci Libera (`ircs://irc.libera.chat:6697/yii`)
  - 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`

[37] `https://github.com/yiisoft/yii2-gii/blob/master/docs/guide-pl/README.md`
[38] `https://www.yiiframework.com/doc-2.0/guide-README.html`
[39] `https://www.yiiframework.com/doc-2.0/index.html`
[40] `https://www.yiiframework.com/wiki/?tag=yii2`
[41] `https://www.yiiframework.com/books`
[42] `https://www.yiiframework.com/extensions/`

Rozdział 3

Struktura aplikacji

3.1 Struktura aplikacji

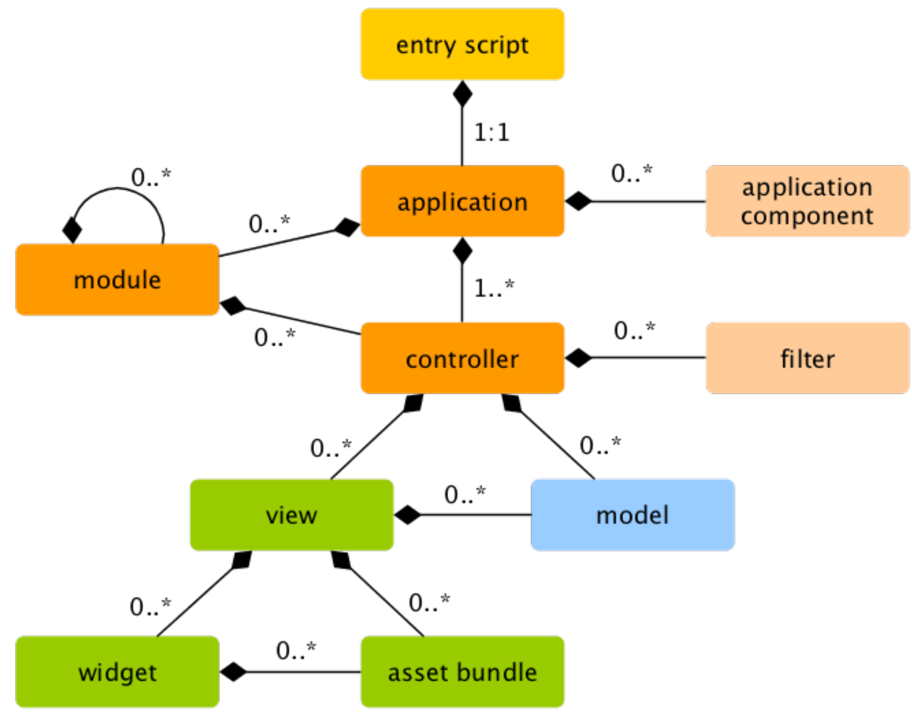
Aplikacja Yii jest zorganizowana według wzorca architektonicznego model-widok-kontroler (MVC)[1]. Modele reprezentują dane, logikę biznesową i zasady walidacji, widoki są odpowiedzialne za wyświetlanie informacji związanych z modelami, a kontrolery przyjmują dane wejściowe i przekształcają je w polecenia dla modeli i widoków.

Oprócz MVC, w aplikacjach Yii zdefiniowane są następujące struktury:

- skrypty wejściowe: skrypty PHP dostępne bezpośrednio dla użytkowników końcowych, odpowiedzialne za uruchomienie obsługi cyklu życia żądania.
- aplikacje: globalnie dostępne obiekty koordynujące działanie i zarządzające komponentami aplikacji.
- komponenty aplikacji: obiekty zarejestrowane w aplikacji, zapewniające dostępność dedykowanych usług.
- moduły: niezależne pakiety kodu zawierające kompletną wewnętrzną strukturę MVC. Aplikacja może być zorganizowana modułowo.
- filtry: reprezentują kod, który musi być wykonany przed i po obsłużeniu każdego z żądań kontrolera.
- widżety: obiekty, które mogą być dołączone w widokach. Mogą zawierać logikę kontrolera i być wykorzystane wielokrotnie w różnych miejscach.

Poniższy diagram ilustruje statyczną strukturę aplikacji:

[1] https://pl.wikipedia.org/wiki/Model-View-Controller



3.2 Skrypty wejściowe

Skrypty wejściowe są pierwszym krokiem procesu bootstrapowania aplikacji. Aplikacja (zarówno Web jak i konsolowa) posiada pojedynczy skrypt wejściowy. Użytkownicy końcowi wysyłają żądania do skryptów wejściowych, które inicjują instancje aplikacji i przekazują do nich te żądania.

Skrypty wejściowe dla aplikacji Web muszą znajdować się w folderach dostępnych dla Web, aby użytkownicy końcowi mogli je wywołać. Zwykle nazywane są `index.php`, ale mogą mieć inne nazwy pod warunkiem, że serwery Web potrafią je zlokalizować.

Skrypty wejściowe dla aplikacji konsolowych trzymane są zwykle w ścieżce głównej aplikacji i nazywane `yii` (z sufiksem `.php`). Powinny być wykonywalne, aby użytkownicy mogli uruchomić aplikacje konsolowe za pomocą komendy `./yii <ścieżka> [argumenty] [opcje]`.

Skrypty wejściowe wykonują głównie następującą pracę:

- Definiują globalne stałe,
- Rejestrują autoloader Composera[2],
- Dołączają plik klasy `Yii`,
- Ładują konfigurację aplikacji,
- Tworzą i konfigurują instancję aplikacji,

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

- Wywołują `run()`, aby przetworzyć wysłane żądanie.

3.2.1 Aplikacje Web

Poniżej znajdziesz kod skryptu wejściowego dla Podstawowego projektu szablonu Web.

```php
<?php

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

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

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

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

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

3.2.2 Aplikacje konsoli

Podobnie poniżej kod skryptu wejściowego dla aplikacji konsolowej:

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

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

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

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

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

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

3.2.3 Definiowanie stałych

Skrypty wejściowe są najlepszym miejscem do definiowania globalnych stałych. Yii wspiera następujące trzy stałe:

- `YII_DEBUG`: określa czy aplikacja działa w trybie debugowania. Podczas tego trybu aplikacja przetrzymuje więcej informacji w logach i zdradza szczegóły stosu błędów, kiedy rzucony jest wyjątek. Z tego powodu tryb debugowania powinien być używany głównie podczas fazy deweloperskiej. Domyślną wartością `YII_DEBUG` jest `false`.
- `YII_ENV`: określa środowisko, w którym aplikacja działa. Opisane jest to bardziej szczegółowo w sekcji Konfiguracje. Domyślną wartością `YII_ENV` jest `'prod'`, co oznacza, że aplikacja jest uruchomiona w środowisku produkcyjnym.
- `YII_ENABLE_ERROR_HANDLER`: określa czy uruchomić obsługę błędów przygotowaną przez Yii. Domyślną wartością tej stałej jest `true`.

Podczas definiowania stałej często korzystamy z poniższego kodu:

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

co odpowiada:

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

Jak widać pierwszy sposób jest bardziej zwięzły i łatwiejszy do zrozumienia.

Definiowanie stałych powinno odbyć się na samym początku skryptu wejściowego, aby odniosło skutek podczas dołączania pozostałych plików PHP.

Error: not existing file: structure-applications.md

3.3 Komponenty aplikacji

Aplikacje są lokatorami usług. Posiadają one zestawy *komponentów aplikacji*, które zajmują się dostarczaniem różnych serwisów do obsługi żądań. Dla przykładu, komponent `urlManager` jest odpowiedzialny za przekierowania żądań do odpowiednich kontrolerów, komponent `db` dostarcza serwisy powiązane z bazami danych, itp.

Każdy komponent aplikacji posiada unikalne ID identyfikujące go w całej aplikacji. Możesz dostać się do tego komponentu przez wyrażenie:

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

Dla przykładu, możesz użyć `\Yii::$app->db` do uzyskania `po␣łączenia z bazą danych` lub `\Yii::$app->cache` do uzyskania `dostępu do pamięci podręcznej` zarejestrowanej w aplikacji.

Komponent jest tworzony przy pierwszym jego wywołaniu przez powyższe wyrażenie, każde kolejne wywołanie zwróci tą samą instancję tego komponentu.

Komponentami aplikacji może być każdy objekt. Możesz je zarejestrować przez skonfigurowanie parametru `components` w konfiguracji aplikacji. Dla przykładu:

```
[
    'components' => [
        // rejestracja komponentu "cache" przy użyciu nazwy klasy
        'cache' => 'yii\caching\ApcCache',

        // rejestracja komponentu "db" przy użyciu tablicy konfiguracyjnej
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],

        // rejestracja komponentu "search" przy użyciu funkcji anonimowej
        'search' => function () {
            return new app\components\SolrService;
        },
    ],
]
```

> **Informacja:** Możesz zarejestrować tak wiele komponentów jak chcesz, jednak powinieneś robić to rozważnie. Komponenty aplikacji są podobne do zmiennych globalnych. Używanie zbyt wielu komponentów może potencjalnie uczynić Twój kod trudniejszym do testowania i utrzymania. W wielu przypadkach możesz po prostu utworzyć lokalny komponent i użyć go, kiedy jest to konieczne.

3.3.1 Bootstrapping komponentów

Tak, jak było wspomniane wcześniej, komponent aplikacji zostanie zinstancjowany tylko w momencie pierwszego wywołania. Czasami jednak chcemy, aby komponent został zainstancjowany dla każdego żądania, nawet jeśli nie jest bezpośrednio wywoływany. Aby to osiągnąć, możesz wylistować ID komponentów we właściwości `bootstrap` aplikacji.

Dla przykładu, następująca konfiguracja aplikacji zapewnia załadowanie komponentu `log` przy każdym żądaniu:

```
[
    'bootstrap' => [
        'log',
    ],
    'components' => [
        'log' => [
            // konfiguracja komponentu `log`
        ],
    ],
]
```

3.3.2 Podstawowe komponenty aplikacji

Yii posiada podstawowe komponenty aplikacji ze stałymi ID oraz domyślną ich konfiguracją. Dla przykładu, komponent `request` jest używany do zbierania informacji na temat żądania użytkownika oraz przekazanie go do route'a; `db` reprezentuje połączenie z bazą danych, dzięki któremu możesz wykonywać zapytania do bazy. Z pomocą tych podstawowych komponentów aplikacja jest w stanie obsłużyć żądania użytkowników.

Poniżej znajduje się lista predefiniowanych podstawowych komponentów aplikacji. Możesz je konfigurować lub zmieniać, tak jak z normalnymi komponentami. Podczas konfigurowania podstawowych komponentów aplikacji, w przypadku nie podania klasy, zostanie użyta klasa domyślna.

- `assetManager`: zarządzanie zasobami oraz ich publikacja. Po więcej informacji zajrzyj do sekcji Assets.
- `db`: reprezentuje połączenie z bazą danych, dzięki której możliwe jest wykonywanie zapytań. Konfigurując ten komponent musisz określić klasę komponentu, tak samo jak inne wymagane właściwości, np. `dsn`. Po więcej informacji zajrzyj do sekcji Obiekty dostępu do danych (DAO).
- `errorHandler`: obsługuje błędy oraz wyjątki PHP. Po więcej informacji zajrzyj do sekcji Obsługa błędów.
- `formatter`: formatuje dane wyświetlane użytkownikom. Dla przykładu liczba może zostać wyświetlona z separatorem tysięcy. Po więcej informacji zajrzyj do sekcji Formatowanie danych.
- `i18n`: wspiera tłumaczenie i formatowanie wiadomości. Po więcej informacji zajrzyj do sekcji Internacjonalizacja.

- `log`: zarządza logowaniem informacji oraz błędów Po więcej informacji zajrzyj do sekcji Logowanie.
- `yii\swiftmailer\Mailer`: wspiera tworzenie oraz wysyłanie emaili. Po więcej informacji zajrzyj do sekcji Wysyłanie poczty.
- `response`: reprezentuje odpowiedź wysyłaną do użytkowników. Po więcej informacji zajrzyj do sekcji Odpowiedzi.
- `request`: reprezentuje żądanie otrzymane od użytkownika. Po więcej informacji zajrzyj do sekcji Żądania.
- `session`: reprezentuje informacje przetrzymywane w sesji. Ten komponent jest dostępny tylko w `aplikacjach WEB`. Po więcej informacji zajrzyj do sekcji Sesje i ciasteczka.
- `urlManager`: wspiera przetwarzania oraz tworzenie adresów URL. Po więcej informacji zajrzyj do sekcji Przetwarzanie i tworzenie adresów URL
- `user`: reprezentuje informacje dotyczące uwierzytelniania użytkownika. Ten komponent jest dostępny tylko w `aplikacjach WEB`. Po więcej informacji zajrzyj do sekcji Uwierzytelnianie.
- `view`: wspiera renderowanie widoków. Po więcej informacji zajrzyj do sekcji Widoki.

Error: not existing file: structure-controllers.md

Error: not existing file: structure-models.md

Error: not existing file: structure-views.md

Error: not existing file: structure-modules.md

Error: not existing file: structure-filters.md

Error: not existing file: structure-widgets.md

Error: not existing file: structure-assets.md

Error: not existing file: structure-extensions.md

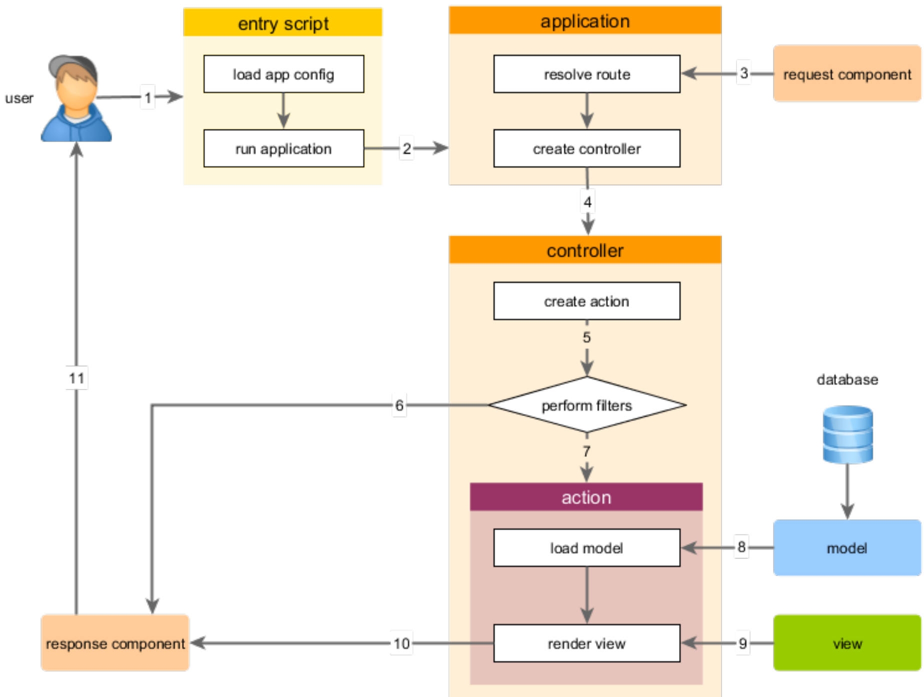
Rozdział 4

Obsługa żądań

4.1 Przegląd

Za każdym razem kiedy aplikacja Yii obsługuje żądanie, przetwarza je w podobny sposób.

1. Użytkownik wykonuje żądanie do skryptu wejściowego `web/index.php`.
2. Skrypt wejściowy ładuje konfigurację aplikacji i tworzy instancję aplikacji, aby obsłużyć zapytanie.
3. Aplikacja osiąga żądaną ścieżkę za pomocą komponentu żądania aplikacji.
4. Aplikacja tworzy instancję kontrolera, który obsłuży żądanie.
5. Kontroler tworzy instancję akcji i przetwarza filtry dla akcji.
6. Jeżeli jakikolwiek filtr się nie wykona, akcja zostanie anulowana.
7. Jeżeli wszystkie filtry przejdą, akcja zostaje wykonana.
8. Akcja wczytuje model danych, być może z bazy danych.
9. Akcja renderuje widok dostarczając go z modelem danych.
10. Wyrenderowana zawartość jest zwracana do komponentu odpowiedzi aplikacji.
11. Komponent odpowiedzi wysyła wyrenderowaną zawartość do przeglądarki użytkownika.



W tej sekcji opiszemy szczegóły dotyczące niektórych kroków przetwarzania żądania.

4.2 Bootstrapping

Bootstrapping to proces przygotowania środowiska działania aplikacji przed jej uruchomieniem, w celu przyjęcia i przetworzenia przychodzącego żądania. Mechanizm ten zachodzi w dwóch miejscach: skrypcie wejściowym i w samej aplikacji.

Skrypt wejściowy zajmuje się rejestracją autoloaderów klas dla poszczególnych bibliotek, wliczając w to autoloader Composera poprzez jego plik `autoload.php` i autoloader Yii poprzez plik klasy `Yii`. Następnie skrypt wejściowy ładuje konfigurację i tworzy instancję aplikacji.

W konstruktorze aplikacji zachodzi następujący proces bootstrappingu:

1. Wywołanie metody `preInit()`, która konfiguruje niektóre z właściwości aplikacji o wysokim priorytecie, takich jak `basePath`.

2. Rejestracja `obs␣lugi b␣łędów`.

3. Inicjalizacja właściwości aplikacji za pomocą utworzonej konfiguracji.

4. Wywołanie metody `init()`, która uruchamia proces bootstrappingu komponentów za pomocą metody `bootstrap()`.

- Załadowanie pliku manifestu dla rozszerzeń `vendor/yiisoft/extensions.php`.
- Utworzenie i uruchomienie komponentów bootstrapowych zadeklarowanych przez rozszerzenia.
- Utworzenie i uruchomienie komponentów aplikacji i/lub modułów, zadeklarowanych we właściwości bootstrapowej aplikacji.

Ponieważ proces bootstrappingu jest wykonywany przed obsługą *każdego* żądania, niezwykle istotnym jest, aby był lekki i zoptymalizowany tak bardzo, jak to tylko możliwe.

Należy unikać rejestrowania zbyt dużej ilości komponentów bootstrappingowych. Wymagane jest to tylko w przypadku, gdy taki komponent powinien uczestniczyć w całym cyklu życia obsługi przychodzącego żądania. Dla przykładu, jeśli moduł wymaga zarejestrowania dodatkowych zasad przetwarzania adresów URL, powinien być wymieniony we właściwości bootstrapowej, aby nowe zasady mogły być wykorzystane do przetworzenia żądania.

W środowisku produkcyjnym zaleca się zastosowanie pamięci podręcznej kodu, takiej jak PHP OPcache[1] lub APC[2], aby zminimalizować czas konieczny do załadowania i przetworzenia plików PHP.

Niektóre duże aplikacje posiadają bardzo skomplikowaną konfigurację, składającą się z wielu mniejszych plików konfiguracyjnych. W takim przypadku zalecane jest zapisanie w pamięci całej wynikowej tablicy konfiguracji i załadowanie jej stamtąd bezpośrednio przed utworzeniem instancji aplikacji w skrypcie wejściowym.

[1] https://www.php.net/manual/en/intro.opcache.php
[2] https://www.php.net/manual/en/book.apcu.php

Error: not existing file: runtime-routing.md

Error: not existing file: runtime-requests.md

Error: not existing file: runtime-responses.md

Error: not existing file: runtime-sessions-cookies.md

Error: not existing file: runtime-handling-errors.md

Error: not existing file: runtime-logging.md

Rozdział 5

Kluczowe koncepcje

5.1 Komponenty

Komponenty są głównym budulcem aplikacji Yii. Komponenty to instancje klasy `Component` lub jej potomnych. Trzy główne funkcjonalności, które zapewniają komponenty innym klasom to:

- Właściwości
- Eventy (zdarzenia)
- Behaviory (zachowania)

Wszystkie razem i każda z tych funkcjonalności osobno zapewnia klasom Yii o wiele większą elastyczność i łatwość użycia. Dla przykładu, dołączony `yii\jui\DatePicker`, komponent interfejsu użytkownika, może być użyty w widoku, aby wygenerować interaktywny kalendarz:

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'pl',
    'name'  => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

Właściwości widżetu są w łatwy sposób konfigurowalne ponieważ jego klasa rozszerza `Component`.

Komponenty zapewniają duże możliwości, ale przez to są też bardziej zasobożerne od standardowych obiektów, ponieważ wymagają dodatkowej pamięci i czasu CPU dla wsparcia eventów i behaviorów w szczególności. Jeśli komponent nie wymaga tych dwóch funkcjonalności, należy rozważyć rozszerzenie jego klasy z `BaseObject` zamiast `Component`. Dzięki temu komponent będzie tak samo wydajny jak standardowy obiekt PHP, ale z dodatkowym wsparciem właściwości.

Rozszerzając klasę `Component` lub `BaseObject`, zalecane jest aby przestrzegać następującej konwencji:

- Przeciążając konstruktor, dodaj parametr `$config` jako *ostatni* na liście jego argumentów i przekaż go do konstruktora rodzica.
- Zawsze wywoływuj konstruktor rodzica *na końcu* przeciążanego konstruktora.
- Przeciążając metodę `init()`, upewnij się, że wywołujesz metodę `init()` rodzica *na początku* własnej implementacji metody `init()`.

Przykład:

```php
<?php

namespace yii\components\MyClass;

use yii\base\BaseObject;

class MyClass extends BaseObject
{
    public $prop1;
    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... inicjalizacja przed zaaplikowaniem konfiguracji

        parent::__construct($config);
    }

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

        // ... inicjalizacja po zaaplikowaniu konfiguracji
    }
}
```

Postępowanie zgodnie z tymi zasadami zapewni konfigurowalność Twojego komponentu, kiedy już zostanie utworzony. Dla przykładu:

```php
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
// lub alternatywnie
$component = \Yii::createObject([
    'class' => MyClass::class,
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

> **Informacja:** Wersja z wywołaniem `Yii::createObject()` wygląda na bardziej skomplikowaną, ale jest o wiele wydajniejsza, ponieważ jej implementację zapewnia kontener wstrzykiwania zależności.

Klasa `BaseObject` wymusza następujący cykl życia obiektu:

1. Preinicjalizacja w konstruktorze. W tym miejscu można ustawić domyślne wartości właściwości.
2. Konfiguracja obiektu poprzez `$config`. Konfiguracja może nadpisać domyślne wartości ustawione w konstruktorze.
3. Postinicjalizacja w metodzie `init()`. Metoda może być przeciążona w celu normalizacji i sanityzacji właściwości.
4. Wywołanie metody obiektu.

Pierwsze trzy kroki są w całości wykonywane w konstruktorze obiektu, co oznacza, że uzyskana instancja klasy jest już poprawnie zainicjowana i stabilna.

Error: not existing file: concept-properties.md

Error: not existing file: concept-events.md

5.2 Behaviory

Behaviory są instancjami klasy `yii\base\Behavior` lub jej pochodnych. Behaviory, zwane także domieszkami[1], pozwalają na wzbogacenie funkcjonalności już istniejącej klasy `komponentu` bez konieczności modyfikacji jej struktury dziedziczenia.
Dołączenie behavioru "wstrzykuje" jego metody i właściwości do komponentu, dzięki czemu są one dostępne w taki sam sposób, jakby były zdefiniowane od razu w klasie komponentu. Ponadto behavior może reagować na eventy wywołane przez komponent, co pozwala na modyfikowanie sposobu, w jaki kod komponentu jest wykonywany.

5.2.1 Definiowane behaviorów

Aby zdefiniować behavior, stwórz klasę, która rozszerza `yii\base\Behavior` lub jej klasę potomną. Przykładowo:

```
namespace app\components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;

    public function getProp2()
    {
        return $this->_prop2;
    }

    public function setProp2($value)
    {
        $this->_prop2 = $value;
    }

    public function foo()
    {
        // ...
    }
}
```

Powyższy kod definiuje klasę behavioru `app\components\MyBehavior` z dwoma właściwościami `prop1` i `prop2` oraz jedną metodą `foo()`. Zwróć uwagę na to, że właściwość `prop2` jest zdefiniowana poprzez getter `getProp2()` i setter `setProp2()`. Jest to możliwe dzięki temu, że `yii\base\Behavior` rozszerza

[1] https://pl.wikipedia.org/wiki/Domieszka_(programowanie_obiektowe)

`yii\base\BaseObject`, przez co ma możliwość definiowania właściwości za pomocą getterów i setterów.

Komponent, po załączeniu tego behavioru, będzie również posiadał właściwości `prop1` i `prop2` oraz metodę `foo()`.

> **Wskazówka:** Wewnątrz behavioru możesz odwołać się do komponentu, do którego jest on załączony, przez właściwość `yii\base\Behavior::$owner`.

> **Uwaga:** Jeśli nadpisujesz metody `yii\base\Behavior::__get()` i/lub `yii\base\Behavior::__set()` behavioru, musisz również nadpisać `yii\base\Behavior::canGetProperty()` i/lub `yii\base\Behavior::canSetProperty()`.

5.2.2 Obsługa eventów komponentu

Jeśli behavior powinien reagować na eventy wywołane przez komponent, do którego jest załączony, należy nadpisać jego metodę `yii\base\Behavior::events()`. Dla przykładu:

```
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)
    {
        // ...
    }
}
```

Metoda `events()` powinna zwrócić listę eventów i odpowiadających im uchwytów. Powyższy przykład deklaruje, że event `EVENT_BEFORE_VALIDATE` istnieje i jego uchwytem jest metoda `beforeValidate()`. Do określenia uchwytów eventów możesz użyć następujących formatów:

- łańcuch znaków odnoszący się do nazwy metody w klasie behavioru, jak w przykładzie powyżej,

- tablica obiektu lub nazwy klasy i nazwy metody w postaci łańcucha znaków (bez nawiasów), np. `[$obiekt, 'nazwaMetody']`,
- funkcja anonimowa.

Sygnatura funkcji uchwytu eventu powinna wyglądać jak poniżej, gdzie `$event` odwołuje się do obsługiwanego eventu. W sekcji Eventy znajdziesz więcej szczegółów dotyczących samych eventów.

```
function ($event) {
}
```

5.2.3 Załączanie behaviorów

Możesz załączyć behavior do **komponentu** zarówno statycznie, jak i dynamicznie. Pierwszy sposób jest częściej wykorzystywany w praktyce.

Aby załączyć behavior statycznie, nadpisz metodę `behaviors()` w klasie komponentu, do której behavior ma być załączony. Metoda `behaviors()` powinna zwracać listę konfiguracji behaviorów.
Każda konfiguracja behavioru może być zarówno nazwą klasy behavioru jak i tablicą konfiguracyjną:

```
namespace app\models;

use yii\db\ActiveRecord;
use app\components\MyBehavior;

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // anonimowy behavior, tylko nazwa klasy behavioru
            MyBehavior::class,

            // imienny behavior, tylko nazwa klasy behavioru
            'myBehavior2' => MyBehavior::class,

            // anonimowy behavior, tablica konfiguracyjna
            [
                'class' => MyBehavior::class,
                'prop1' => 'value1',
                'prop2' => 'value2',
            ],

            // imienny behavior, tablica konfiguracyjna
            'myBehavior4' => [
                'class' => MyBehavior::class,
                'prop1' => 'value1',
                'prop2' => 'value2',
            ],
        ];
```

```
    }
}
```

Możesz przypisać konkretną nazwę dla behavioru, definiując klucz tablicy odpowiadający jego konfiguracji - w tym przypadku mówimy o *imiennym behaviorze*. W powyższym przykładzie znajdują się dwa imienne behaviory: `myBehavior2` i `myBehavior4`. Jeśli behavior nie ma przypisanej nazwy, nazywamy go *anonimowym*.

Aby załączyć behavior dynamicznie, wywołaj metodę `yii\base\Component::attachBehavior()` na komponencie, do którego behavior ma być załączony:

```
use app\components\MyBehavior;

// za␣łącz obiekt behavioru
$component->attachBehavior('myBehavior1', new MyBehavior);

// za␣łącz klasę behavioru
$component->attachBehavior('myBehavior2', MyBehavior::class);

// za␣łącz tablicę konfiguracyjną
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::class,
    'prop1' => 'value1',
    'prop2' => 'value2',
]);
```

Możesz załączyć wiele behaviorów jednocześnie, korzystając z metody `yii\base\Component::attachBehaviors()`:

```
$component->attachBehaviors([
    'myBehavior1' => new MyBehavior,  // imienny behavior
    MyBehavior::class,          // anonimowy behavior
]);
```

Możliwe jest również załączenie behaviorów poprzez konfigurację, jak widać to poniżej:

```
[
    'as myBehavior2' => MyBehavior::class, // zwróć uwagę na konstrukcję "as
    nazwaBehavioru"

    'as myBehavior3' => [
        'class' => MyBehavior::class,
        'prop1' => 'value1',
        'prop2' => 'value2',
    ],
]
```

Więcej szczegółów znajdziesz w sekcji Konfiguracje.

5.2.4 Korzystanie z behaviorów

Aby użyć behavioru, najpierw załącz go do `komponentu` zgodnie z powyższymi instrukcjami. Kiedy behavior jest już załączony, korzystanie z niego jest bardzo proste.

Możesz uzyskać dostęp do *publicznej* zmiennej lub właściwości zdefiniowanej przez getter i/lub setter behavioru z poziomu komponentu, do którego jest on załączony:

```
// "prop1" jest w␣laściwością zdefiniowaną w klasie behavioru
echo $component->prop1;
$component->prop1 = $value;
```

Możesz również wywołać *publiczną* metodę behavioru w podobny sposób:

```
// foo() jest publiczną metodą zdefiniowaną w klasie behavioru
$component->foo();
```

Jak widać, pomimo że `$component` nie definiuje `prop1` ani `foo()`, można ich użyć tak, jakby były zdefiniowane przez komponent, dzięki załączonemu behaviorowi.

Jeśli dwa behaviory definiują tą samą właściwość lub metodę i oba są załączone do tego samego komponentu, behavior, który został załączony jako *pierwszy*, będzie obsługiwał wywołaną właściwość lub metodę.

Behavior może być powiązany z konkretną nazwą podczas załączania do komponentu - w takim przypadku można odwołać się do obiektu behavioru, korzystając z jego nazwy:

```
$behavior = $component->getBehavior('myBehavior');
```

Można również uzyskać listę wszystkich behaviorów załączonych do komponentu:

```
$behaviors = $component->getBehaviors();
```

5.2.5 Odłączanie behaviorów

Aby odłączyć behavior, wywołaj metodę `yii\base\Component::detachBehavior()` z nazwą przypisaną temu behaviorowi:

```
$component->detachBehavior('myBehavior1');
```

Można również odłączyć *wszystkie* behaviory jednocześnie:

```
$component->detachBehaviors();
```

5.2.6 Korzystanie z

Behavior `yii\behaviors\TimestampBehavior` pozwala na automatyczne aktualizowanie atrybutów znaczników czasu dla modelu `Active Record` za każdym razem, gdy model jest zapisywany za pomocą metod `insert()`, `update()` lub `save()`.

Załącz ten behavior do klasy `Active Record`, której chcesz użyć:

```
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'],
                ],
                // opcjonalnie jeśli używasz kolumny typu datetime zamiast
                uniksowego znacznika czasu:
                // 'value' => new Expression('NOW()'),
            ],
        ];
    }
}
```

Powyższa konfiguracja behavioru określa, co powinno stać się z wierszem danych podczas:

- dodawania; behavior powinien ustawić aktualny uniksowy znacznik czasu dla atrybutów `created_at` i `updated_at`.
- aktualizacji; behavior powinien ustawić aktualny uniksowy znacznik czasu dla atrybutu `updated_at`.

> **Uwaga:** Aby powyższa implementacja zadziałała dla bazy danych MySQL, zadeklaruj kolumny (`created_at`, `updated_at`) jako int(11) na potrzeby przechowania uniksowego znacznika czasu.

Z tak wprowadzonym kodem, po zapisie obiektu `User` zobaczysz, że jego atrybuty `created_at` i `updated_at` zostały automatycznie ustawione na aktualny uniksowy znacznik czasu:

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
echo $user->created_at;  // wyświetli znacznik czasu z momentu zapisu
```

`TimestampBehavior` oferuje również użyteczną metodę `touch()`, która ustawia aktualny znacznik czasu określonemu atrybutowi i zapisuje go w bazie danych:

```
$user->touch('login_time');
```

5.2.7 Inne behaviory

Poniżej znajdziesz kilka behaviorów wbudowanych lub też dostępnych w zewnętrznych bibliotekach:

- `yii\behaviors\BlameableBehavior` - automatycznie wypełnia wskazane atrybuty ID aktualnego użytkownika.
- `yii\behaviors\SluggableBehavior` - automatycznie wypełnia wskazany atrybut wartością, która może być użyta jako poprawna część adresu URL (slug).
- `yii\behaviors\AttributeBehavior` - automatycznie ustawia określoną wartość jednemu lub więcej atrybutom obiektu ActiveRecord w momencie wystąpienia konkretnych eventów.
- yii2tech\ar\softdelete\SoftDeleteBehavior[2] - udostępnia metody do "miękkiego usunięcia" ActiveRecordu i przywrócenia go z powrotem np. poprzez ustawienie flagi lub statusu oznaczającego rekord jako usunięty.
- yii2tech\ar\position\PositionBehavior[3] - pozwala na zarządzanie kolejnością rekordów w polu typu integer.

5.2.8 Różnice pomiędzy behaviorami a traitami

Pomimo że behaviory są podobne do traitów[4] w taki sposób, że również "wstrzykują" swoje właściwości i metody do klasy, struktury te różnią się w wielu aspektach. Obie mają swoje wady i zalety, jak opisano to poniżej, i powinny być raczej traktowane jako swoje uzupełnienia, a nie alternatywy.

Zalety używania behaviorów

Klasy behaviorów, tak jak zwyczajne klasy, pozwalają na dziedziczenie. Traity można raczej nazwać wspieranym przez język programowania "kopiuj-wklej", jako że nie oferują dziedziczenia.

Behaviory można załączać i odłączać od komponentu dynamicznie bez konieczności modyfikowania klasy komponentu.

[2] `https://github.com/yii2tech/ar-softdelete`
[3] `https://github.com/yii2tech/ar-position`
[4] `https://www.php.net/traits`

Aby użyć traita, konieczne jest zmodyfikowanie kodu klasy, która będzie go używać.

Behaviory są konfigurowalne w przeciwieństwie do traitów.

Behaviory mogą modyfikować wykonywanie kodu komponentu, poprzez reagowanie na jego eventy.

W przypadku, gdy zdarza się konflikt nazw pomiędzy różnymi behaviorami załączonymi do tego samego komponentu, jest on automatycznie rozwiązywany przez przyznanie pierwszeństwa behaviorowi załączonemu jako pierwszy.
Konflikty nazw spowodowane przez różne traity wymagają ręcznego rozwiązania poprzez zmianę nazw dotkniętych problemem właściwości i metod.

Zalety używania traitów

Traity są znacznie bardziej wydajne niż behaviory, ponieważ behaviory są obiektami, które wymagają czasu i pamięci.

Środowiska IDE o wiele lepiej wspierają traity, ponieważ są one natywnymi konstruktami języka programowania.

Error: not existing file: concept-configurations.md

5.3 Aliasy

Aliasy używane są do reprezentowania ścieżek do plików lub adresów URL i pozwalają uniknąć konieczności wielokrotnego definiowania ich w kodzie aplikacji. Alias musi zaczynać się od znaku `@`, dla odróżnienia od zwykłej ścieżki i adresu URL. W przypadku zdefiniowania aliasu bez tego znaku, będzie on automatycznie dodany na początku.

Yii korzysta z wielu predefiniowanych aliasów. Dla przykładu, alias `@yii` reprezentuje ścieżkę instalacji frameworka, a `@web` bazowy adres URL aktualnie uruchomionej aplikacji Web.

5.3.1 Definiowanie aliasów

Możesz zdefiniować alias do ścieżki pliku lub adresu URL wywołując `Yii::setAlias()`:

```
// alias do ścieżki pliku
Yii::setAlias('@foo', '/path/to/foo');

// alias do adresu URL
Yii::setAlias('@bar', 'https://www.example.com');

// alias istniejącego pliku, zawierającego klasę \foo\Bar
Yii::setAlias('@foo/Bar.php', '/zdecydowanie/nie/foo/Bar.php');
```

> **Uwaga:** *nie* jest konieczne, aby aliasowana ścieżka pliku lub URL wskazywał istniejący plik lub zasób.

Mając już zdefiniowany alias, możesz rozbudować go, tworząc nowy alias (bez konieczności wywołania `Yii::setAlias()`), dodając ukośnik `/` i kolejne segmenty ścieżki. Aliasy zdefiniowane za pomocą `Yii::setAlias()` nazywane są *bazowymi aliasami*, a te, które je rozbudowują, *aliasami pochodnymi*. Dla przykładu, `@foo` jest aliasem bazowym, a `@foo/bar/file.php` pochodnym.

Możesz definiować aliasy używając innych aliasów (zarówno bazowych, jak i pochodnych):

```
Yii::setAlias('@foobar', '@foo/bar');
```

Aliasy bazowe są zwykle definiowane podczas fazy bootstrappingu. Możliwe jest wywołanie `Yii::setAlias()` już w skrypcie wejściowym. Aplikacja dla wygody deweloperów posiada właściwość `aliases`, którą można zmodyfikować w konfiguracji:

```
return [
    // ...
    'aliases' => [
        '@foo' => '/path/to/foo',
        '@bar' => 'https://www.example.com',
    ],
];
```

5.3.2 Rozwiązywanie aliasów

Możesz wywołać `Yii::getAlias()`, aby rozwiązać alias, czyli zamienić go na ścieżkę pliku lub adres URL, który reprezentuje. Dotyczy to zarówno bazowych aliasów, jak i pochodnych:

```
echo Yii::getAlias('@foo');               // wyświetla: /ścieżka/do/foo
echo Yii::getAlias('@bar');               // wyświetla:
https://www.example.com
echo Yii::getAlias('@foo/bar/file.php');  // wyświetla:
/ścieżka/do/foo/bar/file.php
```

Ścieżka/URL reprezentowany przez pochodny alias jest ustalany poprzez zamianę części z bazowym aliasem na jego rozwiązaną reprezentację.

> **Uwaga:** Metoda `Yii::getAlias()` nie sprawdza, czy reprezentowana ścieżka/URL wskazuje na istniejący plik lub zasób.

Alias bazowy może również zawierać ukośnik `/`. Metoda `Yii::getAlias()` potrafi określić, która część aliasu jest aliasem bazowym i prawidłowo określić odpowiadającą mu ścieżkę pliku lub URL:

```
Yii::setAlias('@foo', '/path/to/foo');
Yii::setAlias('@foo/bar', '/path2/bar');
Yii::getAlias('@foo/test/file.php');  // wyświetla:
/path/to/foo/test/file.php
Yii::getAlias('@foo/bar/file.php');   // wyświetla: /path2/bar/file.php
```

Gdyby `@foo/bar` nie był zdefiniowany jako bazowy alias, ostatnia instrukcja wyświetliłaby `/path/to/foo/bar/file.php`.

5.3.3 Korzystanie z aliasów

Aliasy są rozwiązywane automatycznie w wielu miejscach w Yii bez konieczności wywołania bezpośrednio metody `Yii::getAlias()`. Przykładowo, `yii\caching\FileCache::$cachePath` akceptuje zarówno ścieżkę pliku, jak i alias ją reprezentujący, odróżniając je od siebie dzięki prefiksowi `@`.

```
use yii\caching\FileCache;

$cache = new FileCache([
    'cachePath' => '@runtime/cache',
]);
```

Aby sprawdzić, czy dana właściwość lub parametr metody wspierają użycie aliasów, zapoznaj się z dokumentacją API.

5.3.4 Predefiniowane aliasy

Yii predefiniuje zestaw aliasów do łatwego wskazywania często używanych ścieżek plików i adresów URL:

- `@yii`, folder, w którym znajduje się plik `BaseYii.php` (nazywany także folderem frameworka).
- `@app`, `bazowa ścieżka` aktualnie używanej aplikacji.
- `@runtime`, `ścieżka cyklu życia` aktualnie używanej aplikacji. Domyślnie wskazuje na `@app/runtime`.
- `@webroot`, folder bazowy Web aktualnie używanej aplikacji Web. Określany jest jako lokalizacja folderu zawierającego skrypt wejścia.
- `@web`, bazowy adres URL aktualnie używanej aplikacji Web. Wskazuje na tą samą wartość co `yii\web\Request::$baseUrl`.
- `@vendor`, `folder pakietów composera`. Domyślnie wskazuje na `@app/vendor`.
- `@bower`, bazowy folder zawierający pakiety bowera[5]. Domyślnie wskazuje na `@vendor/bower`.
- `@npm`, bazowy folder zawierający pakiety npm[6]. Domyślnie wskazuje na `@vendor/npm`.

Alias `@yii` jest definiowany poprzez dołączenie pliku `Yii.php` w skrypcie wejścia. Pozostałe aliasy są definiowane w konstruktorze aplikacji podczas ładowania konfiguracji.

> **Uwaga:** aliasy `@web` i `@webroot`, zgodnie z ich opisami, są zdefiniowane w `aplikacji Web` i z tego powodu niedostępne domyślnie w `aplikacji konsolowej`.

5.3.5 Aliasy rozszerzeń

Instalacja rozszerzenia za pomocą composera automatycznie definiuje dla niego alias. Każdy z takich aliasów jest nazwany zgodnie z bazową przestrzenią nazw rozszerzenia określonej w swoim pliku `composer.json` i reprezentuje bazowy folder pakietu. Dla przykładu, instalując rozszerzenie `yiisoft/yii2-jui` automatycznie uzyskasz alias `@yii/jui` zdefiniowany podczas fazy bootstrappingu, będący odpowiednikiem wywołania:

```
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

5.4 Autoładowanie klas

Yii opiera się na mechanizmie automatycznego ładowania klas[7] służącym do zlokalizowania i dołączenia wszystkich wymaganych plików klas. Wbudo-

[5] https://bower.io/
[6] https://www.npmjs.com/
[7] https://www.php.net/manual/pl/language.oop5.autoload.php

wany wysoce wydajny autoloader klas, zgodny ze standardem PSR-4[8], jest instalowany po załączeniu pliku `Yii.php`.

> **Uwaga:** Dla uproszczenia opisów, w tej sekcji zostanie omówione jedynie autoładowanie klas. Należy mieć jednak na uwadze, że poniższe informacje odnoszą się również do autoładowania interfejsów i traitów.

5.4.1 Korzystanie z autoloadera Yii

Aby skorzystać z autoloadera klas Yii, powinieneś przestrzegać dwóch prostych zasad tworzenia i nazywania własnych klas:

- Każda klasa musi znajdować się w przestrzeni nazw[9] (np. `foo\bar\MyClass`)
- Każda klasa musi być zapisana jako oddzielny plik, do którego ścieżka określona jest poniższym algorytmem:

```
// $className jest w pe␣lni uściśloną nazwą klasy bez początkowego
odwrotnego ukośnika
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) .
'.php');
```

Przykładowo, jeśli nazwa klasy i przestrzeń nazw to `foo\bar\MyClass`, odpowiadającym ścieżce pliku klasy aliasem jest `@foo/bar/MyClass.php`. Aby ten alias mógł być przetłumaczony na ścieżkę pliku, `@foo` lub `@foo/bar` musi być aliasem bazowym.

Używając podstawowego szablonu projektu, możesz umieścić swoje klasy w bazowej przestrzeni nazw `app`, dzięki czemu mogą być autoładowane przez Yii bez potrzeby definiowania nowego aliasu. Dzieje się tak dzięki temu, że `@app` jest predefiniowanym aliasem i klasa `app\components\MyClass` może być odszukana w pliku `AppBasePath/components/MyClass.php`, zgodnie z opisanym algorytmem.

W zaawansowanym szablonie projektu[10] każdy poziom aplikacji posiada swój własny bazowy alias. Dla przykładu, front-end określony jest przez bazowy alias `@frontend`, a back-end - `@backend`. Dzięki temu możesz umieścić klasy front-endu w przestrzeni nazw `frontend`, a klasy back-endu w przestrzeni nazw `backend`. Wszystkie te klasy będą automatycznie załadowane przez autoloader Yii.

5.4.2 Mapa klas

Autoloader klas Yii wspiera mechanizm *mapy klas*, która mapuje nazwy klas do odpowiadających im ścieżek plików. Kiedy autoloader ładuje klasę, naj-

[8] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md

[9] https://www.php.net/manual/pl/language.namespaces.php

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

pierw sprawdza czy klasa znajduje się w mapie. Jeśli tak, odpowiadająca nazwie ścieżka pliku zostanie dołączona od razu, bez dalszej weryfikacji, co jest powodem, dla którego autoładowanie klas jest błyskawiczne. Wszystkie podstawowe klasy Yii są autoładowane właśnie w ten sposób.

Możesz dodać klasę do mapy klas, przechowywanej w `Yii::$classMap`, za pomocą instrukcji:

```
Yii::$classMap['foo\bar\MyClass'] = 'path/to/MyClass.php';
```

Do określenia ścieżek plików klas można użyć aliasów. Zapisywanie mapy klas powinno odbywać się w procesie bootstrappingu, aby mapa była gotowa zanim rozpocznie się korzystanie z klas.

5.4.3 Korzystanie z innych autoloaderów

Ponieważ Yii opiera się głównie na composerze, jako menedżerze pakietów zależności, zalecane jest również zainstalowanie autoloadera composera. Jeśli używasz zewnętrznych bibliotek, korzystających z własnych autoloaderów, powinieneś również je zainstalować.

Używając autoloadera Yii razem z innymi autoloaderami, powinieneś dołączyć plik `Yii.php` *po* wszystkich pozostałych autoloaderach. Dzięki temu autoloader Yii jako pierwszy odpowie na żądanie autoładowania klasy. Dla przykładu, poniższy kod znajduje się w skrypcie wejściowym podstawowego szablonu projektu. Pierwsza linia jest instrukcją instalacji autoloadera composera, a druga instaluje autoloader Yii:

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

Możesz używać jedynie autoloadera composera bez autoloadera Yii, ale wydajność autoładowania klas może być wtedy obniżona i, dodatkowo, musisz przestrzegać zasad ustalonych przez composera, aby Twoje klasy mogły być autoładowane.

> **Informacja:** Jeśli nie chcesz korzystać z autoloadera Yii, musisz stworzyć swoją własną wersję pliku `Yii.php` i dołączyć ją w skrypcie wejściowym.

5.4.4 Autoładowanie klas rozszerzeń

Autoloader Yii potrafi również automatycznie ładować klasy rozszerzeń. Jedynym wymaganiem ze strony rozszerzenia jest prawidłowy zapis sekcji `autoload` w swoim pliku `composer.json`. Szczegóły na temat specyfikacji `autoload` znajdują się dokumentacji composera[11].

Jeśli nie korzystasz z autoloadera Yii, autoloader composera załaduje dla Ciebie automatycznie klasy rozszerzeń.

[11] https://getcomposer.org/doc/04-schema.md#autoload

Error: not existing file: concept-service-locator.md

Error: not existing file: concept-di-container.md

Rozdział 6

Praca z bazami danych

Error: not existing file: db-dao.md

Error: not existing file: db-query-builder.md

6.1 Active Record

Active Record[1] zapewnia zorientowany obiektowo interfejs dostępu i manipulacji danymi zapisanymi w bazie danych. Klasa typu Active Record jest powiązana z tabelą bazodanową, a instacja tej klasy odpowiada pojedynczemu wierszowi w tabeli - *atrybut* obiektu Active Record reprezentuje wartość konkretnej kolumny w tym wierszu. Zamiast pisać bezpośrednie kwerendy bazy danych, można skorzystać z atrybutów i metod klasy Active Record.

Dla przykładu, załóżmy, że `Customer` jest klasą Active Record, powiązaną z tabelą `customer` i `name` jest kolumną w tabeli `customer`. Aby dodać nowy wiersz do tabeli `customer`, wystarczy wykonać następujący kod:

```
$customer = new Customer();
$customer->name = 'Qiang';
$customer->save();
```

Kod z przykładu jest odpowiednikiem poniższej komendy SQL dla MySQL, która jest mniej intuicyjna, bardziej podatna na błędy i, co bardzo prawdopodobne, niekompatybilna z innymi rodzajami baz danych:

```
$db->createCommand('INSERT INTO `customer` (`name`) VALUES (:name)', [
    ':name' => 'Qiang',
])->execute();
```

Yii zapewnia wsparcie Active Record dla następujących typów relacyjnych baz danych:

- MySQL 4.1 lub nowszy: poprzez `yii\db\ActiveRecord`
- PostgreSQL 8.4 lub nowszy: poprzez `yii\db\ActiveRecord`
- SQLite 2 i 3: poprzez `yii\db\ActiveRecord`
- Microsoft SQL Server 2008 lub nowszy: poprzez `yii\db\ActiveRecord`
- Oracle: poprzez `yii\db\ActiveRecord`
- CUBRID 9.3 lub nowszy: poprzez `yii\db\ActiveRecord` (zwróć uwagę, że z powodu błędu[2] w rozszerzeniu PDO cubrid, umieszczanie wartości w cudzysłowie nie będzie działać, zatem wymagane jest zainstalowanie CUBRID 9.3 zarówno jako klienta jak i serwer)
- Sphinx: poprzez `yii\sphinx\ActiveRecord`, wymaga rozszerzenia `yii2-sphinx`
- ElasticSearch: poprzez `yii\elasticsearch\ActiveRecord`, wymaga rozszerzenia `yii2-elasticsearch`

Dodatkowo Yii wspiera również Active Record dla następujących baz danych typu NoSQL:

- Redis 2.6.12 lub nowszy: poprzez `yii\redis\ActiveRecord`, wymaga rozszerzenia `yii2-redis`

---
[1] https://en.wikipedia.org/wiki/Active_record_pattern
[2] https://jira.cubrid.org/browse/APIS-658

- MongoDB 1.3.0 lub nowszy: poprzez `yii\mongodb\ActiveRecord`, wymaga rozszerzenia `yii2-mongodb`

W tej sekcji przewodnika opiszemy sposób użycia Active Record dla baz relacyjnych, jednakże większość zagadnień można zastosować również dla NoSQL.

6.1.1 Deklarowanie klas Active Record

Na początek zadeklaruj klasę typu Active Record rozszerzając `ActiveRecord`.
Deklarowanie nazwy tabeli

Domyślnie każda klasa Active Record jest powiązana ze swoją tabelą w bazie danych. Metoda `tableName()` zwraca nazwę tabeli konwertując nazwę klasy za pomocą `yii\helpers\Inflector::camel2id()`. Możesz przeciążyć tę metodę, jeśli tabela nie jest nazwana zgodnie z tą konwencją.

Identycznie zastosowany może być domyślny prefiks tabeli `tablePrefix`. Przykładowo, jeśli `tablePrefix` to `tbl_`, tabelą klasy `Customer` staje się `tbl_customer`, a dla `OrderItem` jest to `tbl_order_item`.

Jeśli nazwa tabeli zostanie podana jako `{{%NazwaTabeli}}`, znak procent `%` zostanie zamieniony automatycznie na prefiks tabeli. Dla przykładu, `{{%post}}` staje się `{{tbl_post}}`. Nawiasy wokół nazwy tabeli są używane dla odpowiedniego podawania nazw w kwerendach SQL.

W poniższym przykładzie deklarujemy klasę Active Record nazwaną `Customer` dla tabeli `customer` w bazie danych.

```
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    const STATUS_INACTIVE = 0;
    const STATUS_ACTIVE = 1;

    /**
     * @return string nazwa tabeli powiązanej z klasą ActiveRecord.
     */
    public static function tableName()
    {
        return '{{customer}}';
    }
}
```

Aktywne rekordy nazywane są "modelami"

Instancje Active Record są traktowane jak modele. Z tego powodu zwykle dodajemy klasy Active Record do przestrzeni nazw `app\models` (lub innej, przeznaczonej dla klas modeli).

Dzięki temu, że `ActiveRecord` rozszerza `Model`, dziedziczy *wszystkie* funkcjonalności modelu, takie jak atrybuty, zasady walidacji, serializację danych itd.

6.1.2 Łączenie się z bazą danych

Domyślnie Active Record używa komponentu aplikacji `db` jako `po␣łączenia z bazą danych`, do uzyskania dostępu i manipulowania jej danymi. Jak zostało to już wyjaśnione w sekcji Obiekty dostępu do danych (DAO), komponent `db` można skonfigurować w pliku konfiguracyjnym aplikacji jak poniżej:

```
return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
        ],
    ],
];
```

Jeśli chcesz użyć innego połączenia do bazy danych niż za pomocą komponentu `db`, musisz nadpisać metodę `getDb()`:

```
class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        // użyj komponentu aplikacji "db2"
        return \Yii::$app->db2;
    }
}
```

6.1.3 Kwerendy

Po zadeklarowaniu klasy Active Record, możesz użyć jej do pobrania danych z powiązanej tabeli bazy danych. Proces ten zwykle sprowadza się do następujących trzech kroków:

1. Stworzenie nowego obiektu kwerendy za pomocą metody `find()`;
2. Zbudowanie obiektu kwerendy za pomocą metod konstruktora kwerend;
3. Wywołanie metod kwerendy w celu uzyskania danych jako instancji klasy Active Record.

Jak widać, procedura jest bardzo podobna do tej używanej przy konstruktorze kwerend. Jedyna różnica jest taka, że zamiast użycia operatora `new` do stworzenia obiektu kwerendy, wywołujemy metodę `find()`, która zwraca nowy obiekt kwerendy klasy `ActiveQuery`.

Poniżej znajdziesz kilka przykładów pokazujących jak używać Active Query do pobierania danych:

```
// zwraca pojedynczego klienta o ID 123
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::find()
    ->where(['id' => 123])
    ->one();

// zwraca wszystkich aktywnych klientów posortowanych po ID
// SELECT * FROM `customer` WHERE `status` = 1 ORDER BY `id`
$customers = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->orderBy('id')
    ->all();

// zwraca liczbę aktywnych klientów
// SELECT COUNT(*) FROM `customer` WHERE `status` = 1
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// zwraca wszystkich klientów w tablicy zaindeksowanej wg ID
// SELECT * FROM `customer`
$customers = Customer::find()
    ->indexBy('id')
    ->all();
```

W powyższych przykładach `$customer` jest obiektem typu `Customer`, a `$customers` jest tablicą obiektów typu `Customer`. W obu przypadkach dane pobrane są z tabeli `customer`.

> **Informacja:** Dzięki temu, że `ActiveQuery` rozszerza klasę `Query`, możesz użyć *wszystkich* metod dotyczących kwerend i ich budowania opisanych w sekcji Konstruktor kwerend.

Ponieważ zwykle kwerendy korzystają z zapytań zawierających klucz główny lub też zestaw wartości dla kilku kolumn, Yii udostępnia dwie skrótowe metody, pozwalające na szybsze ich użycie:

- `findOne()`: zwraca pojedynczą instancję klasy Active Record, zawierającą dane z pierwszego pobranego odpowiadającego zapytaniu wiersza danych.
- `findAll()`: zwraca tablicę instancji klasy Active Record zawierających *wszystkie* wyniki zapytania.

Obie metody mogą przyjmować jeden z następujących formatów parametrów:

- wartość skalarna: wartość jest traktowana jako wartość klucza głównego, który należy odszukać. Yii automatycznie ustali, która kolumna jest kluczem głównym, odczytując informacje ze schematu bazy.
- tablica wartości skalarnych: tablica jest traktowana jako lista poszukiwanych wartości klucza głównego.
- tablica asocjacyjna: klucze tablicy są poszukiwanymi nazwami kolumn a wartości tablicy są odpowiadającymi im wartościami kolumn. Po więcej szczegółów zajrzyj do rozdziału Format asocjacyjny.

Poniższy kod pokazuje, jak mogą być użyte opisane metody:

```
// zwraca pojedynczego klienta o ID 123
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// zwraca klientów o ID 100, 101, 123 i 124
// SELECT * FROM `customer` WHERE `id` IN (100, 101, 123, 124)
$customers = Customer::findAll([100, 101, 123, 124]);

// zwraca aktywnego klienta o ID 123
// SELECT * FROM `customer` WHERE `id` = 123 AND `status` = 1
$customer = Customer::findOne([
    'id' => 123,
    'status' => Customer::STATUS_ACTIVE,
]);

// zwraca wszystkich nieaktywnych klientów
// SELECT * FROM `customer` WHERE `status` = 0
$customers = Customer::findAll([
    'status' => Customer::STATUS_INACTIVE,
]);
```

> **Uwaga:** Ani metoda `findOne()` ani `one()` nie dodaje `LIMIT 1` do wygenerowanej kwerendy SQL. Jeśli zapytanie może zwrócić więcej niż jeden wiersz danych, należy wywołać bezpośrednio `limit(1)`, w celu zwiększenia wydajności aplikacji, np. `Customer::find()->limit(1)->one()`.

Oprócz korzystania z metod konstruktora kwerend możesz również użyć surowych zapytań SQL w celu pobrania danych do obiektu Active Record za pomocą metody `findBySql()`:

```
// zwraca wszystkich nieaktywnych klientów
$sql = 'SELECT * FROM customer WHERE status=:status';
$customers = Customer::findBySql($sql, [':status' =>
Customer::STATUS_INACTIVE])->all();
```

Nie wywołuj dodatkowych metod konstruktora kwerend po wywołaniu `findBySql()`, ponieważ zostaną one pominięte.

6.1.4 Dostęp do danych

Jak wspomniano wyżej, dane pobrane z bazy danych są dostępne w obiekcie Active Record i każdy wiersz wyniku zapytania odpowiada pojedynczej instancji Active Record. Możesz odczytać wartości kolumn odwołując się do atrybutów obiektu Active Record, dla przykładu:

```
// "id" i "email" są nazwami kolumn w tabeli "customer"
$customer = Customer::findOne(123);
$id = $customer->id;
$email = $customer->email;
```

> **Uwaga:** nazwy atrybutów Active Record odpowiadają nazwom powiązanych z nimi kolumn z uwzględnieniem wielkości liter. Yii automatycznie definiuje atrybut Active Record dla każdej kolumny powiązanej tabeli. NIE należy definiować ich własnoręcznie.

Ponieważ atrybuty Active Record nazywane są zgodnie z nazwami kolumn, możesz natknąć się na kod PHP typu `$customer->first_name`, gdzie podkreślniki używane są do oddzielenia poszczególnych słów w nazwach atrybutów, w przypadku, gdy kolumny tabeli nazywane są właśnie w ten sposób. Jeśli masz wątpliwości dotyczące spojności takiego stylu programowania, powinieneś zmienić odpowiednio nazwy kolumn tabeli (używając np. formatowania typu "camelCase").

Transformacja danych

Często zdarza się, że dane wprowadzane i/lub wyświetlane zapisane są w formacie różniącym się od tego używanego w bazie danych. Dla przykładu, w bazie danych przechowywane są daty urodzin klientów jako uniksowe znaczniki czasu, podczas gdy w większości przypadków pożądana forma zapisu daty to `'RRRR/MM/DD'`. Aby osiągnąć ten format, można zdefiniować metody *transformujące dane* w klasie `Customer`:

```
class Customer extends ActiveRecord
{
    // ...

    public function getBirthdayText()
    {
        return date('Y/m/d', $this->birthday);
    }

    public function setBirthdayText($value)
    {
        $this->birthday = strtotime($value);
    }
}
```

Od tego momentu, w kodzie PHP, zamiast odwołać się do `$customer->birthday`, można użyć `$customer->birthdayText`, co pozwala na wprowadzenie i wyświetlenie daty urodzin klienta w formacie `'RRRR/MM/DD'`.

> **Wskazówka:** Powyższy przykład pokazuje podstawowy sposób transformacji danych. Podczas zwyczajowej pracy z formularzami danych można skorzystać z DateValidator i `yii\jui\DatePicker`, co jest prostsze w użyciu i daje więcej możliwości.

Pobieranie danych jako tablice

Pobieranie danych jako obiekty Active Record jest wygodne i elastyczne, ale nie zawsze pożądane, zwłaszcza kiedy konieczne jest uzyskanie ogromnej liczby danych, z powodu użycia sporej ilości pamięci. W takim przypadku można pobrać dane jako tablicę PHP, wywołując metodę `asArray()` przed wykonaniem kwerendy:

```
// zwraca wszystkich klientów
// każdy klient jest zwracany w postaci tablicy asocjacyjnej
$customers = Customer::find()
    ->asArray()
    ->all();
```

> **Uwaga:** Powyższy sposób zwiększa wydajność aplikacji i pozwala na zmniejszenie zużycia pamięci, ale ponieważ jest on znacznie bliższy niskiej warstwie abstrakcji DB, traci się większość funkcjonalności Active Record. Bardzo ważną różnicą jest zwracany typ danych dla wartości kolumn. Kiedy dane zwracane są jako obiekt Active Record, wartości kolumn są automatycznie odpowiednio rzutowane zgodnie z typem kolumny; przy danych zwracanych jako tablice wartości kolumn są zawsze typu string (jako rezultat zapytania PDO bez żadnego przetworzenia), niezależnie od typu kolumny.

Pobieranie danych seriami

W sekcji Konstruktor kwerend wyjaśniliśmy, że można użyć *kwerendy serii*, aby zmniejszyć zużycie pamięci przy pobieraniu dużej ilości danych z bazy. Tej samej techniki można użyć w przypadku Active Record. Dla przykładu:

```
// pobiera dziesięciu klientów na raz
foreach (Customer::find()->batch(10) as $customers) {
    // $customers jest tablicą dziesięciu lub mniej obiektów Customer
}

// pobiera dziesięciu klientów na raz i iteruje po nich pojedynczo
foreach (Customer::find()->each(10) as $customer) {
    // $customer jest obiektem Customer
```

```
}

// kwerenda seryjna z gorliwym ␣ladowaniem
foreach (Customer::find()->with('orders')->each() as $customer) {
    // $customer jest obiektem Customer
}
```

6.1.5 Zapisywanie danych

Używając Active Record możesz w łatwy sposób zapisać dane w bazie, w następujących krokach:

1. Przygotowanie instancji Active Record
2. Przypisanie nowych wartości do atrybutów Active Record
3. Wywołanie metody `save()` w celu zapisania danych w bazie.

Przykład:

```
// dodaj nowy wiersz danych
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();

// zaktualizuj istniejący wiersz danych
$customer = Customer::findOne(123);
$customer->email = 'james@newexample.com';
$customer->save();
```

Metoda `save()` może zarówno dodawać jak i aktualizować wiersz danych, w zależności od stanu instacji Active Record. Jeśli instancja została dopiero utworzona poprzez operator `new`, wywołanie `save()` spowoduje dodanie nowego wiersza. Jeśli instacja jest wynikiem użycia kwerendy, wywołanie `save()` zaktualizuje wiersz danych powiązanych z instancją.

Można odróżnić dwa stany instancji Active Record sprawdzając wartość jej właściwości `isNewRecord`. Jest ona także używana przez `save()` w poniższy sposób:

```
public function save($runValidation = true, $attributeNames = null)
{
    if ($this->getIsNewRecord()) {
        return $this->insert($runValidation, $attributeNames);
    } else {
        return $this->update($runValidation, $attributeNames) !== false;
    }
}
```

> **Wskazówka:** Możesz również wywołać `insert()` lub `update()` bezpośrednio, aby, odpowiednio, dodać lub uaktualnić wiersz.

Walidacja danych

Dzięki temu, że `ActiveRecord` rozszerza klasę `Model`, korzysta z tych samych mechanizmów walidacji danych. Możesz definiować zasady walidacji nadpisując metodę `rules()` i uruchamiać procedurę walidacji wywołując metodę `validate()`.

Wywołanie `save()` automatycznie wywołuje również metodę `validate()`. Dopiero po pomyślnym przejściu walidacji rozpocznie się proces zapisywania danych; w przeciwnym wypadku zostanie zwrócona flaga `false` - komunikaty z błędami walidacji można odczytać sprawdzając właściwość `errors`.

> **Wskazówka:** Jeśli masz pewność, że dane nie potrzebują przechodzić procesu walidacji (np. pochodzą z zaufanych źródeł), możesz wywołać `save(false)`, aby go pominąć.

Masowe przypisywanie

Tak jak w zwyczajnych modelach, instancje Active Record posiadają również mechanizm masowego przypisywania. Funkcjonalność ta umożliwia przypisanie wartości wielu atrybutom Active Record za pomocą pojedynczej instrukcji PHP, jak pokazano to poniżej. Należy jednak pamiętać, że w ten sposób mogą być przypisane tylko bezpieczne atrybuty.

```
$values = [
    'name' => 'James',
    'email' => 'james@example.com',
];

$customer = new Customer();

$customer->attributes = $values;
$customer->save();
```

Aktualizowanie liczników

Jednym z częstych zadań jest zmniejszanie lub zwiększanie wartości kolumny w tabeli bazy danych. Takie kolumny nazywamy licznikami. Metoda `updateCounters()` służy do aktualizacji jednego lub wielu liczników. Przykład:

```
$post = Post::findOne(100);

// UPDATE `post` SET `view_count` = `view_count` + 1 WHERE `id` = 100
$post->updateCounters(['view_count' => 1]);
```

> **Uwaga:** Jeśli używasz `save()` do aktualizacji licznika, możesz otrzymać nieprawidłowe rezultaty, ponieważ jest możliwe, że ten sam licznik zostanie odczytany i zapisany jednocześnie przez wiele zapytań.

Brudne atrybuty

Kiedy wywołujesz `save()`, aby zapisać instancję Active Record, tylko *brudne atrybuty* są zapisywane. Atrybut uznawany jest za *brudny* jeśli jego wartość została zmodyfikowana od momentu pobrania z bazy danych lub ostatniego zapisu. Pamiętaj, że walidacja danych zostanie przeprowadzona niezależnie od tego, czy instancja Active Record zawiera brudne atrybuty czy też nie.

Active Record automatycznie tworzy listę brudnych atrybutów, poprzez porównanie starej wartości atrybutu do aktualnej. Możesz wywołać metodę `getDirtyAttributes()`, aby otrzymać najnowszą listę brudnych atrybutów. Dodatkowo można wywołać `markAttributeDirty()`, aby oznaczyć konkretny atrybut jako brudny.

Jeśli chcesz sprawdzić wartość atrybutu sprzed ostatniej zmiany, możesz wywołać `getOldAttributes()` lub `getOldAttribute()`.

> **Uwaga:** Porównanie starej i nowej wartości atrybutu odbywa się za pomocą operatora `===`, zatem atrybut zostanie uznany za brudny nawet jeśli ma tą samą wartość, ale jest innego typu. Taka sytuacja zdarza się często, kiedy model jest aktualizowany danymi pochodzącymi z formularza HTML, gdzie każda wartość jest reprezentowana jako string. Aby upewnić się, że wartości będą odpowiednich typów, np. integer, możesz zaaplikować filtr walidacji: `['attributeName', 'filter', 'filter' => 'intval']`. Działa on z wszystkimi funkcjami PHP rzutującymi typy jak intval()[3], floatval()[4], boolval[5], itp...

Domyślne wartości atrybutów

Niektóre z kolumn tabeli bazy danych mogą mieć przypisane domyślne wartości w bazie danych. W przypadku, gdy chcesz wypełnić takimi wartościami formularz dla instancji Active Record, zamiast ponownie ustawiać wszystkie domyślne wartości, możesz wywołać metodę `loadDefaultValues()`, która przypisze wszystkie domyślne wartości odpowiednim atrybutom:

```
$customer = new Customer();
$customer->loadDefaultValues();
// $customer->xyz otrzyma domyślną wartość, zadeklarowaną przy definiowaniu
kolumny "xyz"
```

Rzutowanie typów atrybutów

Po wypełnieniu rezultatem kwerendy, `yii\db\ActiveRecord` przeprowadza automatyczne rzutowanie typów na wartościach swoich atrybutów, używając

[3] https://www.php.net/manual/en/function.intval.php
[4] https://www.php.net/manual/en/function.floatval.php
[5] https://www.php.net/manual/en/function.boolval.php

do tego celu informacji zawartych w schemacie tabeli bazy danych. Pozwala to na prawidłowe przedstawienie danych pobranych z kolumny tabeli zadeklarowanej jako liczba całkowita, w postaci wartości typu PHP integer w instancji klasy ActiveRecord (typu boolean jako boolean itp.). Mechanizm rzutowania ma jednak kilka ograniczeń:

- Wartości typu zmiennoprzecinkowego nie są konwertowane na float, a zamiast tego są przedstawiane jako łańcuch znaków, aby zachować dokładność ich liczbowej prezentacji.
- Konwersja typu integer zależy od zakresu liczb całkowitych używanego systemu operacyjnego. Wartości kolumn zadeklarowanych jako 'unsigned integer' lub 'big integer' będą przekonwertowane do PHP integer tylko na systemach 64-bitowych, a na 32-bitowych będą przedstawione jako łańcuchy znaków.

Zwróć uwagę na to, że rzutowanie typów jest wykonywane tylko podczas wypełniania instancji ActiveRecord rezultatem kwerendy. Automatyczna konwersja nie jest przeprowadzana dla wartości załadowanych poprzez żądanie HTTP lub ustawionych bezpośrednio dla właściwości klasy. Schemat tabeli będzie również użyty do przygotowania instrukcji SQL przy zapisywaniu danych ActiveRecord, aby upewnić się, że wartości są przypisane w kwerendzie z prawidłowymi typami. Atrybuty instancji ActiveRecord nie będą jednak przekonwertowane w procesie zapisywania.

> **Wskazówka:** możesz użyć `yii\behaviors\AttributeTypecastBehavior`, aby skonfigurować proces rzutowania typów dla wartości atrybutów w momencie ich walidacji lub zapisu.

Aktualizowanie wielu wierszy jednocześnie

Metody przedstawione powyżej działają na pojedynczych instancjach Active Record, dodając lub aktualizując indywidualne wiersze tabeli. Aby uaktualnić wiele wierszy jednocześnie, należy wywołać statyczną metodę `updateAll()`.

```
// UPDATE `customer` SET `status` = 1 WHERE `email` LIKE `%@example.com%`
Customer::updateAll(['status' => Customer::STATUS_ACTIVE], ['like', 'email',
'@example.com']);
```

W podobny sposób można wywołać `updateAllCounters()`, aby uaktualnić liczniki wielu wierszy w tym samym czasie.

```
// UPDATE `customer` SET `age` = `age` + 1
Customer::updateAllCounters(['age' => 1]);
```

6.1.6 Usuwanie danych

Aby usunąć pojedynczy wiersz danych, utwórz najpierw instancję Active Record odpowiadającą temu wierszowi, a następnie wywołaj metodę `delete()`.

```
$customer = Customer::findOne(123);
$customer->delete();
```

Możesz również wywołać `deleteAll()`, aby usunąć kilka lub wszystkie wiersze danych. Dla przykładu:

```
Customer::deleteAll(['status' => Customer::STATUS_INACTIVE]);
```

> **Uwaga:** Należy być bardzo ostrożnym przy wywoływaniu `deleteAll()`, ponieważ w efekcie można całkowicie usunąć wszystkie dane z tabeli bazy, jeśli popełni się błąd przy ustalaniu warunków dla metody.

6.1.7 Cykl życia Active Record

Istotnym elementem pracy z Yii jest zrozumienie cyklu życia Active Record w zależności od metodyki jego użycia. Podczas każdego cyklu wykonywane są określone sekwencje metod i aby dopasować go do własnych potrzeb, wystarczy je nadpisać. Można również śledzić i odpowiadać na eventy Active Record uruchamiane podczas cyklu życia, aby wstrzyknąć swój własny kod. Takie eventy są szczególnie użyteczne podczas tworzenia wpływających na cykl życia behaviorów Active Record.

Poniżej znajdziesz wyszczególnione cykle życia Active Record wraz z metodami/eventami, które są w nie zaangażowane.

Cykl życia nowej instancji

Podczas tworzenia nowej instancji Active Record za pomocą operatora `new`, zachodzi następujący cykl:

1. Konstruktor klasy.
2. `init()`: uruchamia event `EVENT_INIT`.

Cykl życia przy pobieraniu danych

Podczas pobierania danych za pomocą jednej z metod kwerendy, każdy świeżo wypełniony obiekt Active Record przechodzi następujący cykl:

1. Konstruktor klasy.
2. `init()`: uruchamia event `EVENT_INIT`.
3. `afterFind()`: uruchamia event `EVENT_AFTER_FIND`.

Cykl życia przy zapisywaniu danych

Podczas wywołania `save()`, w celu dodania lub uaktualnienia danych instancji Active Record, zachodzi następujący cykl:

1. `beforeValidate()`: uruchamia event `EVENT_BEFORE_VALIDATE`. Jeśli metoda zwróci `false` lub właściwość `isValid` ma wartość `false`, kolejne kroki są pomijane.
2. Proces walidacji danych. Jeśli proces zakończy się niepowodzeniem, kolejne kroki po kroku 3. są pomijane.
3. `afterValidate()`: uruchamia event `EVENT_AFTER_VALIDATE`.
4. `beforeSave()`: uruchamia event `EVENT_BEFORE_INSERT` lub `EVENT_BEFORE_UPDATE`. Jeśli metoda zwróci `false` lub właściwość `isValid` ma wartość `false`, kolejne kroki są pomijane.
5. Proces właściwego dodawania lub aktulizowania danych.
6. `afterSave()`: uruchamia event `EVENT_AFTER_INSERT` lub `EVENT_AFTER_UPDATE`.

Cykl życia przy usuwaniu danych

Podczas wywołania `delete()`, w celu usunięcia danych instancji Active Record, zachodzi następujący cykl:

1. `beforeDelete()`: uruchamia event `EVENT_BEFORE_DELETE`. Jeśli metoda zwróci `false` lub właściwość `isValid` ma wartość `false`, kolejne kroki są pomijane.
2. Proces właściwego usuwania danych.
3. `afterDelete()`: uruchamia event `EVENT_AFTER_DELETE`.

   Uwaga: Wywołanie poniższych metod NIE uruchomi żadnego z powyższych cykli:

   - `updateAll()`
   - `deleteAll()`
   - `updateCounters()`
   - `updateAllCounters()`

Odświeżanie cyklu życia danych

Wywołanie `refresh()` w celu odświeżenia instancji Active Record, uruchamia event `EVENT_AFTER_REFRESH`, o ile odświeżenie się powiedzie i metoda zwróci `true`.

6.1.8 Praca z transakcjami

Są dwa sposoby użycia transakcji podczas pracy z Active Record.

Pierwszy zakłada bezpośrednie ujęcie wywołań metod Active Record w blok transakcji, jak pokazano to poniżej:

```
$customer = Customer::findOne(123);

Customer::getDb()->transaction(function($db) use ($customer) {
    $customer->id = 200;
    $customer->save();
    // ...inne operacje bazodanowe...
});

// lub alternatywnie

$transaction = Customer::getDb()->beginTransaction();
try {
    $customer->id = 200;
    $customer->save();
    // ...inne operacje bazodanowe...
    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

> **Uwaga:** w powyższym kodzie znajdują się dwa bloki catch dla kompatybilności z PHP 5.x i PHP 7.x. `\Exception` implementuje interfejs `\Throwable`[6] od PHP 7.0, zatem można pominąć część z `\Exception`, jeśli Twoja aplikacja używa tylko PHP 7.0 lub wyższego.

Drugi sposób polega na utworzeniu listy operacji bazodanowych, które wymagają transakcji za pomocą metody `transactions()`. Dla przykładu:

```
class Customer extends ActiveRecord
{
    public function transactions()
    {
        return [
            'admin' => self::OP_INSERT,
            'api' => self::OP_INSERT | self::OP_UPDATE | self::OP_DELETE,
            // powyższy zapis jest odpowiednikiem następującego skróconego:
            // 'api' => self::OP_ALL,
        ];
    }
}
```

[6] https://www.php.net/manual/en/class.throwable.php

Metoda `transactions()` powinna zwracać tablicę, której klucze są nazwami scenariuszy, a wartości to operacje bazodanowe, które powinny być objęte transakcją. Używaj następujących stałych do określenia typu operacji:

- `OP_INSERT`: operacja dodawania wykonywana za pomocą `insert()`;
- `OP_UPDATE`: operacja aktualizacji wykonywana za pomocą `update()`;
- `OP_DELETE`: operacja usuwania wykonywana za pomocą `delete()`.

Używaj operatora `|`, aby podać więcej niż jedną operację za pomocą powyższych stałych. Możesz również użyć stałej dla skróconej definicji wszystkich trzech powyższych operacji `OP_ALL`.

6.1.9 Optymistyczna blokada

Optymistyczne blokowanie jest jednym ze sposobów uniknięcia konfliktów, które mogą wystąpić, kiedy pojedynczy wiersz danych jest aktualizowany przez kilku użytkowników. Dla przykładu, użytkownik A i użytkownik B edytują artykuł wiki w tym samym czasie - po tym jak użytkownik A zapisał już swoje zmiany, użytkownik B klika przycisk "Zapisz", aby również wykonać identyczną operację. Ponieważ użytkownik B pracował w rzeczywistości na "starej" wersji artykułu, byłoby wskazane powstrzymać go przed nadpisaniem wersji użytkownika A i wyświelić komunikat wyjaśniający sytuację.

Optymistyczne blokowanie rozwiązuje ten problem za pomocą dodatkowej kolumny w bazie przechowującej numer wersji każdego wiersza. Kiedy taki wiersz jest zapisywany z wcześniejszym numerem wersji niż aktualna rzucany jest wyjątek `StaleObjectException`, który powstrzymuje zapis wiersza. Optymistyczne blokowanie może być użyte tylko przy aktualizacji lub usuwaniu istniejącego wiersza za pomocą odpowiednio `update()` lub `delete()`.

Aby skorzystać z optymistycznej blokady:

1. Stwórz kolumnę w tabeli bazy danych powiązaną z klasą Active Record do przechowywania numeru wersji każdego wiersza. Kolumna powinna być typu big integer (przykładowo w MySQL `BIGINT DEFAULT 0`).

2. Nadpisz metodę `optimisticLock()`, aby zwrócić nazwę tej kolumny.

3. W formularzu pobierającym dane od użytkownika, dodaj ukryte pole, gdzie przechowasz aktualny numer wersji uaktualnianego wiersza. Upewnij się, że atrybut wersji ma dodaną zasadę walidacji i przechodzi poprawnie jej proces.

4. W akcji kontrolera uaktualniającej wiersz za pomocą Active Record, użyj bloku try-catch, aby wyłapać wyjątek `StaleObjectException`. Zaimplemetuj odpowiednią logikę biznesową (np. scalenie zmian, wyświetlenie komunikatu o nieaktualnej wersji, itp.), aby rozwiązać konflikt.

Dla przykładu, załóżmy, że kolumna wersji nazywa się `version`. Implementację optymistycznego blokowania można wykonać za pomocą następującego kodu:

```
// ------ kod widoku -------

use yii\helpers\Html;

// ...inne pola formularza
echo Html::activeHiddenInput($model, 'version');


// ------ kod kontrolera -------

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) {
        // logika rozwiązująca konflikt
    }
}
```

6.1.10 Praca z danymi relacji

Oprócz korzystania z indywidualnych tabel bazy danych, Active Record umożliwia również na uzyskanie danych relacji, pozwalając na odczytanie ich z poziomu głównego obiektu. Dla przykładu, dane klienta są powiązane relacją z danymi zamówienia, ponieważ jeden klient może złożyć jedno lub wiele zamówień. Odpowiednio deklarując tę relację, można uzyskać dane zamówienia klienta, używając wyrażenia `$customer->orders`, które zwróci informacje o zamówieniu klienta jako tablicę instancji `Order` typu Active Record.

Deklarowanie relacji

Aby móc pracować z relacjami używając Active Record, najpierw musisz je zadeklarować w obrębie klasy. Deklaracja odbywa się za pomocą utworzenia prostej *metody relacyjnej* dla każdej relacji osobno, jak w przykładach poniżej:

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

W powyższym kodzie, zadeklarowano relację `orders` dla klasy `Customer` i relację `customer` dla klasy `Order`.

Każda metoda relacyjna musi mieć nazwę utworzoną według wzoru `getXyz`. `xyz` (pierwsza litera jest mała) jest *nazwą relacji*. Zwróć uwagę na to, że nazwy relacji *uwzględniają wielkość liter*.

Deklarując relację powinno się zwrócić uwagę na następujące dane:

- mnogość relacji: określona przez wywołanie odpowiednio `hasMany()` lub `hasOne()`. W powyższym przykładzie można łatwo zobaczyć w definicji relacji, że klient może mieć wiele zamówień, podczas gdy zamówienie ma tylko jednego klienta.
- nazwę powiązanej klasy Active Record: określoną jako pierwszy argument w `hasMany()` lub `hasOne()`. Rekomendowany sposób uzyskania nazwy klasy to wywołanie `Xyz::class`, dzięki czemu możemy posiłkować się wsparciem autouzupełniania IDE i wykrywaniem błędów na poziomie kompilacji.
- powiązanie pomiędzy dwoma rodzajami danych: określone jako kolumna(y), poprzez którą dane nawiązują relację. Wartości tablicy są kolumnami głównych danych (reprezentowanymi przez klasę Active Record, w której deklaruje się relacje), a klucze tablicy są kolumnami danych relacyjnych.

  Aby łatwo opanować technikę deklarowania relacji wystarczy zapamiętać, że kolumnę należącą do relacyjnej klasy Active Record zapisuje się zaraz obok jej nazwy (jak to widać w przykładzie powyżej - `customer_id` jest właściwością `Order` a `id` jest właściwością `Customer`).

Uzyskiwanie dostępu do danych relacji

Po zadeklarowaniu relacji, możesz uzyskać dostęp do danych poprzez jej nazwę. Odbywa się to w taki sam sposób jak uzyskiwanie dostępu do właściwości obiektu zdefiniowanego w metodzie relacyjnej. Właśnie dlatego też nazywamy je *właściwościami relacji*. Przykład:

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);
```

```
// SELECT * FROM `order` WHERE `customer_id` = 123
// $orders jest tablicą obiektów typu Order
$orders = $customer->orders;
```

> **Informacja:** Deklarując relację o nazwie `xyz` poprzez metodę-getter `getXyz()`, uzyskasz dostęp do `xyz` jak do właściwości obiektu. Zwróć uwagę na to, że nazwa uwzględnia wielkość liter.

Jeśli relacja jest zadeklarowana poprzez `hasMany()`, zwraca tablicę powiązanych instancji Active Record; jeśli deklaracja odbywa się poprzez `hasOne()`, zwraca pojedynczą powiązaną instancję Active Record lub wartość `null`, w przypadku, gdy nie znaleziono powiązanych danych.

Podczas pierwszego odwołania się do właściwości relacji wykonywana jest kwerenda SQL, tak jak pokazano to w przykładzie powyżej. Odwołanie się do tej samej właściwości kolejny raz zwróci poprzedni wynik, bez wykonywanie ponownie kwerendy. Aby wymusić wykonanie kwerendy w takiej sytuacji, należy najpierw usunąć z pamięci właściwość relacyjną poprzez `unset($customer->orders)`.

> **Uwaga:** Pomimo podobieństwa mechanizmu relacji do właściwości obiektu, jest tutaj znacząca różnica. Wartości właściwości zwykłych obiektów są tego samego typu jak definiująca je metoda-getter. Metoda relacyjna zwraca jednak instancję `ActiveQuery`, a właściwości relacji są instancjami `ActiveRecord` lub tablicą takich obiektów.
>
> ```
> $customer->orders; // tablica obiektów `Order`
> $customer->getOrders(); // instancja ActiveQuery
> ```
>
> Taka funkcjonalność jest użyteczna przy tworzeniu kwerend dostosowanych do potrzeb programisty, co opisane jest w następnej sekcji.

Dynamiczne kwerendy relacyjne

Dzięki temu, że metoda relacyjna zwraca instancję `ActiveQuery`, możliwe jest dalsze rozbudowanie takiej kwerendy korzystając z metod konstruowania kwerend. Dla przykładu:

```
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `subtotal` > 200 ORDER BY `id`
$orders = $customer->getOrders()
    ->where(['>', 'subtotal', 200])
    ->orderBy('id')
    ->all();
```

Inaczej niż w przypadku właściwości relacji, za każdym razem, gdy wywyłujesz dynamiczną kwerendę relacyjną poprzez metodę relacji, wykonywane jest zapytanie do bazy, nawet jeśli identyczna kwerenda została już wywołana wcześniej.

Możliwe jest także sparametryzowanie deklaracji relacji, dzięki czemu można w łatwiejszy sposób wykonywać relacyjne kwerendy. Dla przykładu, możesz zadeklarować relację `bigOrders` jak to pokazano poniżej:

```
class Customer extends ActiveRecord
{
    public function getBigOrders($threshold = 100)
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id'])
            ->where('subtotal > :threshold', [':threshold' => $threshold])
            ->orderBy('id');
    }
}
```

Dzięki czemu możesz wykonać następujące relacyjne kwerendy:

```
// 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;
```

Relacje za pomocą tabeli węzła

W projekcie bazy danych, kiedy połączenie pomiędzy dwoma relacyjnymi tabelami jest typu wiele-do-wielu, zwykle stosuje się tzw. tabelę węzła[7]. Dla przykładu, tabela `order` i tabela `item` mogą być powiązane poprzez węzeł nazwany `order_item`. Jedno zamówienie będzie posiadało wiele produktów zamówienia (pozycji), a każdy indywidualny produkt będzie także powiązany z wieloma pozycjami zamówienia.

Deklarując takie relacje, możesz wywołać zarówno metodę `via()` jak i `viaTable()`, aby określić tabelę węzła. Różnica pomiędzy `via()` i `viaTable()` jest taka, że pierwsza metoda definiuje tabelę węzła dla istniejącej nazwy relacji, podczas gdy druga definiuje bezpośrednio węzeł. Przykład:

```
class Order extends ActiveRecord
{
    public function getItems()
    {
        return $this->hasMany(Item::class, ['id' => 'item_id'])
            ->viaTable('order_item', ['order_id' => 'id']);
    }
}
```

[7] https://en.wikipedia.org/wiki/Junction_table

lub alternatywnie,

```
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');
    }
}
```

Sposób użycia relacji zadeklarowanych z pomocą tabeli węzła jest taki sam jak dla zwykłych relacji. Dla przykładu:

```
// 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 (...)
// zwraca tablicę obiektów Item
$items = $order->items;
```

Pobieranie leniwe i gorliwe

W sekcji Uzyskiwanie dostępu do danych relacji wyjaśniliśmy, że można uzyskać dostęp do właściwości relacji instancji Active Record w identyczny sposób jak w przypadku zwykłych właściwości obiektu. Kwerenda SQL zostanie wykonana tylko w momencie pierwszego odwołania się do właściwości relacji. Taki sposób uzyskiwania relacyjnych danych nazywamy *pobieraniem leniwym*. Przykład:

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$orders = $customer->orders;

// bez wykonywania zapytania SQL
$orders2 = $customer->orders;
```

Leniwe pobieranie jest bardzo wygodne w użyciu, może jednak powodować spadek wydajności aplikacji, kiedy konieczne jest uzyskanie dostępu do tej samej relacyjnej właściwości dla wielu instancji Active Record. Rozważmy poniższy przykład - ile zapytań SQL zostanie wykonanych?

```
// SELECT * FROM `customer` LIMIT 100
$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {
    // SELECT * FROM `order` WHERE `customer_id` = ...
    $orders = $customer->orders;
}
```

Jak wynika z opisu powyżej, zostanie wykonanych aż 101 kwerend SQL! Dzieje się tak, ponieważ za każdym razem, gdy uzyskujemy dostęp do właściwości relacyjnej `orders` dla kolejnego obiektu `Customer` w pętli, wykonywane jest nowe zapytanie SQL.

Aby rozwiązać ten wydajnościowy problem, należy użyć tak zwanego *gorliwego pobierania*, jak w przykładzie poniżej:

```
// SELECT * FROM `customer` LIMIT 100;
// SELECT * FROM `orders` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->with('orders')
    ->limit(100)
    ->all();

foreach ($customers as $customer) {
    // kwerenda SQL nie jest wykonywana
    $orders = $customer->orders;
}
```

Wywołanie metody `with()` powoduje pobranie zamówień dla pierwszych 100 klientów w pojedynczej kwerendzie SQL, dzięki czemu redukujemy ilość zapytań ze 101 do 2!

Możliwe jest gorliwe pobranie jednej lub wielu relacji, a nawet gorliwe pobranie *zagnieżdżonych relacji*. Zagnieżdżona relacja to taka, która została zadeklarowana w relacyjnej klasie Active Record. Dla przykładu, `Customer` jest powiązany z `Order` poprzez relację `orders`, a `Order` jest powiązany z `Item` poprzez relację `items`. Ładując dane dla `Customer`, możesz gorliwie pobrać `items` używając notacji zagnieżdżonej relacji `orders.items`.

Poniższy kod pokazuje różne sposoby użycia `with()`. Zakładamy, że klasa `Customer` posiada dwie relacje `orders` i `country`, a klasa `Order` jedną relację `items`.

```
// gorliwe pobieranie "orders" i "country"
$customers = Customer::find()->with('orders', 'country')->all();
// odpowiednik powyższego w zapisie tablicowym
$customers = Customer::find()->with(['orders', 'country'])->all();
// kwerenda SQL nie jest wykonywana
$orders= $customers[0]->orders;
// kwerenda SQL nie jest wykonywana
$country = $customers[0]->country;

// gorliwe pobieranie "orders" i zagnieżdżonej relacji "orders.items"
$customers = Customer::find()->with('orders.items')->all();
```

```
// uzyskanie dostępu do produktów pierwszego zamówienia pierwszego klienta
// kwerenda SQL nie jest wykonywana
$items = $customers[0]->orders[0]->items;
```

Możesz pobrać gorliwie także głęboko zagnieżdżone relacje, jak np. `a.b.c.d`. Każda z kolejnych następujących po sobie relacji zostanie pobrana gorliwie - wywołując `with()` z `a.b.c.d`, pobierzesz `a`, `a.b`, `a.b.c` i `a.b.c.d`.

> **Informacja:** Podsumowując, podczas gorliwego pobierania `N` relacji, pośród których `M` relacji jest zdefiniowanych za pomocą tabeli węzła, zostanie wykonanych łącznie `N+M+1` kwerend SQL. Zwróć uwagę na to, że zagnieżdżona relacja `a.b.c.d` jest liczona jako 4 relacje.

Podczas gorliwego pobierania relacji, możesz dostosować kwerendę do własnych potrzeb korzystając z funkcji anonimowej. Przykład:

```
// znajdź klientów i pobierz ich kraje zamieszkania i aktywne zamówienia
// 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();
```

Dostosowując relacyjną kwerendę należy podać nazwę relacji jako klucz tablicy i użyć funkcji anonimowej jako odpowiadającej kluczowi wartości. Funkcja anonimowa otrzymuje parametr `$query`, reprezentujący obiekt `ActiveQuery`, służący do wykonania relacyjnej kwerendy. W powyższym przykładzie modyfikujemy relacyjną kwerendę dodając warunek ze statusem zamówienia.

> **Uwaga:** Wywołując `select()` podczas gorliwego pobierania relacji, należy upewnić się, że kolumny określone w deklaracji relacji znajdują się na liście pobieranych. W przeciwnym razie powiązany model może nie zostać poprawnie załadowany. Przykład:
>
> ```
> $orders = Order::find()->select(['id',
> 'amount'])->with('customer')->all();
> // $orders[0]->customer ma zawsze wartość `null`. Aby rozwiązać ten
> problem, należy użyć:
> $orders = Order::find()->select(['id', 'amount',
> 'customer_id'])->with('customer')->all();
> ```

Przyłączanie relacji

> **Uwaga:** Zawartość tej sekcji odnosi się tylko do relacyjnych baz danych, takich jak MySQL, PostgreSQL, itp.

Relacyjne kwerendy opisane do tej pory jedynie nawiązują do głównych kolumn tabeli podczas pobierania danych. W rzeczywistości często musimy odnieść się do kolumn w powiązanych tabelach. Przykładowo chcemy pobrać klientów, którzy złożyli przynajmniej jedno aktywne zamówienie - możemy tego dokonać za pomocą następującej przyłączającej kwerendy:

```
// 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();
```

> **Uwaga:** Podczas tworzenia relacyjnych kwerend zawierających instrukcję SQL JOIN koniecznym jest ujednoznacznienie nazw kolumn. Standardową praktyką w takim wypadku jest poprzedzenie nazwy kolumny odpowiadającą jej nazwą tabeli.

Jeszcze lepszym rozwiązaniem jest użycie istniejącej deklaracji relacji wywołując metodę `joinWith()`:

```
$customers = Customer::find()
    ->joinWith('orders')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->all();
```

Oba rozwiązania wykonują te same zestawy instrukcji SQL, ale ostatnie jest o wiele schludniejsze.

`joinWith()` domyślnie korzysta z `LEFT JOIN` do przyłączenia głównej tabeli z relacyjną. Możesz określić inny typ przyłączenia (np. `RIGHT JOIN`) podając trzeci parametr `$joinType`. Jeśli chcesz użyć typu przyłączenia `INNER JOIN`, możesz bezpośrednio wywołać metodę `innerJoinWith()`.

Wywołanie `joinWith()` domyślnie pobierze gorliwie dane relacyjne. Jeśli nie chcesz pobierać danych w ten sposób, możesz ustawić drugi parametr `$eagerLoading` na `false`.

Tak jak w przypadku `with()`, możesz przyłączyć jedną lub wiele relacji na raz, dodać do nich dodatkowe warunki, przyłączyć zagnieżdżone relacje i korzystać z zarówno `with()` jak i `joinWith()`. Przykładowo:

```
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->andWhere(['>', 'subtotal', 100]);
    },
])->with('country')
    ->all();
```

Czasem, przyłączając dwie tabele, musisz sprecyzować dodatkowe warunki dla części `ON` kwerendy JOIN. Można to zrobić wywołując metodę `onCondition()` w poniższy sposób:

```
// 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();
```

Powyższa kwerenda pobiera *wszystkich* klientów i dla każdego z nich pobiera wszystkie aktywne zamówienia. Zwróć uwagę na to, że ten przykład różni się od poprzedniego, gdzie pobierani byli tylko klienci posiadający przynajmniej jedno aktywne zamówienie.

> **Informacja:** Jeśli `ActiveQuery` zawiera warunek podany za pomocą `onCondition()`, będzie on umieszczony w części instrukcji `ON` tylko jeśli kwerenda zawiera JOIN. W przeciwnym wypadku warunek ten będzie automatycznie dodany do części `WHERE`. Może zatem składać się z warunków opierających się tylko na kolumnach powiązanej tabeli.

Aliasy dołączanych tabeli Jak już wspomniano wcześniej, używając JOIN w kwerendzie, musimy ujednoznacznić nazwy kolumn. Z tego powodu często stosuje się aliasy dla tabel. Alias dla kwerendy relacyjnej można ustawić, modyfikując ją w następujący sposób:

```
$query->joinWith([
    'orders' => function ($q) {
        $q->from(['o' => Order::tableName()]);
    },
])
```

Powyższy sposób wygląda jednak na bardzo skomplikowany i wymaga ręcznej modyfikacji w kodzie nazwy tabeli obiektu relacji lub wywołania `Order::tableName()`. Od wersji 2.0.7, Yii udostępnia do tego celu skróconą metodę. Możliwe jest zdefiniowanie i używanie aliasu dla tabeli relacji w poniższy sposób:

```
// do␣łącz relację 'orders' i posortuj wyniki po 'orders.id'
$query->joinWith(['orders o'])->orderBy('o.id');
```

Powyższy kod działa dla prostych relacji. Jeśli jednak potrzebujesz aliasu dla tabeli dołączonej w zagnieżdżonej relacji, np. `$query->joinWith(['orders.product'])`, musisz rozwinąć wywołanie `joinWith` jak w poniższym przykładzie:

```
$query->joinWith(['orders o' => function($q) {
        $q->joinWith('product p');
    }])
    ->where('o.amount > 100');
```

Odwrócone relacje

Deklaracje relacji są zazwyczaj obustronne dla dwóch klas Active Record. Przykładowo `Customer` jest powiązany z `Order` poprzez relację `orders`, a `Order` jest powiązany jednocześnie z `Customer` za pomocą relacji `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']);
    }
}
```

Rozważmy teraz poniższy kod:

```
// 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;

// zwraca "różne"
echo $customer2 === $customer ? 'takie same' : 'różne';
```

Wydawałoby się, że `$customer` i `$customer2` powinny być identyczne, ale jednak nie są! W rzeczywistości zawierają takie same dane klienta, ale są różnymi

obiektami. Wywołując `$order->customer` wykonywana jest dodatkowa kwerenda SQL do wypełnienia nowego obiektu `$customer2`.

Aby uniknąć nadmiarowego wykonywania ostatniej kwerendy SQL w powyższym przykładzie, powinniśmy wskazać Yii, że `customer` jest *odwróconą relacją* `orders` wywołując metodę `inverseOf()` jak pokazano to poniżej:

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

Z tą dodatkową instrukcją w deklaracji relacji uzyskamy:

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// kwerenda SQL nie jest wykonywana
$customer2 = $order->customer;

// wyświetla "takie same"
echo $customer2 === $customer ? 'takie same' : 'różne';
```

> **Uwaga:** Odwrócone relacje nie mogą być definiowane dla relacji zawierających tabelę węzła, dlatego też definiując relację z użyciem `via()` lub `viaTable()` nie powinno się już wywoływać `inverseOf()`.

6.1.11 Zapisywanie relacji

Podczas pracy z danymi relacyjnymi często konieczne jest ustalenie związku pomiędzy różnymi danymi lub też usunięcie istniejącego połączenia. Takie akcje wymagają ustalenia właściwych wartości dla kolumn definiujących relacje. Korzystając z Active Record można użyć następujących instrukcji:

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

// ustawianie wartości dla atrybutu definiującego relację "customer" dla
Order
$order->customer_id = $customer->id;
$order->save();
```

Active Record zawiera metodę `link()`, która pozwala na uzyskanie powyższego w efektywniejszy sposób:

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

$order->link('customer', $customer);
```

Metoda `link()` wymaga podania konkretnej nazwy relacji i docelowej instancji Active Record, z którą powinna być nawiązany związek. Mechanizm ten zmodyfikuje wartości atrybutów łączących obie instancje Active Record i zapisze je w bazie danych. W powyższym przykładzie atrybut `customer_id` instancji `Order` otrzyma wartość atrybutu `id` instancji `Customer`, a następnie zostanie zapisany w bazie danych.

> **Uwaga:** Nie możesz łączyć w ten sposób dwóch świeżo utworzonych instancji Active Record.

Zaleta używania `link()` jest jeszcze bardziej widoczna, jeśli relacja jest zdefiniowana poprzez tabelę węzła. Przykładowo możesz użyć następującego kodu, aby połączyć instancję `Order` z instancją `Item`:

```
$order->link('items', $item);
```

Powyższy przykład automatycznie doda nowy wiersz w tabeli węzła `order_item`, aby połączyć zamówienie z produktem.

> **Informacja:** Metoda `link()` NIE wykona automatycznie żadnego procesu walidacji danych podczas zapisywania instancji Active Record. Na Tobie spoczywa obowiązek walidacji wszystkich danych przed wywołaniem tej metody.

Odwrotną operacją do `link()` jest `unlink()`, która usuwa istniejący związek pomiędzy dwoma instancjami Active Record. Przykładowo:

```
$customer = Customer::find()->with('orders')->where(['id' => 123])->one();
$customer->unlink('orders', $customer->orders[0]);
```

Domyślnie metoda `unlink()` ustawia wartość klucza obcego (lub wielu kluczy obcych), który definiuje istniejącą relację, na `null`. Można jednak zamiast tego wybrać opcję usuwania wiersza tabeli, który zawiera klucz obcy, ustawiając w metodzie parametr `$delete` na `true`.

Jeśli w relacji użyty jest węzeł, wywołanie `unlink()` spowoduje wyczyszczenie kluczy obcych w tabeli węzła lub też usunięcie odpowiadających im wierszy, jeśli `$delete` jest ustawione na `true`.

6.1.12 Relacje międzybazowe

Active Record pozwala na deklarowanie relacji pomiędzy klasami Active Record zasilanymi przez różne bazy danych. Bazy danych mogę być różnych typów (np. MySQL i PostgreSQL lub MS SQL i MongoDB) i mogą pracować na różnych serwerach. Do wykonania relacyjnych zapytań używa się takich samych procedur, jak w przypadku relacji w obrębie jednej bazy danych. Przykład:

```
// Customer jest powiązany z tabelą "customer" w relacyjnej bazie danych
(np. MySQL)
class Customer extends \yii\db\ActiveRecord
{
    public static function tableName()
    {
        return 'customer';
    }

    public function getComments()
    {
        // klient posiada wiele komentarzy
        return $this->hasMany(Comment::class, ['customer_id' => 'id']);
    }
}

// Comment jest powiązany z kolekcją "comment" w bazie danych MongoDB
class Comment extends \yii\mongodb\ActiveRecord
{
    public static function collectionName()
    {
        return 'comment';
    }

    public function getCustomer()
    {
        // komentarz jest przypisany do jednego klienta
        return $this->hasOne(Customer::class, ['id' => 'customer_id']);
    }
}

$customers = Customer::find()->with('comments')->all();
```

Możesz używać większości funkcjonalności dostępnych dla relacyjnych kwerend opisanych w tym rozdziale.

> **Uwaga:** Użycie `joinWith()` jest ograniczone do baz danych pozwalających na międzybazowe kwerendy JOIN, dlatego też nie możesz użyć tej metody w powyższym przykładzie, ponieważ MongoDB nie wspiera instrukcji JOIN.

6.1.13 Niestandardowe klasy kwerend

Domyślnie wszystkie kwerendy Active Record używają klasy `ActiveQuery`. Aby użyć niestandardowej klasy kwerend razem z klasą Active Record, należy nadpisać metodę `find()`, aby zwracała instancję żądanej klasy kwerend. Przykład:

```
// plik Comment.php
namespace app\models;

use yii\db\ActiveRecord;

class Comment extends ActiveRecord
{
    public static function find()
    {
        return new CommentQuery(get_called_class());
    }
}
```

Od tego momentu, za każdym razem, gdy wykonywana będzie kwerenda (np. `find()`, `findOne()`) lub pobierana relacja (np. `hasOne()`) klasy `Comment`, praca będzie odbywać się na instancji `CommentQuery` zamiast `ActiveQuery`.

Teraz należy zdefiniować klasę `CommentQuery`, którą można dopasować do własnych kreatywnych potrzeb, dzięki czemu budowanie zapytań bazodanowych będzie o wiele bardziej ułatwione. Dla przykładu,

```
// plik CommentQuery.php
namespace app\models;

use yii\db\ActiveQuery;

class CommentQuery extends ActiveQuery
{
    // dodatkowe warunki relacyjnej kwerendy do␣łączone jako domyślne (ten
    krok można pominąć)
    public function init()
    {
        $this->andOnCondition(['deleted' => false]);
        parent::init();
    }

    // ... dodaj zmodyfikowane metody kwerend w tym miejscu ...

    public function active($state = true)
    {
        return $this->andOnCondition(['active' => $state]);
    }
}
```

> **Uwaga:** Zwykle, zamiast wywoływać metodę `onCondition()`, powinno się używać metody `andOnCondition()` lub `orOnCondition()`,

aby dołączać kolejne warunki zapytania w konstruktorze kwerend, dzięki czemu istniejące warunki nie zostaną nadpisane.

Powyższy przykład pozwala na użycie następującego kodu:

```
$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();
```

Wskazówka: Dla dużych projektów rekomendowane jest, aby używać własnych, odpowiednio dopasowanych do potrzeb, klas kwerend, dzięki czemu klasy Active Record pozostają przejrzyste.

Możesz także użyć nowych metod budowania kwerend przy definiowaniu relacji z `Comment` lub wykonywaniu relacyjnych kwerend:

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

$customers = Customer::find()->joinWith('activeComments')->all();

// lub alternatywnie
class Customer extends \yii\db\ActiveRecord
{
    public function getComments()
    {
        return $this->hasMany(Comment::class, ['customer_id' => 'id']);
    }
}

$customers = Customer::find()->joinWith([
    'comments' => function($q) {
        $q->active();
    }
])->all();
```

Informacja: W Yii 1.1 do tego celu służy mechanizm *podzbiorów (scope)*, nie jest on jednak bezpośrednio wspierany w Yii 2.0, a zamiast tego powinno się używać dopasowanych do własnych potrzeb klas kwerend.

6.1.14 Pobieranie dodatkowych pól

W momencie, gdy instancja Active Record pobiera dane z wyniku kwerendy, wartości kolumn przypisywane są do odpowiadających im atrybutów.

Możliwe jest pobranie dodatkowych kolumn lub wartości za pomocą kwerendy i przypisanie ich w Active Record. Przykładowo załóżmy, że mamy tabelę `room`, która zawiera informacje o pokojach dostępnych w hotelu. Każdy pokój przechowuje informacje na temat swojej wielkości za pomocą pól `length`, `width` i `height`. Teraz wyobraźmy sobie, że potrzebujemy pobrać listę wszystkich pokojów posortowaną po ich kubaturze w malejącej kolejności. Nie możemy obliczyć kubatury korzystając z PHP, ponieważ zależy nam na szybkim posortowaniu rekordów i dodatkowo chcemy wyświetlić pole `volume` na liście. Aby osiągnąć ten cel, musimy zadeklarować dodatkowe pole w klasie `Room` rozszerzającej Active Record, które przechowa wartość `volume`:

```
class Room extends \yii\db\ActiveRecord
{
    public $volume;

    // ...
}
```

Następnie należy skonstruować kwerendę, która obliczy kubaturę i wykona sortowanie:

```
$rooms = Room::find()
    ->select([
        '{{room}}.*', // pobierz wszystkie kolumny
        '([[length]] * [[width]] * [[height]]) AS volume', // oblicz
        kubaturę
    ])
    ->orderBy('volume DESC') // posortuj
    ->all();

foreach ($rooms as $room) {
    echo $room->volume; // zawiera wartość obliczoną przez SQL
}
```

Możliwość pobrania dodatkowych pól jest szczególnie pomocna przy kwerendach agregujących. Załóżmy, że potrzebujesz wyświetlić listę klientów wraz z liczbą zamówień, których dokonali. Najpierw musisz zadeklarować klasę `Customer` wraz z relacją `orders` i dodatkowym polem przechowującym liczbę zamówień:

```
class Customer extends \yii\db\ActiveRecord
{
    public $ordersCount;

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}
```

Teraz już możesz skonstruować kwerendę, która przyłączy zamówienia i policzy ich liczbę:

```
$customers = Customer::find()
    ->select([
        '{{customer}}.*', // pobierz wszystkie kolumny klienta
        'COUNT({{order}}.id) AS ordersCount' // oblicz ilość zamówień
    ])
    ->joinWith('orders') // przy␣łącz tabelę węz␣la
    ->groupBy('{{customer}}.id') // pogrupuj wyniki dla funkcji
    agregacyjnej
    ->all();
```

Wadą tej metody jest to, że jeśli informacja nie może zostać pobrana za pomocą kwerendy SQL, musi ona być obliczona oddzielnie. Zatem po pobraniu konkretnego wiersza tabeli za pomocą regularnej kwerendy bez dodatkowej instrukcji select, niemożliwym będzie zwrócenie wartości dla dodatkowych pól. Tak samo stanie się w przypadku świeżo zapisanych rekordów.

```
$room = new Room();
$room->length = 100;
$room->width = 50;
$room->height = 2;

$room->volume; // ta wartość będzie wynosić `null` ponieważ nie zosta␣la
jeszcze zadeklarowana
```

Używając magicznych metod `__get()` i `__set()`, możemy emulować zachowania właściwości:

```
class Room extends \yii\db\ActiveRecord
{
    private $_volume;

    public function setVolume($volume)
    {
        $this->_volume = (float) $volume;
    }

    public function getVolume()
    {
        if (empty($this->length) || empty($this->width) ||
        empty($this->height)) {
            return null;
        }

        if ($this->_volume === null) {
            $this->setVolume(
                $this->length * $this->width * $this->height
            );
        }
```

```
        return $this->_volume;
    }

    // ...
}
```

Kiedy kwerenda nie zapewni wartości kubatury, model będzie w stanie automatycznie ją obliczyć, używając swoich atrybutów.

Możesz obliczyć sumaryczne pola rónież korzystając ze zdefiniowanych relacji:

```
class Customer extends \yii\db\ActiveRecord
{
    private $_ordersCount;

    public function setOrdersCount($count)
    {
        $this->_ordersCount = (int) $count;
    }

    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // dzięki temu unikamy wywo␣lania kwerendy
            szukającej g␣lównych kluczy o wartości null
        }

        if ($this->_ordersCount === null) {
            $this->setOrdersCount($this->getOrders()->count()); // oblicz
            sumę na żądanie z relacji
        }

        return $this->_ordersCount;
    }

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }
}
```

Dla powyższego kodu, kiedy 'ordersCount' występuje w instrukcji 'select' - `Customer::ordersCount` zostanie wypełnione rezultatem kwerendy, w pozostałych przypadkach zostanie obliczone na żądanie używając relacji `Customer::orders`.

Takie podejście może być równie dobrze użyte do stworzenia skrótów dla niektórych danych relacji, zwłąszcza tych służących do obliczania sumarycznego. Przykładowo:

```
class Customer extends \yii\db\ActiveRecord
{
```

```
    /**
     * Deklaracja wirtualnej w␣łaściwości tylko do odczytu dla danych
sumarycznych.
     */
    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // to pozwala na uniknięcie uruchamiania
            wyszukującej kwerendy dla pustych kluczy g␣łównych
        }

        return empty($this->ordersAggregation) ? 0 :
        $this->ordersAggregation[0]['counted'];
    }

    /**
     * Deklaracja zwyk␣łej relacji 'orders'.
     */
    public function getOrders()
    {
        return $this->hasMany(Order::class, ['customer_id' => 'id']);
    }

    /**
     * Deklaracja nowej relacji bazującej na 'orders', ale zapewniającej
pobranie danych sumarycznych.
     */
    public function getOrdersAggregation()
    {
        return $this->getOrders()
            ->select(['customer_id', 'counted' => 'count(*)'])
            ->groupBy('customer_id')
            ->asArray(true);
    }

    // ...
}

foreach (Customer::find()->with('ordersAggregation')->all() as $customer) {
    echo $customer->ordersCount; // wyświetla dane sumaryczne z relacji bez
    dodatkowej kwerendy dzięki gorliwemu pobieraniu
}

$customer = Customer::findOne($pk);
$customer->ordersCount; // wyświetla dane sumaryczne z relacji pobranej
leniwie
```

6.2 Migracje bazy danych

W czasie rozwoju i utrzymywania aplikacji zasilanej danymi z bazy danych, struktura tej ostatniej ewoluuje podobnie jak sam kod źródłowy. Przykładowo, rozbudowując aplikację konieczne jest dodanie nowej tabeli, lub też już po

wydaniu aplikacji na serwerze produkcyjnym przydałby się indeks, aby poprawić wydajność zapytania itd. Zmiana struktury bazy danych często pociąga za sobą zmiany w kodzie źródłowym, dlatego też Yii udostępnia funkcjonalność tak zwanych *migracji bazodanowych*, która pozwala na kontrolowanie zmian w bazie danych (*migracji*).

Poniższe kroki pokazują, jak migracje mogą być wykorzystane przez zespół deweloperski w czasie pracy:

1. Tomek tworzy nową migrację (np. dodaje nową tabelę, zmienia definicję kolumny, itp.).
2. Tomek rejestruje (ćommit") nową migrację w systemie kontroli wersji (np. Git, Mercurial).
3. Mariusz uaktualnia swoje repozytorium z systemu kontroli wersji i otrzymuje nową migrację.
4. Mariusz dodaje migrację do swojej lokalnej bazy danych, dzięki czemu synchronizuje ją ze zmianami, które wprowadził Tomek.

A poniższe kroki opisują w skrócie jak stworzyć nowe wydanie z migracją bazy danych na produkcji:

1. Rafał tworzy tag wydania dla repozytorium projektu, który zawiera nowe migracje bazy danych.
2. Rafał uaktualnia kod źródłowy na serwerze produkcyjnym do otagowanej wersji.
3. Rafał dodaje zebrane nowe migracje do produkcyjnej bazy danych.

Yii udostępnia zestaw narzędzi konsolowych, które pozwalają na:
- utworzenie nowych migracji;
- dodanie migracji;
- cofnięcie migracji;
- ponowne zaaplikowanie migracji;
- wyświetlenie historii migracji i jej statusu.

Powyższe narzędzia są dostępne poprzez komendę `yii migrate`. W tej sekcji opiszemy szczegółowo w jaki sposób z nich korzystać. Możesz również zapoznać się ze sposobem użycia narzędzi w konsoli za pomocą komendy pomocy `yii help migrate`.

> **Wskazówka:** Migracje mogą modyfikować nie tylko schemat bazy danych, ale również same dane, a także mogą służyć do innych zadań jak tworzenie hierarchi kontroli dostępu dla ról (RBAC) lub czyszczenie pamięci podręcznej.

> **Uwaga:** Modyfikowanie danych w migracji zwykle jest znacznie prostsze, jeśli użyje się do tego klas Active Record, dzięki logice już tam zaimplementowanej. Należy jednak pamiętać, że logika aplikacji jest podatna na częste zmiany, a naturalnym stanem kodu migracji jest jego stałość - w przypadku zmian w warstwie Active Record aplikacji ryzykujemy zepsucie migracji, które z niej korzystają. Z tego powodu kod migracji powinien być utrzymywany niezależnie od pozostałej logiki aplikacji.

6.2.1 Tworzenie migracji

Aby utworzyć nową migrację, uruchom poniższą komendę:

```
yii migrate/create <nazwa>
```

Wymagany argument `nazwa` przekazuje zwięzły opis migracji. Przykładowo, jeśli migracja ma dotyczyć utworzenia nowej tabeli o nazwie *news*, możesz użyć jako argumentu `create_news_table` i uruchomić komendę:

```
yii migrate/create create_news_table
```

> **Uwaga:** Argument `nazwa` zostanie użyty jako część nazwy klasy nowej migracji i z tego powodu powinien składać się tylko z łacińskich liter, cyfr i/lub znaków podkreślenia.

Powyższa komenda utworzy nowy plik klasy PHP o nazwie podobnej do `m150101_185401_create_news_table.php` w folderze `@app/migrations`. Plik będzie zawierał poniższy kod, gdzie zadeklarowany jest szkielet klasy `m150101_185401_create_news_table`:

```php
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {

    }

    public function down()
    {
        echo "m101129_185401_create_news_table cannot be reverted.\n";

        return false;
    }

    /*
    // Use safeUp/safeDown to run migration code within a transaction
```

```
    public function safeUp()
    {
    }

    public function safeDown()
    {
    }
    */
}
```

Każda migracja zdefiniowana jest jako klasa PHP rozszerzająca `yii\db\Migration`. Nazwa klasy migracji jest generowana automatycznie w formacie `m<YYMMDD_HHMMSS>_<Nazwa>`, gdzie

- `<YYMMDD_HHMMSS>` to data i czas UTC wskazujące na moment utworzenia migracji,
- `<Nazwa>` jest identyczna z wartością argumentu `nazwa` podanego dla komendy.

Wewnątrz klasy migracji należy napisać kod w metodzie `up()`, która wprowadzi zmiany w strukturze bazy danych. Można również napisać kod w metodzie `down()`, który spowoduje cofnięcie zmian wprowadzonych w `up()`. Metoda `up()` jest uruchamiana w momencie aktualizacji bazy, a `down()` w momencie przywracania jej do poprzedniego stanu. Poniższy kod pokazuje, jak można zaimplementować klasę migracji, aby utworzyć tabelę `news`:

```php
<?php

use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => Schema::TYPE_PK,
            'title' => Schema::TYPE_STRING . ' NOT NULL',
            'content' => Schema::TYPE_TEXT,
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

> **Informacja:** Nie wszystkie migracje są odwracalne. Dla przykładu, jeśli w `up()` usuwane są wiersze z tabeli, możesz nie być w stanie przywrócić ich w metodzie `down()`. Może też zdarzyć się, że celowo nie podasz nic w `down()` - cofanie zmian migracji bazy danych

> nie jest czymś powszechnym - w takim wypadku należy zwrócić `false` w metodzie `down()`, aby wyraźnie wskazać, że migracja nie jest odwracalna.

Podstawowa klasa migracji `yii\db\Migration` umożliwia połączenie z bazą danych poprzez właściwość `db`. Możesz użyć jej do modyfikowania schematu bazy za pomocą metod opisanych w sekcji Praca ze schematem bazy danych.

Przy tworzeniu tabeli albo kolumny zamiast używać rzeczywistych typów, powinno się stosować *typy abstrakcyjne*, dzięki czemu migracje będą niezależne od pojedynczych silników bazodanowych. Klasa `yii\db\Schema` definiuje zestaw stałych, które reprezentują wspierane typy abstrakcyjne. Stałe te nazwane są według schematu `TYPE_<Nazwa>`. Dla przykładu, `TYPE_PK` odnosi się do typu klucza głównego z autoinkrementacją; `TYPE_STRING` do typu łańcucha znaków. Kiedy migracja jest dodawana do konkretnej bazy danych, typy abstrakcyjne są tłumaczone na odpowiadające im typy rzeczywiste. W przypadku MySQL, `TYPE_PK` jest zamieniony w `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY`, a `TYPE_STRING` staje się `varchar(255)`.

Możesz łączyć abstrakcyjne typy z dodatkowymi definicjami - w powyższym przykładzie `NOT NULL` jest dodane do `Schema::TYPE_STRING`, aby oznaczyć, że kolumna nie może być ustawiona jako `null`.

> **Informacja:** Mapowanie typów abstrakcyjnych na rzeczywiste jest określone we właściwości `$typeMap` dla każdej klasy `QueryBuilder` poszczególnych wspieranych silników baz danych.

Począwszy od wersji 2.0.6, możesz skorzystać z nowej klasy budowania schematów, która pozwala na znacznie wygodniejszy sposób definiowana kolumn. Dzięki temu migracja z przykładu powyżej może być napisana następująco:

```php
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

Lista wszystkich metod do definiowania typów kolumn dostępna jest w dokumentacji API dla `yii\db\SchemaBuilderTrait`.

6.2.2 Generowanie migracji

Począwszy od wersji 2.0.7 konsola migracji pozwala na wygodne utworzenie nowej migracji.

Jeśli nazwa migracji podana jest w jednej z rozpoznawalnych form, np. `create_xxx_table` lub `drop_xxx_table`, wtedy wygenerowany plik migracji będzie zawierał dodatkowy kod, w tym przypadku odpowiednio kod tworzenia i usuwania tabeli. Poniżej opisane są wszystkie warianty tej funkcjonalności.

Tworzenie tabeli

```
yii migrate/create create_post_table
```

generuje

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey()
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

Aby jednocześnie od razu dodać kolumny tabeli, zdefiniuj je za pomocą opcji `--fields`.

```
yii migrate/create create_post_table --fields="title:string,body:text"
```

generuje

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

Możesz określić też więcej parametrów kolumny.

```
yii migrate/create create_post_table
--fields="title:string(12):notNull:unique,body:text"
```

generuje

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
```

```
    {
        $this->dropTable('post');
    }
}
```

> **Uwaga:** Klucz główny jest dodawany automatycznie i nazwany domyślnie `id`. Jeśli chcesz użyć innej nazwy, możesz zdefiniować go bezpośrednio np. `--fields="name:primaryKey"`.

Klucze obce Począwszy od wersji 2.0.8 generator pozwala na zdefiniowanie kluczy obcych za pomocą opcji `foreignKey`.

```
yii migrate/create create_post_table
--fields="author_id:integer:notNull:foreignKey(user),category_id:integer:defaultValue(1):foreignK
```

generuje

```
/**
 * Handles the creation for table `post`.
 * Has foreign keys to the tables:
 *
 * - `user`
 * - `category`
 */
class m160328_040430_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'author_id' => $this->integer()->notNull(),
            'category_id' => $this->integer()->defaultValue(1),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);

        // creates index for column `author_id`
        $this->createIndex(
            'idx-post-author_id',
            'post',
            'author_id'
        );

        // add foreign key for table `user`
        $this->addForeignKey(
            'fk-post-author_id',
            'post',
            'author_id',
            'user',
```

```
            'id',
            'CASCADE'
        );

        // creates index for column `category_id`
        $this->createIndex(
            'idx-post-category_id',
            'post',
            'category_id'
        );

        // add foreign key for table `category`
        $this->addForeignKey(
            'fk-post-category_id',
            'post',
            'category_id',
            'category',
            'id',
            'CASCADE'
        );
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        // drops foreign key for table `user`
        $this->dropForeignKey(
            'fk-post-author_id',
            'post'
        );

        // drops index for column `author_id`
        $this->dropIndex(
            'idx-post-author_id',
            'post'
        );

        // drops foreign key for table `category`
        $this->dropForeignKey(
            'fk-post-category_id',
            'post'
        );

        // drops index for column `category_id`
        $this->dropIndex(
            'idx-post-category_id',
            'post'
        );

        $this->dropTable('post');
    }
}
```

Umiejscowienie słowa `foreignKey` w definicji kolumny nie ma znaczenia dla generatora, zatem:

- `author_id:integer:notNull:foreignKey(user)`
- `author_id:integer:foreignKey(user):notNull`
- `author_id:foreignKey(user):integer:notNull`

wygenerują ten sam kod.

Opcja `foreignKey` może być wzbogacona o parametr w nawiasach, który oznacza nazwę tabeli relacji dla generowanego klucza obcego. Bez tego parametru użyta zostanie nazwa tabeli relacji zgodna z nazwą kolumny.

W przykładzie powyżej `author_id:integer:notNull:foreignKey(user)` wygeneruje kolumnę o nazwie `author_id` z kluczem obcym wskazującym na tabelę `user`, natomiast `category_id:integer:defaultValue(1):foreignKey` wygeneruje kolumnę `category_id` z kluczem obcym wskazującym na tabelę `category`.

Począwszy od wersji 2.0.11, dla `foreignKey` można podać drugi parametr, oddzielony białym znakiem, z nazwą kolumny relacji dla generowanego klucza obcego. Jeśli drugi parametr nie jest podany, nazwa kolumny jest pobierana ze schematu tabeli. Jeśli schemat nie istnieje , klucz główny nie jest ustawiony lub jest kluczem kompozytowym, używana jest domyślna nazwa `id`.

Usuwanie tabeli

```
yii migrate/create drop_post_table
--fields="title:string(12):notNull:unique,body:text"
```

generuje

```
class m150811_220037_drop_post_table extends Migration
{
    public function up()
    {
        $this->dropTable('post');
    }

    public function down()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }
}
```

Dodawanie kolumny

Jeśli nazwa migracji jest w postaci `add_xxx_column_to_yyy_table`, wtedy plik będzie zawierał wywołania metod `addColumn` i `dropColumn`.

Aby dodać kolumnę:

```
yii migrate/create add_position_column_to_post_table
--fields="position:integer"
```

co generuje

```
class m150811_220037_add_position_column_to_post_table extends Migration
{
    public function up()
    {
        $this->addColumn('post', 'position', $this->integer());
    }

    public function down()
    {
        $this->dropColumn('post', 'position');
    }
}
```

Możesz dodać wiele kolumn jednocześnie:

```
yii migrate/create add_xxx_column_yyy_column_to_zzz_table
--fields="xxx:integer,yyy:text"
```

Usuwanie kolumny

Jeśli nazwa migracji jest w postaci `drop_xxx_column_from_yyy_table`, wtedy plik będzie zawierał wywołania metod `dropColumn` i `addColumn`.

```
yii migrate/create drop_position_column_from_post_table
--fields="position:integer"
```

generuje

```
class m150811_220037_drop_position_column_from_post_table extends Migration
{
    public function up()
    {
        $this->dropColumn('post', 'position');
    }

    public function down()
    {
        $this->addColumn('post', 'position', $this->integer());
    }
}
```

Dodawanie tabeli węzła

Jeśli nazwa migracji jest w postaci `create_junction_table_for_xxx_and_yyy_tables` lub `create_junction_xxx_and_yyy_tables`, wtedy plik będzie zawierał kod potrzebny do wygenerowania tabeli węzła pomiędzy tabelami `xxx` i `yyy`.

```
yii migrate/create create_junction_table_for_post_and_tag_tables
--fields="created_at:dateTime"
```

generuje

```
/**
 * Handles the creation for table `post_tag`.
 * Has foreign keys to the tables:
 *
 * - `post`
 * - `tag`
 */
class m160328_041642_create_junction_table_for_post_and_tag_tables extends
Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post_tag', [
            'post_id' => $this->integer(),
            'tag_id' => $this->integer(),
            'created_at' => $this->dateTime(),
            'PRIMARY KEY(post_id, tag_id)',
        ]);

        // creates index for column `post_id`
        $this->createIndex(
            'idx-post_tag-post_id',
            'post_tag',
            'post_id'
        );

        // add foreign key for table `post`
        $this->addForeignKey(
            'fk-post_tag-post_id',
            'post_tag',
            'post_id',
            'post',
            'id',
            'CASCADE'
        );

        // creates index for column `tag_id`
        $this->createIndex(
            'idx-post_tag-tag_id',
            'post_tag',
            'tag_id'
        );

        // add foreign key for table `tag`
        $this->addForeignKey(
            'fk-post_tag-tag_id',
```

```
            'post_tag',
            'tag_id',
            'tag',
            'id',
            'CASCADE'
        );
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        // drops foreign key for table `post`
        $this->dropForeignKey(
            'fk-post_tag-post_id',
            'post_tag'
        );

        // drops index for column `post_id`
        $this->dropIndex(
            'idx-post_tag-post_id',
            'post_tag'
        );

        // drops foreign key for table `tag`
        $this->dropForeignKey(
            'fk-post_tag-tag_id',
            'post_tag'
        );

        // drops index for column `tag_id`
        $this->dropIndex(
            'idx-post_tag-tag_id',
            'post_tag'
        );

        $this->dropTable('post_tag');
    }
}
```

Począwszy od wersji 2.0.11, nazwy kolumn kluczy obcych dla tabeli węzła są pobierane ze schematu tabel. Jeśli tabela nie jest zdefiniowana w schemacie, lub jej klucz główny nie jest ustawiony lub jest kluczem kompozytowym, używana jest domyślna nazwa `id`.

Migracje transakcyjne

Przy wykonywaniu skomplikowanych migracji bazodanowych, bardzo ważnym jest zapewnienie, aby wszystkie ich operacje zakończyły się sukcesem, a w przypadku niepowodzenia nie zostały wprowadzone tylko częściowo, dzięki

czemu baza danych może zachować spójność. Zalecane jest, aby w tym celu wykonywać operacje migracji wewnątrz transakcji.

Najprostszym sposobem implementacji migracji transakcyjnych jest umieszczenie ich kodu w metodach `safeUp()` i `safeDown()`. Metody te różnią się od `up()` i `down()` tym, że są wywoływane automatycznie wewnątrz transakcji. W rezultacie niepowodzenie wykonania dowolnej z operacji skutkuje automatycznym cofnięciem wszystkich poprzenich udanych operacji.

W poniższym przykładzie oprócz stworzenia tabeli `news` dodatkowo dodajemy pierwszy wiersz jej danych.

```php
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function safeUp()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);

        $this->insert('news', [
            'title' => 'test 1',
            'content' => 'content 1',
        ]);
    }

    public function safeDown()
    {
        $this->delete('news', ['id' => 1]);
        $this->dropTable('news');
    }
}
```

Zwróć uwagę na to, że dodając wiele operacji bazodanowych w `safeUp()`, zwykle powinieneś odwrócić kolejność ich wykonywania w `safeDown()`. W naszym przykładzie najpierw tworzymy tabelę, a potem dodajemy wiersz w `safeUp()`, natomiast w `safeDown()` najpierw kasujemy wiersz, a potem usuwamy tabelę.

> **Uwaga:** Nie wszystkie silniki baz danych wspierają transakcje i nie wszystkie rodzaje komend bazodanowych można umieszczać w transakcjach. Dla przykładu, zapoznaj się z rozdziałem dokumentacji MySQL Statements That Cause an Implicit Commit[8]. W przypadku braku możliwości skorzystania z transakcji, powinieneś użyć `up()` i `down()`.

[8] https://dev.mysql.com/doc/refman/5.7/en/implicit-commit.html

Metody pozwalające na dostęp do bazy danych

Bazowa klasa migracji `yii\db\Migration` udostępnia zestaw metod, dzięki którym można połączyć się z i manipulować bazą danych. Metody te są nazwane podobnie jak metody DAO klasy `yii\db\Command`. Przykładowo metoda `yii\db\Migration::createTable()` pozwala na stworzenie nowej tabeli, tak jak `yii\db\Command::createTable()`.

Zaletą korzystania z metod `yii\db\Migration` jest brak konieczności bezpośredniego tworzenia instancji `yii\db\Command`, a wywołanie każdej z tych metod dodatkowo wyświetli użyteczne informacje na temat operacji bazodanowych i czasu ich wykonywania.

Poniżej znajdziesz listę wspomnianych wcześniej metod:

- `execute()`: wykonywanie komendy SQL
- `insert()`: dodawanie pojedynczego wiersza
- `batchInsert()`: dodawanie wielu wierszy
- `upsert()`: dodawanie pojedynczego wiersza lub aktualizowanie go, jeśli już istnieje (od 2.0.14)
- `update()`: aktualizowanie wierszy
- `delete()`: usuwanie wierszy
- `createTable()`: tworzenie tabeli
- `renameTable()`: zmiana nazwy tabeli
- `dropTable()`: usuwanie tabeli
- `truncateTable()`: usuwanie wszystkich wierszy w tabeli
- `addColumn()`: dodawanie kolumny
- `renameColumn()`: zmiana nazwy kolumny
- `dropColumn()`: usuwanie kolumny
- `alterColumn()`: zmiana definicji kolumny
- `addPrimaryKey()`: dodawanie klucza głównego
- `dropPrimaryKey()`: usuwanie klucza głównego
- `addForeignKey()`: dodawanie klucza obcego
- `dropForeignKey()`: usuwanie klucza obcego
- `createIndex()`: tworzenie indeksu
- `dropIndex()`: usuwanie indeksu
- `addCommentOnColumn()`: dodawanie komentarza do kolumny
- `dropCommentFromColumn()`: usuwanie komentarza z kolumny
- `addCommentOnTable()`: dodawanie komentarza do tabeli
- `dropCommentFromTable()`: usuwanie komentarza z tabeli

> **Informacja:** `yii\db\Migration` nie udostępnia metod dla kwerendy danych. Wynika to z tego, że zwykle nie jest potrzebne wyświetlanie dodatkowych informacji na temat pobieranych danych z bazy. Dodatkowo możesz zawsze użyć potężnego Konstruktora kwerend do zbudowania i wywołania skomplikowanych kwerend. Użycie konstruktora kwerend w migracji może wyglądać następująco:

```
// uaktualnij kolumnę statusu dla wszystkich użytkowników
foreach((new Query)->from('users')->each() as $user) {
    $this->update('users', ['status' => 1], ['id' => $user['id']]);
}
```

6.2.3 Stosowanie migracji

Aby uaktualnić bazę danych do najświeższej wersji jej struktury, należy zastosować wszystkie dostępne nowe migracje, korzystając z poniższej komendy:

```
yii migrate
```

Komenda ta wyświetli listę wszystkich migracji, które jeszcze nie zostały zastosowane. Jeśli potwierdzisz, że chcesz je zastosować, wywoła ona metodę `up()` lub `safeUp()` dla każdej z migracji na liście, w kolejności ich znaczników czasu. Jeśli którakolwiek z migracji nie powiedzie się, komenda zakończy działanie bez stosowania pozostałych migracji.

> **Wskazówka:** Jeśli nie masz dostępu do linii komend na serwerze, wypróbuj rozszerzenie web shell[9].

Dla każdej udanej migracji komenda doda wiersz do bazy danych w tabeli `migration`, aby oznaczyć fakt zastosowania migracji. Pozwoli to na identyfikację, która z migracji została już zastosowana, a która jeszcze nie.

> **Informacja:** Narzędzie do migracji automatycznie utworzy tabelę `migration` w bazie danych, wskazaną przez opcję `db` komendy. Domyślnie jest to baza danych określona w komponencie aplikacji `db`.

Czasem możesz mieć potrzebę zastosowania tylko jednej bądź kilku nowych migracji, zamiast wszystkich na raz. Możesz tego dokonać określając liczbę migracji, które chcesz zastosować uruchamiając komendę. Przykładowo, poniższa komenda spróbuje zastosować następne trzy dostępne migracje:

```
yii migrate 3
```

Możesz również dokładnie wskazać konkretną migrację, która powinna być zastosowana na bazie danych, używając komendy `migrate/to` na jeden z poniższych sposobów:

```
yii migrate/to 150101_185401                      # używając znacznika czasu
z nazwy migracji
yii migrate/to "2015-01-01 18:54:01"              # używając ␣łańcucha
znaków, który może być sparsowany przez strtotime()
yii migrate/to m150101_185401_create_news_table   # używając pe␣lnej nazwy
yii migrate/to 1392853618                         # używając UNIXowego
znacznika czasu
```

[9] https://github.com/samdark/yii2-webshell

Jeśli dostępne są niezaaplikowane migracje wcześniejsze niż ta wyraźnie wskazane w komendzie, zostaną one zastosowane automatycznie przed wskazaną migracją.

Jeśli wskazana migracja została już wcześniej zaaplikowana, wszystkie zaaplikowane aplikacje z późniejszą datą zostaną cofnięte.

6.2.4 Cofanie migracji

Aby odwrócić (wycofać) jedną lub więcej migracji, które zostały wcześniej zastosowane, możesz uruchomić następującą komendę:

```
yii migrate/down     # cofa ostatnio dodaną migrację
yii migrate/down 3   # cofa 3 ostatnio dodane migracje
```

> **Uwaga:** Nie wszystkie migracje są odwracalne. Próba cofnięcia takiej migracji spowoduje błąd i zatrzyma cały proces.

6.2.5 Ponawianie migracji

Ponawianie migracji oznacza najpierw wycofanie jej, a potem ponowne zastosowanie. Można tego dokonać następująco:

```
yii migrate/redo        # ponawia ostatnio zastosowaną migrację
yii migrate/redo 3      # ponawia ostatnie 3 zastosowane migracje
```

> **Uwaga:** Jeśli migracja nie jest odwracalna, nie będziesz mógł jej ponowić.

6.2.6 Odświeżanie migracji

Począwszy od Yii 2.0.13 możliwe jest usunięcie wszystkich tabel i kluczy obcych z bazy danych i zastosowanie wszystkich migracji od początku.

```
yii migrate/fresh       # czyści bazę danych i wykonuje wszystkie migracje
od początku
```

6.2.7 Lista migracji

Aby wyświetlić listę wszystkich zastosowanych i oczekujących migracji, możesz użyć następujących komend:

```
yii migrate/history     # pokazuje ostatnie 10 zastosowanych migracji
yii migrate/history 5   # pokazuje ostatnie 5 zastosowanych migracji
yii migrate/history all # pokazuje wszystkie zastosowane migracje

yii migrate/new         # pokazuje pierwsze 10 nowych migracji
yii migrate/new 5       # pokazuje pierwsze 5 nowych migracji
yii migrate/new all     # pokazuje wszystkie nowe migracje
```

6.2.8 Modyfikowanie historii migracji

Czasem zamiast aplikowania lub odwracania migracji, możesz chcieć po prostu zaznaczyć, że baza danych zostałą już uaktualniona do konkretnej migracji. Może się tak zdarzyć, gdy ręcznie modyfikujesz bazę i nie chcesz, aby migracja(e) z tymi zmianami zostały potem ponownie zaaplikowane. Możesz to osiągnąć w następujący sposób:

```
yii migrate/mark 150101_185401                      # używając znacznika
czasu z nazwy migracji
yii migrate/mark "2015-01-01 18:54:01"              # używając ␣łańcucha
znaków, który może być sparsowany przez strtotime()
yii migrate/mark m150101_185401_create_news_table   # używając pe␣lnej nazwy
yii migrate/mark 1392853618                         # używając UNIXowego
znacznika czasu
```

Komenda zmodyfikuje tabelę `migration` poprzez dodanie lub usunięcie wierszy, aby zaznaczyć, że baza danych ma już zastosowane migracje aż do tej określonej w komendzie. Migracje nie zostaną faktycznie zastosowane lub usunięte.

6.2.9 Dostosowywanie migracji

Dostępnych jest kilka opcji pozwalających na dostosowanie komendy migracji do własnych potrzeb.

Użycie opcji linii komend

Komenda migracji ma kilka opcji, które pozwalają na zmianę jej działąnia:

- `interactive`: boolean (domyślnie `true`), określa czy przeprowadzić migrację w trybie interaktywnym. Jeśli ustawione jest `true`, użytkownik będzie poproszony o potwierdzenie przed wykonaniem określonych operacji. Możesz chcieć zmienić to ustawienie na `false`, jeśli komenda ma być używana w tle.
- `migrationPath`: string|array (domyślnie `@app/migrations`), określa folder, gdzie znajdują się wszystkie pliki migracji. Parametr może być określony jako rzeczywista ścieżka lub alias. Zwróć uwagę na to, że folder musi istnieć, inaczej okmenda może wywołać błąd. Począwszy od wersji 2.0.12 można tutaj podać tablicę, aby załadować migracje z wielu źródeł.
- `migrationTable`: string (domyślnie `migration`), określa nazwę tabeli w bazie danych, gdzie trzymana będzie historia migracji. Tabela będzie automatycznie stworzona, jeśli nie istnieje. Możesz również utworzyć ją ręcznie używając struktury `version varchar(255) primary key, apply_time integer`.

- `db`: string (domyślnie `db`), określa identyfikator bazodanowego komponentu aplikacji. Reprezentuje on bazę danych, na której będą zastosowane migracje.
- `templateFile`: string (domyślnie `@yii/views/migration.php`), określa ścieżkę pliku szablonu, używanego do generowania szkieletu plików migracji. Parametr może być określony jako rzeczywista ścieżka lub alias. Plik szablonu jest skryptem PHP, w którym możesz użyć predefiniowanej zmiennej `$className`, aby pobrać nazwę klasy migracji.
- `generatorTemplateFiles`: array (domyślnie '[

  ```
  'create_table' => '@yii/views/createTableMigration.php',
  'drop_table' => '@yii/views/dropTableMigration.php',
  'add_column' => '@yii/views/addColumnMigration.php',
  'drop_column' => '@yii/views/dropColumnMigration.php',
  'create_junction' => '@yii/views/createTableMigration.php'
  ```

  ]'), określa pliki szablonów do generowania kodu migracji. Po więcej szczegółów przejdź do "Generowanie migracji".
- `fields`: tablica definicji kolumn w postaci łańcuchów znaków do wygenerowania kodu migracji. Domyślnie `[]`. Format każdej definicji to `NAZWA_KOLUMNY:TYP_KOLUMNY:DEKORATOR_KOLUMNY`. Dla przykładu, `--fields=name:string(12):notNull` generuje kolumnę typu "string" o rozmiarze 12, która nie może mieć wartości `null`.

Poniższy przykład pokazuje jak można użyć tych opcji.

Chcemy zmigrować moduł `forum`, którego pliki migracji znajdują się w folderze `migrations` modułu - używamy następującej komendy:

```
# stosuje migracje dla modu␣lu forum w trybie nieinteraktywnym
yii migrate --migrationPath=@app/modules/forum/migrations --interactive=0
```

Konfigurowanie komendy globalnie

Zamiast podawać żmudnie te same opcje za każdym razem, gdy uruchamiamy komendę migracji, można ją skonfigurować w konfiguracji aplikacji:

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationTable' => 'backend_migration',
        ],
    ],
];
```

Powyższa konfiguracja powoduje, że z każdym uruchomieniem komendy migracji, tabela `backend_migration` jest używana do zapisu historii migracji i nie musisz już określać jej za pomocą opcji linii komend `migrationTable`.

Migracje w przestrzeni nazw

Począwszy od 2.0.10 możliwe jest używanie przestrzeni nazw w klasach migracji. Możesz zdefiniować listę przestrzeni nazw za pomocą `migrationNamespaces`. Korzystanie z przestrzeni nazw pozwala na łatwe używanie wielu źródeł migracji. Przykładowo:

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => null, // wy␣łącz ścieżkę do folderu migracji,
            jeśli dodajesz app\migrations na liście poniżej
            'migrationNamespaces' => [
                'app\migrations', // Wspólne migracje dla ca␣łej aplikacji
                'module\migrations', // Migracje konkretnego modu␣łu
                'some\extension\migrations', // Migracje konkretnego
                rozszerzenia
            ],
        ],
    ],
];
```

> **Uwaga:** Migracje zaaplikowane z różnych przestrzeni nazw będą dodane do **pojedynczej** historii migracji, przez co np. niemożliwym jest zastosowanie lub cofnięcie migracji z tylko wybranej przestrzeni nazw.

Wykonując operacje na migracjach z przestrzeni nazw: dodając nowe, odwracając je, itd., należy podać pełną przestrzeń nazw przed nazwą migracji. Zwróć uwagę na to, że odwrotny ukośnik (\) jest zwykle uważany za znak specjalny linii komend, zatem musisz odpowiednio zastosować symbol ucieczki, aby uniknąć błędów konsoli i niespodziewanych skutków komendy. Dla przykładu:

```
yii migrate/create app\\migrations\\CreateUserTable
```

> **Uwaga:** Migracje, których lokalizacja określona jest poprzez `migrationPath` nie mogą zawierać przestrzeni nazw. Migracje w przestrzeni nazw mogą być zaaplikowane tylko jeśli są wymienione we właściwości `yii\console\controllers\MigrateController ::$migrationNamespaces`.

Począwszy od wersji 2.0.12 właściwość `migrationPath` pozwala również na podanie tablicy wymieniającej wszystkie foldery zawierające migracje bez przestrzeni nazw. Zmiana ta została wprowadzona dla istniejących projektów, które używają migracji z wielu lokalizacji, głównie z zewnętrznych źródeł jak rozszerzenia Yii tworzone przez innych deweloperów, które z tego powodu nie mogą łatwo być zmodyfikowane, aby używać przestrzeni nazw.

Generowanie migracji w przestrzeni nazw Migracje w przestrzeni nazw korzystają z formatu nazw "CamelCase" `M<YYMMDDHHMMSS><Nazwa>` (przykładowo `M190720100234CreateUserTable`). Generując taką migrację pamiętaj, że nazwa tabeli będzie przekonwertowana z formatu "CamelCase" na format "podkreślnikowy". Dla przykładu:

```
yii migrate/create app\\migrations\\DropGreenHotelTable
```

generuje migrację w przestrzeni nazw `app\migrations` usuwającą tabelę `green_hotel`, a

```
yii migrate/create app\\migrations\\CreateBANANATable
```

generuje migrację w przestrzeni nazw `app\migrations` tworzącą tabelę `b_a_n_a_n_a`.

Jeśli nazwa tabeli zawiera małe i wielkie litery (like `studentsExam`), poprzedź nazwę podkreślnikiem:

```
yii migrate/create app\\migrations\\Create_studentsExamTable
```

To wygeneruje migrację w przestrzeni nazw `app\migrations` tworzącą tabelę `studentsExam`.

Rozdzielenie migracji

Czasem korzystanie z pojedynczej historii migracji dla wszystkich migracji w projekcie jest uciążliwe. Dla przykładu, możesz zainstalować rozszerzenie 'blog', zawierające całkowicie oddzielne funkcjonalności i dostarczające własne migracje, które nie powinny wpływać na te dedykowane dla funkcjonalności głównego projektu.

Jeśli chcesz, aby część migracji mogła być zastosowana i śledzona całkowicie niezależnie od pozostałych, możesz skonfigurować kilka komend migracji, które będą używać różnych przestrzeni nazw i tabeli historii migracji:

```
return [
    'controllerMap' => [
        // Wspólne migracje dla ca␣lej aplikacji
        'migrate-app' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['app\migrations'],
            'migrationTable' => 'migration_app',
            'migrationPath' => null,
        ],
        // Migracje dla konkretnego modu␣lu
        'migrate-module' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['module\migrations'],
            'migrationTable' => 'migration_module',
            'migrationPath' => null,
        ],
```

```
        // Migrations dla konkretnego rozszerzenia
        'migrate-rbac' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => '@yii/rbac/migrations',
            'migrationTable' => 'migration_rbac',
        ],
    ],
];
```

Zwróć uwagę na to, że teraz, aby zsynchronizować bazę danych, musisz uruchomić kilka komend zamiast jednej:

```
yii migrate-app
yii migrate-module
yii migrate-rbac
```

6.2.10 Migrowanie wielu baz danych

Domyślnie migracje są stosowane do jednej bazy danych określonej przez komponent aplikacji `db`. Jeśli chcesz, aby były zastosowane do innej bazy, musisz zdefiniować opcję `db` w linii komend, jak poniżej,

```
yii migrate --db=db2
```

Ta komenda zastosuje migracje do bazy `db2`.

Czasem konieczne jest, aby zastosować *niektóre* migracje do jednej bazy, a inne do drugiej. Aby to uzyskać, podczas implementacji klasy migracji należy bezpośrednio wskazać identyfikator komponentu bazy danych, który migracja ma użyć, jak poniżej:

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function init()
    {
        $this->db = 'db2';
        parent::init();
    }
}
```

Ta migracja będzie zastosowana do bazy `db2`, nawet jeśli w opcjach komendy określona będzie inna baza. Zwróć uwagę na to, że historia migracji będzie uaktualniona wciąż w bazie danych określonej przez opcję `db` linii komend.

Jeśli masz wiele migracji korzystających z tej samej bazy danych, zalecane jest utworzenie bazowej klasy migracji z powyższym kodem metody `init()`, a następnie dziedziczenie po niej w każdej kolejnej migracji.

> **Wskazówka:** Oprócz ustawiania właściwości `db`, możesz również operować na różnych bazach poprzez tworzenie nowych połączeń bazodanowych w klasach migracji, a następnie korzystanie z metod DAO i tych połączeń.

Inną strategią migracji wielu baz danych jest utrzymywanie migracji dla różnych baz w różnych folderach migracji. Dzięki temu możesz te bazy migrować w osobnych komendach:

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1
yii migrate --migrationPath=@app/migrations/db2 --db=db2
...
```

Pierwsza komenda zastosuje migracje z folderu `@app/migrations/db1` na bazie `db1`, druga migracje z folderu `@app/migrations/db2` na bazie `db2`, itd.

Rozdział 7

Odbieranie danych od użytkowników

7.1 Tworzenie formularzy

7.1.1 Formularze oparte na ActiveRecord: ActiveForm

Podstawowym sposobem korzystania z formularzy w Yii jest użycie `ActiveForm`. Ten sposób powinien być używany, jeśli formularz jest bazowany na modelu. Dodatkowo, klasa `Html` zawiera sporo użytecznych metod, które zazwyczaj używane są do dodawania przycisków i tekstów pomocniczych do każdego formularza.

Formularz, który jest wyświetlany po stronie klienta, w większości przypadków, posiada odpowiedni model, który jest używany do walidacji danych wejściowych po stronie serwera (sprawdź sekcję Walidacja danych wejściowych aby uzyskać więcej szczegółów).
Podczas tworzenia formularza na podstawie modelu, pierwszym krokiem jest zdefiniowanie samego modelu. Model może być bazowany na klasie Active Record, reprezentując dane z bazy danych, lub może być też bazowany na klasie generycznej `Model`, aby przechwytywać dowolne dane wejściowe, np. formularz logowania.

> **Wskazówka:** Jeśli pola formularza są różne od kolumn tabeli w bazie danych lub też występuje tu formatowanie i logika specyficzna tylko dla tego formularza, zaleca się stworzenie oddzielnego modelu rozszerzającego `yii\base\Model`.

W poniższym przykładzie pokażemy, jak model generyczny może być użyty do stworzenia formularza logowania:

```
<?php

class LoginForm extends \yii\base\Model
```

```
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // zasady walidacji
        ];
    }
}
```

W kontrolerze przekażemy instancję tego modelu do widoku, gdzie widżet `ActiveForm` zostanie użyty do wyświetlenia formularza:

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

$form = ActiveForm::begin([
    'id' => 'login-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>

    <div class="form-group">
        <div class="col-lg-offset-1 col-lg-11">
            <?= Html::submitButton('Login', ['class' => 'btn btn-primary'])
?>
        </div>
    </div>
<?php ActiveForm::end() ?>
```

Otaczanie kodu przez i

W powyższym kodzie, `begin()` nie tylko tworzy instancję formularza, ale zaznacza też jego początek. Cała zawartość położona pomiędzy `begin()` i `end()` zostanie otoczona tagiem HTML'owym `<form>`. Jak w przypadku każdego widżetu, możesz określić kilka opcji z jakimi widżet powinien być skonfigurowany przez przekazanie tablicy do metody `begin`. W tym przypadku dodatkowa klasa CSS i identyfikator ID zostały przekazane do otwierającego tagu `<form>`. Aby zobaczyć wszystkie dostępne opcje, zajrzyj do dokumentacji API `ActiveForm`.

Do utworzenia formularza, wraz z elementami etykiet oraz wszelkimi walidacjami JavaScript, wywoływana jest metoda `field()`, która zwraca instancję obiektu `ActiveField`. Kiedy rezultat tej metody jest bezpośrednio wyświetlany, tworzone jest regularne pole tekstowe. Aby dostosować pola, możesz używać dodatkowych metod łączonych `ActiveField`:

```
// pole has␣la
<?= $form->field($model, 'password')->passwordInput() ?>
// dodanie podpowiedzi oraz zmiana etykiety
<?= $form->field($model, 'username')->textInput()->hint('Please enter your
name')->label('Name') ?>
// utworzenie pola email w formacie HTML5
<?= $form->field($model, 'email')->input('email') ?>
```

Powyższy kod utworzy tagi <label>, <input> oraz wszystkie inne, według pól formularza zdefiniowanych w template. Nazwa pola określana jest automatycznie z modelu formName() i nazwy atrybutu. Dla przykładu, nazwą pola dla atrybutu username w powyższym przykładzie będzie LoginForm[username]. Ta zasada nazewnictwa spowoduje, że tablica wszystkich atrybutów z formularza logowania będzie dostępna w zmiennej $_POST['LoginForm'] po stronie serwera.

Określanie atrybutów modelu może być wykonane w bardziej wyrafinowany sposób. Dla przykładu, kiedy atrybut będzie potrzebował pobierać tablicę wartości, podczas przesyłania wielu plików lub wybrania wielu pozycji, możesz określić go jako tablicę dodając [] do nazwy atrybutu:

```
// pozwól na przes␣lanie wielu plików
echo $form->field($model,
'uploadFile[]')->fileInput(['multiple'=>'multiple']);

// pozwól na zaznaczenie wielu pozycji
echo $form->field($model, 'items[]')->checkboxList(['a' => 'Item A', 'b' =>
'Item B', 'c' => 'Item C']);
```

Bądź ostrożny podczas nazywania elementów formularza takich jak przyciski wysyłania. Odnosząc się do dokumentacji jQuery[1], istnieje kilka zarezerwowanych nazw, które mogą powodować konflikty.

> Formularz i jego elementy podrzędne powinny nie używać nazw pól lub nazw identyfikatorów które tworzą konflikt z właściwościami formularza, takich jak submit, length lub method. Konflikty nazw mogą powodować mylące błędy. Kompletna lista zasad oraz walidator znaczników dla tych problemów znajduje się na stronie DOMLint[2].

Dodatkowe tagi HTML mogą zostać dodane do formularza używając czystego HTML'a lub używając metody z klasy pomocniczej - Html, tak jak było to zrobione w przykładzie wyżej z submitButton().

> **Wskazówka:** Jeśli używasz Twitter Bootstrap CSS w Twojej aplikacji, możesz użyć yii\bootstrap\ActiveForm zamiast yii

[1] https://api.jquery.com/submit
[2] https://kangax.github.io/domlint

`\widgets\ActiveForm`. Rozszerza on `ActiveForm` i podczas generowania pól formularza używa stylu specyficznego dla Bootstrap.

Wskazówka: Jeśli chcesz oznaczyć wymagane pola gwiazdką, możesz użyć poniższego kodu CSS:

```
div.required label:after {
    content: " *";
    color: red;
}
```

7.1.2 Tworzenie list

Wyróżniamy trzy typy list:

- Listy rozwijane
- Listy opcji typu radio
- Listy opcji typu checkbox

Aby stworzyć listę, musisz najpierw przygotować jej elementy. Można to zrobić ręcznie:

```
$items = [
    1 => 'item 1',
    2 => 'item 2'
]
```

lub też pobierając elementy z bazy danych:

```
$items = Category::find()
        ->select(['label'])
        ->indexBy('id')
        ->column();
```

Elementy `$items` muszą być następnie przetworzone przez odpowiednie widżety list. Wartość pola formularza (i aktualnie aktywny element) będzie automatycznie ustawiony przez aktualną wartość atrybutu `$model`.

Tworzenie listy rozwijanej Możemy użyć metody klasy ActiveForm `yii\widgets\ActiveForm::dropDownList()` do utworzenia rozwijanej listy:

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->dropdownList([
        1 => 'item 1',
        2 => 'item 2'
    ],
    ['prompt'=>'Wybierz kategorię']
);
```

Tworzenie radio listy Do stworzenia takiej listy możemy użyć metody ActiveField `yii\widgets\ActiveField::radioList()`:

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->radioList([
    1 => 'radio 1',
    2 => 'radio 2'
]);
```

Tworzenie checkbox listy Do stworzenia takiej listy możemy użyć metody ActiveField `yii\widgets\ActiveField::checkboxList()`:

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->checkboxList([
    1 => 'checkbox 1',
    2 => 'checkbox 2'
]);
```

7.1.3 Praca z Pjaxem

Widżet `Pjax` pozwala na aktualizację określonej sekcji strony, zamiast przeładowywania jej całkowicie. Możesz użyć go do odświeżenia formularza i podmienić jego zawartość po wysłaniu danych.

Możesz skonfigurować `$formSelector`, aby wskazać, które formularze powinny wyzwalać użycie pjaxa. Jeśli nie zostanie to ustawione inaczej, wszystkie formularze z atrybutem `data-pjax` objęte widżetem Pjax będą wyzwalały jego użycie.

```
use yii\widgets\Pjax;
use yii\widgets\ActiveForm;

Pjax::begin([
    // opcje Pjaxa
]);
    $form = ActiveForm::begin([
        'options' => ['data' => ['pjax' => true]],
        // więcej opcji ActiveForm
    ]);

        // zawartość ActiveForm

    ActiveForm::end();
Pjax::end();
```

> **Wskazówka:** Należy być ostrożnym z użyciem linków wewnątrz widżetu `Pjax`, ponieważ ich cel również zostanie wyrenderowany wewnątrz widżetu. Aby temu zapobiec, należy użyć atrybutu HTML `data-pjax="0"`.

Wartości w przyciskach submit i przesyłanie plików Znane są problemy z użyciem `jQuery.serializeArray()` podczas obsługi [[https://github.com/jquery/jquery/issues/2321|plik\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\endgroup\relax\let\ignorespaces\relax\accent1o\egroup\spacefactor\accent@spacefactorw]] i [[https://github.com/jquery/jquery/issues/2321|warto\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{s\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\endgroup\relax\let\ignorespaces\relax\accent1s\egroup\spacefactor\accent@spacefactorci przycisku submit]], które nie będą jednak rozwiązane i zamiast tego zostały porzucone na rzecz klasy `FormData` wprowadzonej w HTML5.

Oznacza to, że oficjalne wsparcie dla plików i wartości przycisku submit używanych w połączeniu z ajaxem lub widżetem `Pjax` zależy od [[https://developer.mozilla.org/en-US/docs/Web/API/FormData#browser_compatibility|wsparcia przeglądarki]] dla klasy `FormData`.

7.1.4 Dalsza lektura

Następna sekcja Walidacja danych wejściowych dotyczy walidacji przesłanych przed formularz danych po stronie serwera, przy użyciu ajax oraz walidacji po stronie klienta.

Aby przeczytać o bardziej złożonych użyciach formularzy możesz zajrzeć do poniższych sekcji:

- Odczytywanie tablicowych danych wejściowych - do pobierania danych dla wielu modeli tego samego rodzaju.
- Pobieranie danych dla wielu modeli - do obsługi wielu różnych modeli w tym samym formularzu.
- Wysyłanie plików - jak używać formularzy do przesyłania plików.

7.2 Walidacja danych wejściowych

Jedna z głównych zasad mówi, że nigdy nie należy ufać danym otrzymanym od użytkownika oraz że zawsze należy walidować je przed użyciem.

Rozważmy model wypełniony danymi pobranymi od użytkownika. Możemy zweryfikować je poprzez wywołanie metody `validate()`. Metoda zwróci wartość `boolean` wskazującą, czy walidacja się powiodła, czy też nie. Jeśli nie, można pobrać informacje o błędach za pomocą właściwości `errors`. Dla przykładu,

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

// uzupe␣lniamy model danymi od użytkownika
$model->load(\Yii::$app->request->post());
// ten zapis jest tożsamy z poniższą metodą
```

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

if ($model->validate()) {
    // akcja w przypadku poprawnej walidacji
} else {
    // akcja w przypadku niepoprawnej walidacji. Zmienna $errors jest
    tablicą zawierającą wiadomości b␣łędów
    $errors = $model->errors;
}
```

7.2.1 Deklaracja zasad

Aby metoda `validate()` naprawdę zadziałała, należy zdefiniować zasady walidacji dla atrybutów, które mają jej podlegać. Powinno zostać to zrobione przez nadpisanie metody `rules()`. Poniższy przykład pokazuje jak zostały zadeklarowane zasady walidacji dla modelu `ContactForm`:

```
public function rules()
{
    return [
        // atrybuty name, email, subject oraz body są wymagane
        [['name', 'email', 'subject', 'body'], 'required'],

        // atrybut email powinien być poprawnym adresem email
        ['email', 'email'],
    ];
}
```

Metoda `rules()` powinna zwracać tablicę zasad, gdzie każda zasada jest również tablicą o następującym formacie:

```
[
    // wymagane, określa atrybut który powinien zostać zwalidowany przez tę
    zasadę.
    // Dla pojedyńczego atrybutu możemy użyć bezpośrednio jego nazwy, bez
    osadzania go w tablicy
    ['attribute1', 'attribute2', ...],

    // wymagane, określa rodzaj walidacji
    // Może to być nazwa klasy, alias walidatora lub nazwa metody walidacji
    'validator',

    // opcjonalne, określa, w którym scenariuszu/scenariuszach ta zasada
    powinna zostać użyta
    // w przypadku nie podania żadnego argumentu zasada zostanie
    zaaplikowana do wszystkich scenariuszy
    // Możesz również skonfigurować opcję "except", jeśli chcesz użyć tej
    zasady dla wszystkich scenariuszy, z wyjątkiem wymienionych
    'on' => ['scenario1', 'scenario2', ...],

    // opcjonalne, określa dodatkową konfigurację dla obiektu walidatora
    'property1' => 'value1', 'property2' => 'value2', ...
]
```

Dla każdej z zasad musisz określić co najmniej jeden atrybut, którego ma ona dotyczyć, oraz określić rodzaj zasady jako jedną z następujących form:

- alias walidatora podstawowego, np. `required`, `in`, `date` itd. Zajrzyj do sekcji Podstawowe walidatory, aby uzyskać pełną listę walidatorów podstawowych.
- nazwa metody walidacji w klasie modelu lub funkcja anonimowa. Po więcej szczegółów zajrzyj do sekcji Walidatory wbudowane.
- pełna nazwa klasy walidatora. Po więcej szczegółów zajrzyj do sekcji Walidatory niezależne.

Zasada może zostać użyta do walidacji jednego lub wielu atrybutów, a atrybut może być walidowany przez jedną lub wiele zasad. Zasada może zostać użyta dla konkretnych scenariuszy przez dodanie opcji `on`. Jeśli nie dodasz opcji `on` oznacza to, że zasada zostanie użyta w każdym scenariuszu.

Wywołanie metody `validate()` powoduje podjęcie następujących kroków w celu wykonania walidacji:

1. Określenie, które atrybuty powinny zostać zweryfikowane poprzez pobranie ich listy z metody `scenarios()`, używając aktualnego `scenariusza`. Wybrane atrybuty nazywane są *atrybutami aktywnymi*.

2. Określenie, które zasady walidacji powinny zostać użyte przez pobranie ich listy z metody `rules()`, używając aktualnego `scenariusza`. Wybrane zasady nazywane są *zasadami aktywnymi*.

3. Użycie każdej aktywnej zasady do walidacji każdego aktywnego atrybutu, który jest powiązany z konkretną zasadą. Zasady walidacji są wykonywane w kolejności, w jakiej zostały zapisane.

Odnosząc się do powyższych kroków, atrybut zostanie zwalidowany wtedy i tylko wtedy, gdy jest on aktywnym atrybutem zadeklarowanym w `scenarios()` oraz jest powiązany z jedną lub wieloma aktywnymi zasadami zadeklarowanymi w `rules()`.

Uwaga: Czasem użyteczne jest nadanie nazwy zasadzie np.

```
public function rules()
{
    return [
        // ...
        'password' => [['password'], 'string', 'max' => 60],
    ];
}
```

W modelu potomnym można to wykorzystać:

```
public function rules()
{
```

```
        $rules = parent::rules();
        unset($rules['password']);
        return $rules;
    }
```

Dostosowywanie wiadomości błędów

Większość walidatorów posiada domyślne wiadomości błędów, które zostają dodane do poddanego walidacji modelu, kiedy któryś z atrybutów nie przejdzie pomyślnie walidacji. Dla przykładu, walidator `required` dodaje komunikat "Username cannot be blank.", kiedy atrybut `username` nie przejdzie walidacji tej zasady.

Możesz dostosować wiadomość błędu zasady przez określenie właściwości `message` podczas jej deklaracji. Dla przykładu,

```
public function rules()
{
    return [
        ['username', 'required', 'message' => 'Proszę wybrać login.'],
    ];
}
```

Niektóre walidatory mogą wspierać dodatkowe wiadomości błedów, aby bardziej precyzyjnie określić problemy powstałe przy walidacji. Dla przykładu, walidator `number` dodaje `tooBig` oraz `tooSmall` do opisania sytuacji, kiedy poddawana walidacji liczba jest za duża lub za mała. Możesz skonfigurować te wiadomości tak, jak pozostałe właściwości walidatorów podczas deklaracji zasady.

Zdarzenia walidacji

Podczas wywołania metody `validate()` zostaną wywołane dwie metody, które możesz nadpisać, aby dostosować proces walidacji:

- `beforeValidate()`: domyślna implementacja wywoła zdarzenie `EVENT_BEFORE_VALIDATE`. Możesz nadpisać tę metodę lub odnieść się do zdarzenia, aby wykonać dodatkowe operacje przed walidacją. Metoda powinna zwracać wartość `boolean` wskazującą, czy walidacja powinna zostać przeprowadzona, czy też nie.
- `afterValidate()`: domyślna implementacja wywoła zdarzenie `EVENT_AFTER_VALIDATE`. Możesz nadpisać tę metodę lub odnieść się do zdarzenia, aby wykonać dodatkowe operacje po zakończonej walidacji.

Walidacja warunkowa

Aby zwalidować atrybuty tylko wtedy, gdy zostaną spełnione pewne założenia, np. walidacja jednego atrybutu zależy od wartości drugiego atrybutu, możesz użyć właściwości `when`, aby zdefiniować taki warunek. Dla przykładu,

```
[
    ['state', 'required', 'when' => function($model) {
        return $model->country == 'USA';
    }],
]
```

Właściwość `when` pobiera możliwą do wywołania funkcję PHP z następującą definicją:

```
/**
 * @param Model $model model, który podlega walidacji
 * @param string $attribute atrybut, który podlega walidacji
 * @return bool wartość zwrotna; czy regu␣la powinna zostać zastosowana
 */
function ($model, $attribute)
```

Jeśli potrzebujesz również wsparcia walidacji warunkowej po stronie użytkownika, powinieneś skonfigurować właściwość `whenClient`, która przyjmuje wartość `string` reprezentującą funkcję JavaScript, zwracającą wartość `boolean`, która będzie określała, czy zasada powinna zostać zastosowana, czy nie. Dla przykładu,

```
[
    ['state', 'required', 'when' => function ($model) {
        return $model->country == 'USA';
    }, 'whenClient' => "function (attribute, value) {
        return $('#country').val() == 'USA';
    }"],
]
```

Filtrowanie danych

Dane od użytkownika często muszą zostać przefiltrowane. Dla przykładu, możesz chcieć wyciąć znaki spacji na początku i na końcu pola `username`. Aby osiągnąć ten cel, możesz również użyć zasad walidacji.

Poniższy przykład pokazuje, jak wyciąć znaki spacji z pola oraz zmienić puste pole na wartość `null` przy użyciu podstawowych walidatorów trim oraz default:

```
[
    [['username', 'email'], 'trim'],
    [['username', 'email'], 'default'],
]
```

Możesz użyć również bardziej ogólnego walidatora filter, aby przeprowadzić złożone filtrowanie.

Jak pewnie zauważyłeś, te zasady walidacji tak naprawdę nie walidują danych. Zamiast tego przetwarzają wartości, a następnie przypisują je do atrybutów, które zostały poddane walidacji.

Obsługa pustych danych wejściowych

Kiedy dane wejściowe są wysłane przez formularz HTML, często zachodzi potrzeba przypisania im domyślnych wartości, jeśli są puste. Możesz to osiągnąć przez użycie walidatora default. Dla przykładu,

```
[
    // ustawia atrybuty "username" oraz "email" jako `null` jeśli są puste
    [['username', 'email'], 'default'],

    // ustawia atrybut "level" równy "1", jeśli jest pusty
    ['level', 'default', 'value' => 1],
]
```

Domyślnie pole uważane jest za puste, jeśli jego wartość to pusty łańcuch znaków, pusta tablica lub `null`. Możesz dostosować domyślną logikę wykrywania pustych pól przez skonfigurowanie parametru `isEmpty`, przekazując mu funkcję PHP. Dla przykładu,

```
[
    ['agree', 'required', 'isEmpty' => function ($value) {
        return empty($value);
    }],
]
```

> **Uwaga:** Większość walidatorów nie obsługuje pustych pól, jeśli ich właściwość `skipOnEmpty` przyjmuje domyślnie wartość `true`. Zostaną one po prostu pominięte podczas walidacji, jeśli ich powiązany atrybut otrzyma wartość uznawaną za pustą. Wśród podstawowych walidatorów, tylko walidatory `captcha`, `default`, `filter`, `required` oraz `trim` obsługują puste pola.

7.2.2 Walidacja "Ad Hoc"

Czasami potrzebna będzie walidacja *ad hoc* dla wartości które nie są powiązane z żadnym modelem.

Jeśli potrzebujesz wykonać tylko jeden typ walidacji (np. walidację adresu email), możesz wywołać metodę `validate()` wybranego walidatora, tak jak poniżej:

```
$email = 'test@example.com';
$validator = new yii\validators\EmailValidator();

if ($validator->validate($email, $error)) {
    echo 'Email is valid.';
} else {
    echo $error;
}
```

> **Uwaga:** Nie każdy walidator wspiera tego typu walidację. Dla przykładu, podstawowy walidator unique został zaprojektowany do pracy wyłącznie z modelami.

Jeśli potrzebujesz przeprowadzić wielokrotne walidacje, możesz użyć modelu `DynamicModel`, który wspiera deklarację atrybutów oraz zasad walidacji "w locie". Dla przykładu,

```
public function actionSearch($name, $email)
{
    $model = DynamicModel::validateData(compact('name', 'email'), [
        [['name', 'email'], 'string', 'max' => 128],
        ['email', 'email'],
    ]);

    if ($model->hasErrors()) {
        // validation fails
    } else {
        // validation succeeds
    }
}
```

Metoda `validateData()` tworzy instancję `DynamicModel`, definiuje atrybuty używając przekazanych danych (`name` oraz `email` w tym przykładzie), a następnie wywołuje metodę `validate()` z podanymi zasadami walidacji.

Alternatywnie, możesz użyć bardziej "klasycznego" zapisu to przeprowadzenia tego typu walidacji:

```
public function actionSearch($name, $email)
{
    $model = new DynamicModel(compact('name', 'email'));
    $model->addRule(['name', 'email'], 'string', ['max' => 128])
        ->addRule('email', 'email')
        ->validate();

    if ($model->hasErrors()) {
        // validation fails
    } else {
        // validation succeeds
    }
}
```

Po walidacji możesz sprawdzić, czy przebiegła ona poprawnie, poprzez wywołanie metody `hasErrors()`, a następnie pobrać błędy walidacji z właściwości `errors`, tak jak w przypadku zwykłego modelu. Możesz również uzyskać dostęp do dynamicznych atrybutów tej instancji, np. `$model->name` i `$model->email`.

7.2.3 Tworzenie walidatorów

Oprócz używania podstawowych walidatorów dołączonych do wydania Yii, możesz dodatkowo utworzyć własne; wbudowane lub niezależne.

Walidatory wbudowane

Wbudowany walidator jest zdefiniowaną w modelu metodą lub funkcją anonimową. Jej definicja jest następująca:

```
/**
 * @param string $attribute atrybut podlegający walidacji
 * @param mixed $params wartość parametru podanego w zasadzie walidacji
 * @param \yii\validators\InlineValidator $validator powiązana instancja
InlineValidator
 * Ten parametr jest dostępny od wersji 2.0.11.
 * @param mixed $current aktualnie walidowana wartość atrybutu.
 * Ten parametr jest dostępny od wersji 2.0.36.
 */
function ($attribute, $params, $validator, $current)
```

Jeśli atrybut nie przejdzie walidacji, metoda/funkcja powinna wywołać metodę `addError()` do zapisania wiadomości o błędzie w modelu, która może zostać później pobrana i zaprezentowana użytkownikowi.

Poniżej znajduje się kilka przykładów:

```
use yii\base\Model;

class MyForm extends Model
{
    public $country;
    public $token;

    public function rules()
    {
        return [
            // Wbudowany walidator zdefiniowany jako metoda
            validateCountry() w modelu
            ['country', 'validateCountry'],

            // Wbudowany walidator zdefiniowany jako funkcja anonimowa
            ['token', function ($attribute, $params, $validator) {
                if (!ctype_alnum($this->$attribute)) {
                    $this->addError($attribute, 'Token musi zawierać litery
                    lub cyfry.');
                }
            }],
        ];
    }

    public function validateCountry($attribute, $params, $validator)
    {
        if (!in_array($this->$attribute, ['USA', 'Web'])) {
            $this->addError($attribute, 'Wybrany kraj musi być jednym z:
            "USA", "Web".');
        }
    }
}
```

> **Uwaga:** Począwszy od wersji 2.0.11 możesz użyć `yii\validators\InlineValidator::addError()`, aby dodać błędy bezpośrednio. W tym sposobie treść błędu może być sformatowana bezpośrednio za pomocą `yii\i18n\I18N::format()`. Użyj `{attribute}` i `{value}` w treści błędu, aby odwołać się odpowiednio do etykiety atrybutu (bez konieczności pobierania jej ręcznie) i wartości atrybutu:
>
> ```
> $validator->addError($this, $attribute, 'Wartość "{value}" nie jest
> poprawna dla {attribute}.');
> ```

> **Uwaga:** Domyślnie wbudowane walidatory nie zostaną zastosowane, jeśli ich powiązane atrybuty otrzymają puste wartości lub wcześniej nie przeszły którejś z zasad walidacji. Jeśli chcesz się upewnić, że zasada zawsze zostanie zastosowana, możesz skonfigurować właściwość `skipOnEmpty` i/lub `skipOnError`, przypisując jej wartość `false` w deklaracji zasady walidacji. Dla przykładu:
>
> ```
> [
>     ['country', 'validateCountry', 'skipOnEmpty' => false,
>     'skipOnError' => false],
> ]
> ```

Walidatory niezależne

Walidator niezależy jest klasą rozszerzającą `Validator` lub klasy po nim dziedziczące. Możesz zaimplementować jego logikę walidacji poprzez nadpisanie metody `validateAttribute()`. Jeśli atrybut nie przejdzie walidacji, wywołaj metodę `addError()` do zapisania wiadomości błędu w modelu, tak jak w walidatorach wbudowanych.

Dla przykładu, poprzedni wbudowany walidator mógłby zostać przeniesiony do nowej klasy `components/validators/CountryValidator`.

```
namespace app\components;

use yii\validators\Validator;

class CountryValidator extends Validator
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['USA', 'Web'])) {
            $this->addError($model, $attribute, 'Wybrany kraj musi być
            jednym z: "USA", "Web".');
        }
    }
}
```

Jeśli chcesz, aby walidator wspierał walidację wartości bez modelu, powinieneś nadpisać metodę `validate()`. Możesz nadpisać także `validateValue()`

zamiast `validateAttribute()` oraz `validate()`, ponieważ domyślnie te dwie metody są implementowane użyciem metody `validateValue()`.

Poniżej znajduje się przykład użycia powyższej klasy walidatora w modelu.

```
namespace app\models;

use Yii;
use yii\base\Model;
use app\components\validators\CountryValidator;

class EntryForm extends Model
{
    public $name;
    public $email;
    public $country;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['country', CountryValidator::class],
            ['email', 'email'],
        ];
    }
}
```

7.2.4 Walidacja wielu atrybutów na raz

Zdarza się, że walidatory sprawdzają wiele atrybutów jednocześnie. Rozważmy następujący formularz:

```
class MigrationForm extends \yii\base\Model
{
    /**
     * Kwota minimalnych funduszy dla jednej doros␣lej osoby
     */
    const MIN_ADULT_FUNDS = 3000;
    /**
     * Kwota minimalnych funduszy dla jednego dziecka
     */
    const MIN_CHILD_FUNDS = 1500;

    public $personalSalary;
    public $spouseSalary;
    public $childrenCount;
    public $description;

    public function rules()
    {
        return [
            [['personalSalary', 'description'], 'required'],
```

```
            [['personalSalary', 'spouseSalary'], 'integer', 'min' =>
            self::MIN_ADULT_FUNDS],
            ['childrenCount', 'integer', 'min' => 0, 'max' => 5],
            [['spouseSalary', 'childrenCount'], 'default', 'value' => 0],
            ['description', 'string'],
        ];
    }
}
```

Tworzenie walidatora

Powiedzmy, że chcemy sprawdzić, czy dochód rodziny jest wystarczający do utrzymania dzieci. W tym celu możemy utworzyć wbudowany walidator `validateChildrenFunds`, który będzie uruchamiany tylko jeśli `childrenCount` będzie większe niż 0.

Zwróć uwagę na to, że nie możemy użyć wszystkich walidowanych atrybutów (`['personalSalary', 'spouseSalary', 'childrenCount']`) przy dołączaniu walidatora. Wynika to z tego, że ten sam walidator będzie uruchomiony dla każdego z atrybutów oddzielnie (łącznie 3 razy), a musimy użyć go tylko raz dla całego zestawu atrybutów.

Możesz użyć dowolnego z tych atrybutów zamiast podanego poniżej (lub też tego, który uważasz za najbardziej tu odpowiedni):

```
['childrenCount', 'validateChildrenFunds', 'when' => function ($model) {
    return $model->childrenCount > 0;
}],
```

Implementacja `validateChildrenFunds` może wyglądać następująco:

```
public function validateChildrenFunds($attribute, $params)
{
    $totalSalary = $this->personalSalary + $this->spouseSalary;
    // Podwój minimalny fundusz doros␣lych, jeśli ustalono zarobki
    wspó␣lma␣lżonka
    $minAdultFunds = $this->spouseSalary ? self::MIN_ADULT_FUNDS * 2 :
    self::MIN_ADULT_FUNDS;
    $childFunds = $totalSalary - $minAdultFunds;
    if ($childFunds / $this->childrenCount < self::MIN_CHILD_FUNDS) {
        $this->addError('childrenCount', 'Twoje zarobki nie są
        wystarczające, aby utrzymać dzieci.');
    }
}
```

Możesz zignorować parametr `$attribute`, ponieważ walidacja nie jest powiązana bezpośrednio tylko z jednym atrybutem.

Dodawanie informacji o błędach

Dodawanie błędów walidacji w przypadku wielu atrybutów może różnić się w zależności od ustalonej metodyki pracy z formularzami:

- Można wybrać najbardziej w naszej opinii pole i dodać błąd do jego atrybutu:

```
$this->addError('childrenCount', 'Twoje zarobki nie są wystarczające dla
potrzeb dzieci.');
```

- Można wybrać wiele ważnych odpowiednich atrybutów lub też wszystkie i dodać ten sam błąd do każdego z nich. Możemy przechować treść w oddzielnej zmiennej przed przekazaniem jej do `addError`, aby nie powtarzać się w kodzie (zasada DRY - Don't Repeat Yourself).

```
$message = 'Twoje zarobki nie są wystarczające dla potrzeb dzieci.';
$this->addError('personalSalary', $message);
$this->addError('wifeSalary', $message);
$this->addError('childrenCount', $message);
```

Lub też użyć pętli:

```
$attributes = ['personalSalary, 'wifeSalary', 'childrenCount'];
foreach ($attributes as $attribute) {
    $this->addError($attribute, 'Twoje zarobki nie są wystarczające dla
potrzeb dzieci.');
}
```

- Można też dodać ogólny błąd (niepowiązany z żadnym szczególnym atrybutem). Do tego celu możemy wykorzystać nazwę nieistniejącego atrybutu, na przykład `*`, ponieważ to, czy atrybut istnieje, nie jest sprawdzane w tym kroku.

```
$this->addError('*', 'Twoje zarobki nie są wystarczające dla potrzeb
dzieci.');
```

W rezultacie takiej operacji nie zobaczymy błędu zaraz obok pól formularza. Aby go wyświetlić, możemy dodać do widoku podsumowanie błędów formularza:

```
<?= $form->errorSummary($model) ?>
```

> **Uwaga:** Tworzenie walidatora operującego na wielu atrybutach jednocześnie jest dobrze opisane w książce kucharskiej społeczności Yii[3].

[3] https://github.com/samdark/yii2-cookbook/blob/master/book/forms-validator-multiple-attributes.md

7.2.5 Walidacja po stronie klienta

Walidacja po stronie klienta, bazująca na kodzie JavaScript jest wskazana, kiedy użytkownicy dostarczają dane przez formularz HTML, ponieważ pozwala na szybszą walidację błędów, a tym samym zapewnia lepszą ich obsługę dla użytkownika. Możesz użyć lub zaimplementować walidator, który wspiera walidację po stronie klienta jako *dodatek* do walidacji po stronie serwera.

> **Informacja:** Walidacja po stronie klienta nie jest wymagana. Głównym jej celem jest poprawa jakości korzystania z formularzy dla użytkowników. Podobnie jak w przypadku danych wejściowych pochodzących od użytkowników, nigdy nie powinieneś ufać walidacji przeprowadanej po stronie klienta. Z tego powodu należy zawsze przeprowadzać główną walidację po stronie serwera wywołując metodę `validate()`, tak jak zostało to opisane w poprzednich sekcjach.

Używanie walidacji po stronie klienta

Wiele podstawowych walidatorów domyślnie wspiera walidację po stronie klienta. Wszystko, co musisz zrobić, to użyć widżetu `ActiveForm` do zbudowania formularza HTML. Dla przykładu, model `LoginForm` poniżej deklaruje dwie zasady: jedną, używającą podstawowego walidatora required, który wspiera walidację po stronie klienta i serwera, oraz drugą, w której użyto walidatora wbudowanego `validatePassword`, który wspiera tylko walidację po stronie serwera.

```
namespace app\models;

use yii\base\Model;
use app\models\User;

class LoginForm extends Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // atrybuty username oraz password są wymagane
            [['username', 'password'], 'required'],

            // atrybut password jest walidowany przez validatePassword()
            ['password', 'validatePassword'],
        ];
    }

    public function validatePassword()
```

```
    {
        $user = User::findByUsername($this->username);

        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError('password', 'Nieprawid␣lowa nazwa użytkownika
            lub has␣lo.');
        }
    }
}
```

Formularz HTML zbudowany przez następujący kod zawiera dwa pola: `username` oraz `password`. Jeśli wyślesz formularz bez wpisywania jakichkolwiek danych, otrzymasz komunikaty błędów o ich braku, bez konieczności przeprowadzania komunikacji z serwerem.

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

"Za kulisami", widżet `ActiveForm` odczyta wszystkie zasady walidacji zadeklarowane w modelu i wygeneruje odpowiedni kod JavaScript dla walidatorów wspierających walidację po stronie klienta. Kiedy użytkownik zmieni wartość w polu lub spróbuje wysłać formularz, zostanie wywołana walidacja po stronie klienta.

Jeśli chcesz wyłączyć całkowicie walidację po stronie klienta, możesz ustawić właściwość `enableClientValidation` na `false`. Możesz również wyłączyć ten rodzaj walidacji dla konkretnego pola, przez ustawienie jego właściwości `enableClientValidation` na `false`. Jeśli właściwość `enableClientValidation` zostanie skonfigurowana na poziomie pola formularza i w samym formularzu jednocześnie, pierwszeństwo będzie miała opcja określona w formularzu.

> **Informacja:** Od wersji 2.0.11 wszystkie walidatory rozszerzające klasę `yii\validators\Validator` używają opcji klienta przekazywanych z oddzielnej metody - `yii\validators\Validator::getClientOptions()`. Możesz jej użyć:
>
> - jeśli chcesz zaimplementować swoją własną walidację po stronie klienta, ale pozostawić synchronizację z opcjami walidatora po stronie serwera;
> - do rozszerzenia lub zmodyfikowania dla uzyskania specjalnych korzyści:

```
public function getClientOptions($model, $attribute)
{
    $options = parent::getClientOptions($model, $attribute);
    // Zmodyfikuj $options w tym miejscu

    return $options;
}
```

Implementacja walidacji po stronie klienta

Aby utworzyć walidator wspierający walidację po stronie klienta, powinieneś zaimplementować metodę `clientValidateAttribute()`, która zwraca kod JavaScript, odpowiedzialny za przeprowadzenie walidacji. W kodzie JavaScript możesz użyć następujących predefiniowanych zmiennych:

- `attribute`: nazwa atrybutu podlegającego walidacji.
- `value`: wartość atrybutu podlegająca walidacji.
- `messages`: tablica używana do przechowywania wiadomości błędów dla danego atrybutu.
- `deferred`: tablica, do której można dodać zakolejkowane obiekty (wyjaśnione w późniejszej podsekcji).

W poniższym przykładzie, tworzymy walidator `StatusValidator`, który sprawdza, czy wartość danego atrybutu jest wartością znajdującą się na liście statusów w bazie danych. Walidator wspiera obydwa typy walidacji; po stronie klienta oraz serwerową.

```
namespace app\components;

use yii\validators\Validator;
use app\models\Status;

class StatusValidator extends Validator
{
    public function init()
    {
        parent::init();
        $this->message = 'Niepoprawna wartość pola status.';
    }

    public function validateAttribute($model, $attribute)
    {
        $value = $model->$attribute;
        if (!Status::find()->where(['id' => $value])->exists()) {
            $model->addError($attribute, $this->message);
        }
    }

    public function clientValidateAttribute($model, $attribute, $view)
    {
        $statuses =
        json_encode(Status::find()->select('id')->asArray()->column());
        $message = json_encode($this->message, JSON_UNESCAPED_SLASHES |
        JSON_UNESCAPED_UNICODE);
        return <<<JS
if ($.inArray(value, $statuses) === -1) {
    messages.push($message);
}
JS;
    }
}
```

Wskazówka: Powyższy kod został podany głównie do zademonstrowania, jak wspierać walidację po stronie klienta. W praktyce można użyć podstawowego walidatora in, aby osiągnąć ten sam cel. Możesz napisać taką zasadę walidacji następująco:

```
[
    ['status', 'in', 'range' =>
    Status::find()->select('id')->asArray()->column()],
]
```

Wskazówka: Jeśli musisz dodać ręcznie walidację po stronie klienta np. podczas dynamicznego dodawania pól formularza lub przeprowadzania specjalnej logiki w obrębie interfejsu użytkownika, zapoznaj się z rozdziałem Praca z ActiveForm za pomocą JavaScript[4] w Yii 2.0 Cookbook.

Kolejkowa walidacja

Jeśli potrzebujesz przeprowadzić asynchroniczną walidację po stronie klienta, możesz utworzyć obiekt kolejkujący[5]. Dla przykładu, aby przeprowadzić niestandardową walidację AJAX, możesz użyć następującego kodu:

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.push($.get("/check", {value: value}).done(function(data) {
            if ('' !== data) {
                messages.push(data);
            }
        }));
JS;
}
```

W powyższym kodzie, zmienna `deferred` jest dostarczoną przez Yii tablicą zakolejkowanych obiektów. Metoda jQuery `$.get()` tworzy obiekt kolejkowy, który jest dodawany do tablicy `deferred`.

Możesz także utworzyć osobny obiekt kolejkowania i wywołać jego metodę `resolve()` po otrzymaniu asynchronicznej informacji zwrotnej. Poniższy przykład pokazuje, jak zwalidować wymiary przesłanego obrazka po stronie klienta.

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        var def = $.Deferred();
        var img = new Image();
```

[4] https://github.com/samdark/yii2-cookbook/blob/master/book/forms-activeform-js.md

[5] https://api.jquery.com/category/deferred-object/

```
        img.onload = function() {
            if (this.width > 150) {
                messages.push('Image too wide!!');
            }
            def.resolve();
        }
        var reader = new FileReader();
        reader.onloadend = function() {
            img.src = reader.result;
        }
        reader.readAsDataURL(file);

        deferred.push(def);
JS;
}
```

> **Uwaga:** Metoda `resolve()` musi być wywołana po walidacji atrybutu. W przeciwnym razie główna walidacja formularza nie zostanie ukończona.

Dla uproszczenia, tablica `deferred` jest wyposażona w skrótową metodę `add()`, która automatycznie tworzy obiekt kolejkowy i dodaje go do tej tablicy. Używając tej metody, możesz uprościć powyższy przykład:

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.add(function(def) {
            var img = new Image();
            img.onload = function() {
                if (this.width > 150) {
                    messages.push('Image too wide!!');
                }
                def.resolve();
            }
            var reader = new FileReader();
            reader.onloadend = function() {
                img.src = reader.result;
            }
            reader.readAsDataURL(file);
        });
JS;
}
```

Walidacja przy użyciu AJAX

Niektóre walidacje mogą zostać wykonane tylko po stronie serwera, ponieważ tylko serwer posiada niezbędne informacje do ich przeprowadzenia. Dla przykładu, aby sprawdzić, czy login został już zajęty, musimy sprawdzić tabelę użytkowników w bazie danych. W tym właśnie przypadku możesz

użyć walidacji AJAX. Wywoła ona żądanie AJAX w tle, aby spradzić to pole.

Aby uaktywnić walidację AJAX dla pojedyńczego pola formularza, ustaw właściwość `enableAjaxValidation` na `true` oraz zdefiniuj unikalne `id` formularza:

```
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'registration-form',
]);

echo $form->field($model, 'username', ['enableAjaxValidation' => true]);

// ...

ActiveForm::end();
```

Aby uaktywnić walidację AJAX dla całego formularza, ustaw właściwość `enableAjaxValidation` na `true` na poziomie formularza:

```
$form = ActiveForm::begin([
    'id' => 'contact-form',
    'enableAjaxValidation' => true,
]);
```

> **Uwaga:** Jeśli właściwość `enableAjaxValidation` zostanie skonfigurowana na poziomie pola formularza i jednocześnie w samym formularzu, pierwszeństwo będzie miała opcja określona w formularzu.

Musisz również przygotować serwer na obsłużenie AJAXowego zapytanie o walidację. Możesz to osiągnąć przez następujący fragment kodu w akcji kontrolera:

```
if (Yii::$app->request->isAjax && $model->load(Yii::$app->request->post()))
{
    Yii::$app->response->format = Response::FORMAT_JSON;
    return ActiveForm::validate($model);
}
```

Powyższy kod sprawdzi, czy zapytanie zostało wysłane przy użyciu AJAXa. Jeśli tak, w odpowiedzi zwróci wynik walidacji w formacie JSON.

> **Informacja:** Możesz również użyć walidacji kolejkowej do wykonania walidacji AJAX, jednakże walidacja AJAXowa opisana w tej sekcji jest bardziej systematyczna i wymaga mniej wysiłku przy kodowaniu.

Kiedy zarówno `enableClientValidation`, jak i `enableAjaxValidation` ustawione są na `true`, walidacja za pomocą AJAX zostanie uruchomiona dopiero po udanej walidacji po stronie klienta.

7.3 Wysyłanie plików

Przesyłanie plików w Yii jest zazwyczaj wykonywane przy użyciu klasy `UploadedFile`, która hermetyzuje każdy przesłany plik jako obiekt `UploadedFile`. W połączeniu z `ActiveForm` oraz modelem, możesz w łatwy sposób zaimplementować bezpieczny mechanizm przesyłania plików.

7.3.1 Tworzenie modeli

Tak jak przy zwykłych polach tekstowych, aby przesłać pojedyńczy plik musisz utworzyć klasę modelu oraz użyć atrybutu tego modelu do przechowania instancji przesłanego pliku. Powinieneś również zadeklarować zasadę walidacji do zwalidowania przesłanego pliku. Dla przykładu:

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile
     */
    public $imageFile;

    public function rules()
    {
        return [
            [['imageFile'], 'file', 'skipOnEmpty' => false, 'extensions' =>
            'png, jpg'],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            $this->imageFile->saveAs('uploads/' . $this->imageFile->baseName
            . '.' . $this->imageFile->extension);
            return true;
        } else {
            return false;
        }
    }
}
```

W powyższym kodzie, atrybut `imageFile` zostanie użyty do przechowania instancji przesłanego pliku. Jest połączony z zasadą walidacji `file`, która korzysta z walidatora `FileValidator`, aby upewnić się, że przesłany plik posiada rozszerzenie `png` lub `jpg`. Metoda `upload()` wywoła walidację oraz zapis przesłanego pliku na serwerze.

Walidator `file` pozwala na sprawdzenie rozszerzenia, wielkości, typu MIME, itp. Po więcej szczegółów zajrzyj do sekcji Podstawowe walidatory

> **Wskazówka:** Jeśli przesyłasz obrazek, możesz rozważyć użycie walidatora `image`. Walidator ten jest implementowany przez `ImageValidator`, który weryfikuje czy atrybut otrzymał prawidłowy obrazek który może być zapisany i przetworzony przez rozszerzenie Imagine[6].

7.3.2 Renderowanie pola wyboru pliku

Po zapisaniu modelu, utwórz pole wyboru pliku w widoku:

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' =>
'multipart/form-data']]) ?>

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

    <button>Wyślij</button>

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

Należy pamiętać, aby dodać opcję `enctype` do formularza, przez co plik będzie mógł być prawidłowo przesłany. Wywołanie `fileInput()` spowoduje wyrenderowanie tagu `<input type="file">`, który pozwala użytkownikowi na wybranie oraz przesłanie pliku.

> **Wskazówka:** od wersji 2.0.8, `fileInput` dodaje automatycznie opcję `enctype` do formularza, kiedy pole typu 'file input' jest używane.

7.3.3 Implementacja kontrolera

W akcji kontrolera musimy połączyć model oraz widok aby zaimplementować przesyłanie plików:

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
```

[6] https://github.com/yiisoft/yii2-imagine

```
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFile = UploadedFile::getInstance($model,
            'imageFile');
            if ($model->upload()) {
                // plik zosta␣l przes␣lany
                return;
            }
        }

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

W powyższym kodzie, kiedy formularz jest wysłany, metoda `getInstance()` wywoływana jest do reprezentowania pliku jako instancji `UploadedFile`. Następnie przystępujemy do walidacji modelu, aby upewnić się, że przesłany plik jest prawidłowy, po czym zapisujemy go na serwerze.

7.3.4 Przesyłanie wielu plików

Możesz przesyłać wiele plików za jednym razem, modyfikując odrobinę kod wylistowany w powyższych sekcjach.

Najpierw powinieneś dostosować klasę modelu dodając opcję `maxFiles` do zasady walidacji `file`, aby określić dozwoloną maksymalną liczbę przesyłanych plików. Metoda `upload()` powinna również zostać zaktualizowana, aby zapisywać pliki jeden po drugim.

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile[]
     */
    public $imageFiles;

    public function rules()
    {
        return [
            [['imageFiles'], 'file', 'skipOnEmpty' => false, 'extensions' =>
            'png, jpg', 'maxFiles' => 4],
        ];
    }
```

```
    public function upload()
    {
        if ($this->validate()) {
            foreach ($this->imageFiles as $file) {
                $file->saveAs('uploads/' . $file->baseName . '.' .
                $file->extension);
            }
            return true;
        } else {
            return false;
        }
    }
}
```

W pliku widoku, powinieneś dodać opcję `multiple` do wywołania `fileInput()`, aby pole wyboru pliku pozwalało na wybór wielu plików na raz:

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' =>
'multipart/form-data']]) ?>

    <?= $form->field($model, 'imageFiles[]')->fileInput(['multiple' =>
true, 'accept' => 'image/*']) ?>

    <button>Submit</button>

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

Na koniec, w akcji kontrolera musimy zmienić wywołanie `UploadedFile::getInstance()` na `UploadedFile::getInstances()`, aby przypisać tablicę instancji `UploadedFile` do `UploadForm::imageFiles`.

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFiles = UploadedFile::getInstances($model,
            'imageFiles');
            if ($model->upload()) {
                // plik został przesłany
```

```
                return;
            }
        }

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

7.4 Odczytywanie tablicowych danych wejściowych

Czasami zachodzi potrzeba obsłużenia wielu modeli tego samego rodzaju w jednym formularzu. Dla przykładu - ustawienia, gdzie każde z nich jest przechowywane jako para klucz-wartość i każde z nich jest reprezentowane przez model `Setting` active record. Dla kontrastu obsługa wielu modeli różnych rodzajów pokazana jest w sekcji Pobieranie danych dla wielu modeli.

Poniższe przykłady pokazują jak zaimplementować tablicowe dane wejściowe w Yii.

Występują trzy różne sytuacje, które należy obsłużyć inaczej:

- Aktualizacja określonej liczby rekordów z bazy danych
- Dynamiczne tworzenie nowych rekordów
- Aktualizacja, tworzenie oraz usuwanie rekordów na jednej stronie

W porównaniu do formularza z pojedyńczym modelem, wytłumaczonym poprzednio, pracujemy teraz na tablicy modeli. Tablica przekazywana jest do widoku, aby wyświetlić pola wejściowe dla każdego modelu w stylu tabeli, użyjemy do tego metod pomocniczych z `Model`, które pozwalają na wczytywanie oraz walidację wielu modeli na raz:

- `loadMultiple()` wczytuje dane z tablicy POST do tablicy modeli.
- `validateMultiple()` waliduje tablicę modeli.

Aktualizacja określonej liczby rekordów

Zacznijmy od akcji kontrolera:

```
<?php

namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use app\models\Setting;

class SettingsController extends Controller
{
    // ...

    public function actionUpdate()
```

```
    {
        $settings = Setting::find()->indexBy('id')->all();

        if (Model::loadMultiple($settings, Yii::$app->request->post()) &&
        Model::validateMultiple($settings)) {
            foreach ($settings as $setting) {
                $setting->save(false);
            }
            return $this->redirect('index');
        }

        return $this->render('update', ['settings' => $settings]);
    }
}
```

W powyższym kodzie używamy `indexBy()` podczas pobierania danych z bazy danych aby zasilić tablicę modelami zaindeksowanymi przez główny klucz. Będzie to później użyte do zidentyfikowania pól formularza. `loadMultiple()` uzupełnia modele danymi formularza przesłanymi metodą POST a następnie metoda `validateMultiple()` waliduje te modele. Po walidacji przekazujemy parametr `false` do metody `save()`, aby nie uruchamiać walidacji ponownie.

Przejdziemy teraz do formularza w widoku `update`:

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

$form = ActiveForm::begin();

foreach ($settings as $index => $setting) {
    echo $form->field($setting, "[$index]value")->label($setting->name);
}

ActiveForm::end();
```

Dla każdego z ustawień renderujemy nazwę oraz pole wejściowe z wartością. Bardzo ważne jest dodanie odpowiedniego indeksu do nazwy pola, ponieważ dzięki temu metoda `loadMultiple()` określa który model powinna uzupełnić przekazanymi wartościami.

Dynamiczne tworzenie nowych rekordów

Tworzenie nowych rekordów jest podobne do ich aktualizacji, poza częścią, w której instancjujemy modele:

```
public function actionCreate()
{
    $count = count(Yii::$app->request->post('Setting', []));
    $settings = [new Setting()];
    for($i = 1; $i < $count; $i++) {
        $settings[] = new Setting();
```

```
    }

    // ...
}
```

Tworzymy tutaj początkową tablicę `$settings` zawierającą domyślnie jeden model, dlatego zawsze co najmniej jedno pole będzie widoczne w widoku. Dodatkowo dodajemy więcej modeli dla każdej linii pól wejściowych jakie otrzymaliśmy.

W widoku możemy użyć kodu JavaScript do dynamicznego dodawania nowych linii pól wejściowych.

Aktualizacja, tworzenie oraz usuwanie rekordów na jednej stronie

> **Uwaga:** Ta sekcja nie została jeszcze skończona.

TBD

7.5 Pobieranie danych dla wielu modeli

Kiedy mamy do czynienia ze skomplikowanym zestawem danych, jest możliwe, że trzeba będzie użyć wielu różnych modeli, aby pobrać te dane od użytkownika. Dla przykładu - zakładając, że dane logowania użytkownika zapisane są w tabeli `user`, podczas gdy dane profilu użytkownika są przechowywane w tabeli `profile`, będziesz chciał pobrać dane od użytkownika za pomocą modeli `User` oraz `Profile`. Dzięki wsparciu modeli i formularzy przez Yii, możesz rozwiązać ten problem w sposób nie różniący się za bardzo od przetwarzania pojedynczego modelu.

W poniższym przykładzie pokażemy jak utworzyć formularz, który pozwoli Ci na zbieranie danych dla obydwu modeli: `User` oraz `Profile`.

Na początek, akcja w kontrolerze do zbierania danych użytkownika oraz danych profilowych może zostać napisana następująco:

```
namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use app\models\User;
use app\models\Profile;

class UserController extends Controller
{
    public function actionUpdate($id)
    {
        $user = User::findOne($id);
        if (!$user) {
```

```
            throw new NotFoundHttpException("The user was not found.");
        }

        $profile = Profile::findOne($user->profile_id);

        if (!$profile) {
            throw new NotFoundHttpException("The user has no profile.");
        }

        $user->scenario = 'update';
        $profile->scenario = 'update';

        if ($user->load(Yii::$app->request->post()) &&
        $profile->load(Yii::$app->request->post())) {
            $isValid = $user->validate();
            $isValid = $profile->validate() && $isValid;
            if ($isValid) {
                $user->save(false);
                $profile->save(false);
                return $this->redirect(['user/view', 'id' => $id]);
            }
        }

        return $this->render('update', [
            'user' => $user,
            'profile' => $profile,
        ]);
    }
}
```

W akcji `update`, najpierw ładujemy modele `$user` oraz `$profile`, które zostaną zaktualizowane danymi z bazy. Następnie wywołujemy metodę `load()`, aby wypełnić te dwa modele danymi wprowadzonymi przez użytkownika. Na końcu modele zostają poddane walidacji i, jeśli wszystko jest w porządku, zapisane. W przeciwnym razie zostanie wyrenderowany widok `update`, który zawiera następujący kod:

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

$form = ActiveForm::begin([
    'id' => 'user-update-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($user, 'username') ?>

    ...other input fields...

    <?= $form->field($profile, 'website') ?>

    <?= Html::submitButton('Update', ['class' => 'btn btn-primary']) ?>
<?php ActiveForm::end() ?>
```

Jak widzisz, w widoku `update` tworzymy pola tekstowe używając dwóch modeli: `$user` oraz `$profile`.

7.6 Rozszerzanie ActiveForm po stronie klienta

Widżet `yii\widgets\ActiveForm` posiada szereg wbudowanych metod JavaScript, służących do walidacji po stronie klienta. Ich implementacja jest bardzo elastyczna i pozwala na rozszerzanie ich na wiele sposobów.

7.6.1 Zdarzenia ActiveForm

ActiveForm wyzwala serie dedykowanych zdarzeń. Używając poniższego kodu, można przechwycić te zdarzenia i je obsłużyć:

```
$('#contact-form').on('beforeSubmit', function (e) {
        if (!confirm("Wszystko jest w porządku. Wysłać formularz?")) {
                return false;
        }
        return true;
});
```

Poniżej znajdziesz opis dostępnych zdarzeń.

`beforeValidate` jest wyzwalane przed walidacją całego formularza.

Sygnatura metody obsługującej to zdarzenie powinna wyglądać następująco:

```
function (event, messages, deferreds)
```

gdzie

- `event`: obiekt Event.
- `messages`: asocjacyjna tablica, gdzie kluczami są ID atrybutów, a wartościami tablice opisów błędów dla tych atrybutów.
- `deferreds`: tablica obiektów kolejkujących. Możesz użyć `deferreds.add(callback)`, aby dodać nową walidację do kolejki.

Jeśli metoda obsługująca zwróci boolean `false`, zatrzyma dalszą walidację formularza. W takim wypadku zdarzenie `afterValidate` nie będzie już wyzwalane.

`afterValidate` jest wyzwalane po walidacji całego formularza.

Sygnatura metody obsługującej to zdarzenie powinna wyglądać następująco:

```
function (event, messages, errorAttributes)
```

gdzie

- `event`: obiekt Event.
- `messages`: asocjacyjna tablica, gdzie kluczami są ID atrybutów, a wartościami tablice opisów błędów dla tych atrybutów.
- `errorAttributes`: tablica atrybutów z błędami walidacji. Sprawdź konstrukcję `attributeDefaults`, aby dowiedzieć się więcej o strukturze tego parametru.

`beforeValidateAttribute` jest wyzwalane przed walidacją atrybutu.

Sygnatura metody obsługującej to zdarzenie powinna wyglądać następująco:

```
function (event, attribute, messages, deferreds)
```

gdzie

- `event`: obiekt Event.
- `attribute`: atrybut poddawany walidacji. Sprawdź konstrukcję `attributeDefaults`, aby dowiedzieć się więcej o strukturze tego parametru.
- `messages`: tablica, do której możesz dodać opisy błędów walidacji dla wybranego atrybutu.
- `deferreds`: tablica obiektów kolejki. Możesz użyć `deferreds.add(callback)`, aby dodać nową walidację do kolejki.

Jeśli metoda obsługująca zwróci boolean `false`, zatrzyma dalszą walidację wybranego atrybutu. W takim wypadku zdarzenie `afterValidateAttribute` nie będzie już wyzwalane.

`afterValidateAttribute` jest wyzwalane po walidacji całego formularza i każdego atrybutu.

Sygnatura metody obsługującej to zdarzenie powinna wyglądać następująco:

```
function (event, attribute, messages)
```

gdzie

- `event`: obiekt Event.
- `attribute`: atrybut poddawany walidacji. Sprawdź konstrukcję `attributeDefaults`, aby dowiedzieć się więcej o strukturze tego parametru.
- `messages`: tablica, do której możesz dodać opisy błędów walidacji dla wybranego atrybutu.

`beforeSubmit` jest wyzwalane przed wysłaniem formularza, po pomyślnej walidacji.

Sygnatura metody obsługującej to zdarzenie powinna wyglądać następująco:

```
function (event)
```

gdzie event jest obiektem Event.

Jeśli metoda obsługująca zwróci boolean `false`, zatrzyma wysyłanie formularza.

`ajaxBeforeSend` jest wyzwalane przed wysłaniem żądania AJAX w przypadku walidacji AJAX-owej.

Sygnatura metody obsługującej to zdarzenie powinna wyglądać następująco:

```
function (event, jqXHR, settings)
```

gdzie

- `event`: obiekt Event.
- `jqXHR`: obiekt jqXHR.
- `settings`: konfiguracja żądania AJAX.

`ajaxComplete` jest wyzwalane po ukończeniu żądania AJAX w przypadku walidacji AJAX-owej.

Sygnatura metody obsługującej to zdarzenie powinna wyglądać następująco:

```
function (event, jqXHR, textStatus)
```

gdzie

- `event`: obiekt Event.
- `jqXHR`: obiekt jqXHR.
- `textStatus`: status żądania (śuccess", "notmodified", "error", "timeout", "abort" lub "parsererror").

7.6.2 Wysyłanie formularza za pomocą AJAX

Walidacja może być przeprowadzona po stronie klienta lub za pomocą AJAX-a, ale wysyłanie formularza jest domyślnie przeprowadzane za pomocą zwyczajnego żądania. Jeśli chcesz przesłać formularz za pomocą AJAX, możesz to zrobić obsługując zdarzenie `beforeSubmit` formularza w następujący sposób:

```
var $form = $('#formId');
$form.on('beforeSubmit', function() {
    var data = $form.serialize();
    $.ajax({
        url: $form.attr('action'),
        type: 'POST',
        data: data,
        success: function (data) {
```

```
            // Implementacja pomyślnego statusu
        },
        error: function(jqXHR, errMsg) {
            alert(errMsg);
        }
    });
    return false; // powstrzymuje przed domyślnym sposobem wys␣lania
});
```

Aby dowiedzieć się więcej o funkcji jQuery `ajax()`, zapoznaj się z dokumentacją jQuery[7].

7.6.3 Dynamiczne dodawanie pól

We współczesnych aplikacjach webowych często koniecznie jest modyfikowanie formularza już po tym, jak został zaprezentowany użytkownikowi. Dla przykładu może to być dodawanie nowego pola po kliknięciu w ikonę "z plusem". Aby uruchomić walidację takich pól, należy je zarejestrować za pomocą JavaScriptowego pluginu ActiveForm.

Po dodaniu pola do formularza, należy dołączyć je również do listy walidacji:

```
$('#contact-form').yiiActiveForm('add', {
    id: 'address',
    name: 'address',
    container: '.field-address',
    input: '#address',
    error: '.help-block',
    validate:  function (attribute, value, messages, deferred, $form) {
        yii.validation.required(value, messages, {message: "Informacja
        dotycząca walidacji tutaj"});
    }
});
```

Aby usunąć pole z listy walidacji (aby nie było już sprawdzane), możesz wykonać następujący kod:

```
$('#contact-form').yiiActiveForm('remove', 'address');
```

[7] https://api.jquery.com/jQuery.ajax/

Rozdział 8

Wyświetlanie danych

Error: not existing file: output-formatting.md

8.1 Paginacja

Kiedy danych jest zbyt dużo, aby wyświetlić je w całości na jednej stronie, zwykle stosuje się mechanizm podziału na wiele stron, z których każda prezentuje tylko część danych na raz. Mechanizm ten nazywamy *paginacją*.

W Yii obiekt `Pagination` reprezentuje zbiór informacji o schemacie paginacji.

- `liczba wyników` określa całkowitą liczbę elementów zestawu danych. Zwykle jest to znacznie większa liczba niż ilość elementów, które można umieścić na pojedynczej stronie.
- `rozmiar strony` określa jak wiele elementów może znaleźć się na pojedynczej stronie. Domyślna wartość to 20.
- `aktualna strona` wskazuje numer aktualnie wyświetlanej strony (począwszy od zera). Domyślna wartość to 0, wskazująca na pierwszą stronę.

Korzystając z w pełni zdefiniowanego obiektu `Pagination`, można pobrać i wyświetlić dane w partiach. Dla przykładu, przy pobieraniu danych z bazy można użyć wartości `OFFSET` i `LIMIT` w kwerendzie, które będą odpowiadać tym zdefiniowanym przez paginację.

```
use yii\data\Pagination;

// stwórz kwerendę bazodanową, aby pobrać wszystkie artyku␣ly o statusie =
1
$query = Article::find()->where(['status' => 1]);

// ustal ca␣lkowitą liczbę artyku␣lów (ale nie pobieraj jeszcze danych
artyku␣lów)
$count = $query->count();

// stwórz obiekt paginacji z ca␣lkowitą liczbą wyników
$pagination = new Pagination(['totalCount' => $count]);

// ogranicz wyniki kwerendy korzystając z paginacji i pobierz artyku␣ly
$articles = $query->offset($pagination->offset)
    ->limit($pagination->limit)
    ->all();
```

Która strona wyników z artykułami zostanie pobrana w powyższym przykładzie? To zależy od tego, czy ustawiono parametr kwerendy `page`. Domyślnie paginacja próbuje ustawić `aktualną stronę` na odpowiadającą wartości parametru `page`. Jeśli ten parametr nie jest przekazany, wartość będzie domyślna, czyli 0.

Aby ułatwić tworzenie elementów UI, które będą odpowiedzialne za korzystanie z mechanizmu paginacji, Yii posiada wbudowany widżet `LinkPager`, który wyświetla listę przycisków z numerami, po kliknięciu których użytkownik przechodzi do pożądanej strony wyników. Widżet korzysta z obiektu paginacji, dzięki czemu wie, który numer ma aktualnie wyświetlana strona i jak wiele przycisków stron powinien wyświetlić. Przykład:

```
use yii\widgets\LinkPager;

echo LinkPager::widget([
    'pagination' => $pagination,
]);
```

Jeśli chcesz samemu stworzyć takie elementy UI, możesz skorzystać z metody `createUrl()`, aby uzyskać adresy URL poszczególnych stron. Metoda ta wymaga podania parametru page i zwraca poprawnie sformatowany adres URL zawierający ten parametr. Przykładowo,

```
// określa trasę, która będzie zawarta w nowoutworzonym adresie URL
// Jeśli nie będzie podana, użyta zostanie aktualna trasa
$pagination->route = 'article/index';

// wyświetla: /index.php?r=article%2Findex&page=100
echo $pagination->createUrl(100);

// wyświetla: /index.php?r=article%2Findex&page=101
echo $pagination->createUrl(101);
```

> **Wskazówka:** Możesz zmodyfikować nazwę parametru kwerendy `page`, poprzez ustawienie właściwości `pageParam` w czasie tworzenia obiektu paginacji.

Error: not existing file: output-sorting.md

Error: not existing file: output-data-providers.md

Error: not existing file: output-data-widgets.md

8.2 Praca ze skryptami

Uwaga: Ta sekcja nie została jeszcze ukończona.

Rejestrowanie skryptów

Dzięki obiektowi `View` możesz rejestrować skrypty w aplikacji. Przeznaczone są do tego dwie dedykowane metody: `registerJs()` dla skryptów wbudowanych oraz `registerJsFile()` dla skryptów zewnętrznych. Skrypty wbudowane są przydatne przy konfiguracji oraz dynamicznym generowaniu kodu. Możesz dodać je w następujący sposób:

```
$this->registerJs("var options = " . json_encode($options) . ";",
View::POS_END, 'my-options');
```

Pierwszy argument przekazywany do metody `registerJs` to kod JavaScript, który chcemy umieścić na stronie. Jako drugi argument wskazujemy miejsce, w którym skrypt ma zostać umieszczony na stronie. Możliwe wartości to:

- `POS_HEAD` dla sekcji `head`.
- `POS_BEGIN` zaraz po otwarciu tagu `<body>`.
- `POS_END` zaraz przed zamknięciem tagu `</body>`.
- `POS_READY` do wywołania kodu z użyciem zdarzenia `ready` na dokumencie. Ta opcja zarejestruje automatycznie `jQuery`
- `POS_LOAD` do wywołania kodu z użyciem zdarzenia `load` na dokumencie. Ta opcja zarejestruje automatycznie `jQuery`

Ostatnim argumentem jest unikalne ID skryptu, które jest używane do zidentyfikowania bloku kodu i zastąpienia go, jeśli taki został już zarejestrowany. Jeśli ten argument nie zostanie podany, kod JavaScript zostanie użyty jako ID.

Skrypt zewnętrzny może zostać dodany następująco:

```
$this->registerJsFile('https://example.com/js/main.js', ['depends' =>
[\yii\web\JqueryAsset::class]]);
```

Argumenty dla metod `registerCssFile()` są podobne do `registerJsFile()`. W powyższym przykładzie, zarejestrowaliśmy plik `main.js` z zależnością od `JqueryAsset`. Oznacza to, że plik `main.js` zostanie dodany PO pliku `jquery.js`. Bez określenia tej zależności, względny porządek pomiędzy `main.js` a `jquery.js` nie zostałby zachowany.

Tak jak i w przypadku `registerCssFile()`, mocno rekomendujemy, abyś użył assetów do zarejestrowania zewnętrznych plików JS zamiast używania `registerJsFile()`.

Rejestracja assetów

Jak zostało wspomniane wcześniej, korzystniejsze jest stosowanie assetów, zamiast kodu CSS i JS bezpośrednio (po informacje na ten temat sięgnij do

sekcji menedżera assetów). Korzystanie z już zdefiniowanych pakietów jest bardzo proste:

```
\frontend\assets\AppAsset::register($this);
```

Rejestrowanie kodu CSS

Możesz zarejestrować kod CSS przy użyciu metody `registerCss()` lub `registerCssFile()`. Pierwsza z nich rejestruje blok kodu CSS, natomiast druga zewnętrzny plik `.css`. Dla przykładu:

```
$this->registerCss("body { background: #f00; }");
```

Powyższy kod doda kod CSS do sekcji `head` strony:

```
<style>
body { background: #f00; }
</style>
```

Jeśli chcesz określić dodatkowe właściwości dla tagu `style`, przekaż tablicę `nazwa => wartość` jako drugi argument. Jeśli chcesz się upewnić, że jest tylko jeden tag `style`, użyj trzeciego argumentu, tak jak zostało to opisane dla meta tagów.

```
$this->registerCssFile("https://example.com/css/themes/black-and-white.css",
[
    'depends' => [BootstrapAsset::class],
    'media' => 'print',
], 'css-print-theme');
```

Kod powyżej doda link w sekcji `head` strony do pliku CSS.

- Pierwszy argument określa, który plik ma zostać zarejestrowany,
- Drugi argument określa atrybuty tagu `<link>`. Opcja `depends` jest obsługiwana w specjalny sposób, od niej zależy położenie pliku CSS. W tym przypadku, plik link do pliku CSS zostanie umieszony ZA plikami CSS w `yii\bootstrap\BootstrapAsset`,
- Ostatni argument określa ID identyfikujące ten plik CSS. W przypadku jego braku, zostanie użyty do tego celu adres URL pliku CSS.

Jest mocno wskazane używanie assetów do rejestrowania zewnętrznych plików CSS. Użycie ich pozwala Ci na łączenie i kompresowanie wielu plików CSS, które jest wręcz niezbędne na stronach internetowych o dużym natężeniu ruchu.

Error: not existing file: output-theming.md

Rozdział 9

Bezpieczeństwo

9.1 Bezpieczeństwo

Spełnienie wymogów bezpieczeństwa jest podstawą sukcesu i długiej żywotności każdej aplikacji. Niestety, wielu deweloperów idzie w tym temacie drogą na skróty, czy to z powodu braku jego zrozumienia, czy też kłopotów z implementacją we własnym kodzie. Aby uczynić Twoją aplikację opartą na Yii tak bezpieczną, jak to tylko możliwe, Yii zapewnia kilka świetnych i jednocześnie łatwych w użyciu funkcjonalności służących poprawie bezpieczeństwa.

- Uwierzytelnianie
- Autoryzacja
- Praca z hasłami
- Kryptografia
- Bezpieczeństwo widoków
- Klienty autoryzacji[1]
- Bezpieczeństwo w praktyce

[1] https://github.com/yiisoft/yii2-authclient/blob/master/docs/guide/README.md

Error: not existing file: security-authentication.md

Error: not existing file: security-authorization.md

Error: not existing file: security-passwords.md

Error: not existing file: security-cryptography.md

Error: not existing file: security-best-practices.md

Rozdział 10

Pamięć podręczna

10.1 Pamięć podręczna

Mechanizmy wykorzystujące pamięć podręczną pozwalają na poprawienie wydajności aplikacji sieciowej w tani i efektywny sposób. Zapisanie statycznych danych w pamięci podręcznej, zamiast generowania ich od podstaw przy każdym wywołaniu, pozwala na znaczne zaoszczędzenie czasu odpowiedzi aplikacji.

Zapis pamięci podręcznej może odbywać się na wielu poziomach i w wielu miejscach aplikacji. Po stronie serwera, na niskim poziomie, można wykorzystać pamięć podręczną do zapisania podstawowych danych, takich jak zbiór informacji o najnowszych artykułach pobieranych z bazy danych. Na wyższym poziomie, pamięci podręcznej można użyć do przechowania części bądź całości strony www, na przykład w postaci rezultatu wyrenderowania listy ww. najświeższych artykułów. Po stronie klienta, pamięć podręczna HTTP przeglądarki może zapisać zawartość ostatnio odwiedzonej strony.

Yii wpiera wszystkie te mechanizmy zapisu w pamięci podręcznej:

- Pamięć podręczna danych
- Pamięć podręczna fragmentów
- Pamięć podręczna stron
- Pamięć podręczna HTTP

Error: not existing file: caching-data.md

10.2 Pamięć podręczna fragmentów

Pamięć podręczna fragmentów dotyczy zapisywania w pamięci podręcznej części strony Web. Dla przykładu, jeśli strona wyświetla podsumowanie danych rocznej sprzedaży w postaci tabeli, można tę tabelę zapisać w pamięci podręcznej, aby wyeliminować konieczność generowania jej za każdym razem od nowa. Mechanizm pamięci podręcznej fragmentów zbudowany jest w oparciu o pamięć podręczną danych.

Aby wykorzystać pamięć podręczną fragmentów, należy użyć następującego kodu w widoku:

```
if ($this->beginCache($id)) {

    // ... generowanie zawartości w tym miejscu ...

    $this->endCache();
}
```

Jak widać, chodzi tu o zamknięcie bloku generatora zawartości pomiędzy wywołaniem metod `beginCache()` i `endCache()`. Jeśli wskazana zawartość zostanie odnaleziona w pamięci podręcznej, `beginCache()` wyrenderuje zapisaną zawartość i zwróci `false`, przez co pominie blok jej generowania. W przeciwnym wypadku generowanie zostanie uruchomione, a w momencie wywołania `endCache()` wygenerowana zawartość zostanie zapisana w pamięci podręcznej.

Tak, jak w przypadku pamięci podręcznej danych, unikalne `$id` jest wymagane do identyfikacji zawartości.

10.2.1 Opcje zapisu w pamięci podręcznej

Możesz określić dodatkowe opcje zapisu pamięci podręcznej fragmentów, przekazując tablicę opcji jako drugi parametr w metodzie `beginCache()`. Opcje te będą użyte do skonfigurowania widżetu `FragmentCache`, który implementuje właściwą funkcjonalność zapisu pamięci podręcznej.

Czas życia

Prawdopodobnie najczęściej używaną opcją zapisu fragmentów jest `duration`. Parametr ten określa, przez ile sekund zawartość może być przechowywana w pamięci podręcznej, zanim konieczne będzie wygenerowanie jej ponownie. Poniższy kod zapisuje fragment zawartości w pamięci podręcznej na maksymalnie godzinę:

```
if ($this->beginCache($id, ['duration' => 3600])) {

    // ... generowanie zawartości w tym miejscu ...

    $this->endCache();
}
```

Jeśli ta opcja nie jest określona, przyjmuje domyślną wartość 60, co oznacza, że ważność zapisanej zawartości wygaśnie po upływie 60 sekund.

Zależności

Tak, jak w przypadku pamięci podręcznej danych, zapis fragmentów może opierać się na zależnościach. Dla przykładu, zawartość wyświetlanego posta zależy od tego, czy został on zmodyfikowany, bądź nie.

Aby określić zależność, należy ustawić opcję `dependency`, która może przyjąć postać zarówno obiektu klasy `Dependency`, jak i tablicy konfiguracyjnej, służacej do utworzenia obiektu zależności. Poniższy kod określa pamięć podręczną fragmentu jako zależną od zmiany wartości kolumny `updated_at`:

```
$dependency = [
    'class' => 'yii\caching\DbDependency',
    'sql' => 'SELECT MAX(updated_at) FROM post',
];

if ($this->beginCache($id, ['dependency' => $dependency])) {

    // ... generowanie zawartości w tym miejscu ...

    $this->endCache();
}
```

Wariacje

Zapisana zawartość może mieć kilka wersji, zależnych od niektórych parametrów. Przykładowo, w aplikacji Web wspierającej kilka języków, ten sam fragment kodu w widoku może generować zawartość w różnych językach. Z tego powodu wymagana może być konieczność zapisu zawartości w wariacji zależnej od aktualnie wybranego języka aplikacji.

Aby określić wariacje pamięci podręcznej, ustaw opcję `variations`, która powinna mieć postać tablicy wartości skalarnych, z których każda będzie reprezentować odpowiedni czynnik modyfikujący wersję. Dla prykładu, aby zapisać zawartość w zależności od języka, możesz użyć następującego kodu:

```
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {

    // ... generowanie zawartości w tym miejscu ...

    $this->endCache();
}
```

Warunkowe uruchamianie pamięci podręcznej

Czasem konieczne może być uruchamianie pamięci podręcznej fragmentów tylko w przypadku, gdy spełnione są określone warunki. Przykładowo, dla

strony zawierającej formularz, pożądane może być zapisanie i wyświetlenie go z pamięci podręcznej tylko w momencie pierwszego pobrania jego treści (poprzez żądanie GET). Każde kolejne żądanie wyświetlenia formularza (już za pomocą metody POST) nie powinno być zapisane w pamięci, ponieważ może zawierać dane podane przez użytkownika. Aby użyć takiego mechanizmu, należy ustawić opcję `enabled`, jak w przykładzie poniżej:

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {

    // ... generowanie zawartości w tym miejscu ...

    $this->endCache();
}
```

10.2.2 Zagnieżdżony zapis w pamięci

Fragmenty zapisane w pamięci podręcznej mogą być zagnieżdżane. Oznacza to, że zapisany fragment może być częścią innego, również zapisanego w pamięci podręcznej. Przykładowo, komentarze mogą być zapisane jako fragmenty w pamięci podręcznej, które z kolei w całości również są zapisane jako większy fragment w pamięci. Poniższy kod pokazuje, w jaki sposób można zagnieździć dwa fragmenty w pamięci podręcznej:

```
if ($this->beginCache($id1)) {

    // ... generowanie zawartości ...

    if ($this->beginCache($id2, $options2)) {

        // ... generowanie zawartości ...

        $this->endCache();
    }

    // ... generowanie zawartości ...

    $this->endCache();
}
```

Zagnieżdżone fragmenty mogą mieć różne opcje zapisu. Dla przykładu, wewnętrzny i zewnętrzy fragment może mieć inną wartość czasu życia. Nawet w przypadku, gdy zawartość zapisana w zewnętrznym fragmencie straci ważność, wewnętrzny fragment wciąż będzie pobierany z pamięci. Nie zadziała to jednak w przeciwnym przypadku - dopóki zewnętrzny fragment będzie ważny, będzie zwracał tą samą zawartość za każdym razem, niezależnie od tego, czy zawartość wewnętrznego fragmentu już wygasła, czy nie. Z tego powodu należy zwrócić szczególną ostrożność przy ustalaniu czasu życia lub zależności zagnieżdżonych fragmentów, ponieważ “stara” zawartość może być wciąż niezamierzenie przechowywana w zewnętrznym fragmencie.

10.2.3 Dynamiczna zawartość

Używając pamięci podręcznej fragmentów, można napotkać na sytuację, kiedy duża część zawartości strony jest względnie statyczna z wyjątkiem kilku nielicznych miejsc. Przykładowo, nagłówek strony może wyświetlać pasek głównego menu razem z imieniem aktualnie zalogowanego użytkownika. Innym kłopotem może być to, że zapisywana w pamięci zawartość może zawierać kod PHP, który musi być wykonany dla każdego żądania (np. kod rejestrujący paczkę assetów). W obu tych przypadkach pomoże nam skorzystanie z funkcjonalności tzw. *dynamicznej zawartości*.

Dynamiczna zawartość oznacza fragment zwrotki, który nie powinien zostać zapisany w pamięci podręcznej, nawet jeśli znajduje się w bloku objętym zapisem. Aby określić zawartość jako dynamiczną, musi być ona generowana poprzez wykonanie jakiegoś kodu PHP dla każdego zapytania, nawet jeśli całość fragmentu serwowana jest z pamięci podręcznej.

Możesz wywołać metodę `renderDynamic()` wewnątrz zapisywanego fragmentu, aby wstawić w danym miejscu zawartość dynamiczną, jak w przykładzie poniżej:

```
if ($this->beginCache($id1)) {

    // ... generowanie zawartości ...

    echo $this->renderDynamic('return Yii::$app->user->identity->name;');

    // ... generowanie zawartości ...

    $this->endCache();
}
```

Metoda `renderDynamic()` przyjmuje jako parametr kod PHP. Wartość zwracana przez ten kod jest traktowana jako zawartość dynamiczna. Ten sam kod PHP będzie wykonany dla każdego zapytania, niezależnie od tego, czy obejmujący go fragment będzie pobierany z pamięci podręcznej czy też nie.

10.3 Pamięć podręczna stron

Pamięć podręczna stron odnosi się do zapisu zawartości całej strony po stronie serwera. Kiedy zostanie ona ponownie wywołana, zawartość zostanie wyświetlona od razu z pamięci podręcznej zamiast generować ją ponownie od podstaw.

Pamięć podręczna stron jest obsługiwana przez filtr akcji `PageCache`. Poniżej znajdziesz przykładowy sposób użycia go w klasie kontrolera:

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

W powyższym przykładzie zakładamy użycie pamięci podręcznej tylko dla akcji `index` - zawartość strony powinna zostać zapisana na maksymalnie 60 sekund i powinna różnić się w zależności od wybranego w aplikacji języka. Dodatkowo, jeśli całkowita liczba postów w bazie danych ulegnie zmianie, zawartość pamięci powinna natychmiast stracić ważność i zostać pobrana ponownie.

Jak widać, pamięć podręczna stron jest bardzo podobna do pamięci podręcznej fragmentów. W obu przypadkach można użyć opcji takich jak `duration` (czas ważności), `dependencies` (zależności), `variations` (warianty) oraz `enabled` (flaga aktywowania). Główną różnicą w tych dwóch przypadkach jest to, że pamięć podręczna stron jest implemetowana jako filtr akcji, a pamięć podręczna fragmentów jako widżet.

Oczywiście nic nie stoi na przeszkodzie, aby używać pamięci podręcznej fragmentów jak i zawartości dynamicznej w połączeniu z pamięcią podręczną stron.

10.4 Pamięć podręczna HTTP

Oprócz pamięci podręcznej tworzonej po stronie serwera, która została opisana w poprzednich rozdziałach, aplikacje mogą również skorzystać z pamięci podręcznej tworzonej po stronie klienta, aby zaoszczędzić czas poświęcany na ponowne generowanie i przesyłanie identycznej zawartości strony.

Aby skorzystać z tego mechanizmu, należy skonfigurować `yii\filters\HttpCache` jako filtr kontrolera akcji, których wyrenderowana zwrotka może być zapisana w pamięci podręcznej po stronie klienta. `HttpCache` obsługuje tylko żądania typu `GET` i `HEAD` i dla tych typów tylko trzy nagłówki HTTP związane z pamięcią podręczną:

- `Last-Modified`
- `Etag`
- `Cache-Control`

10.4.1 Nagłówek

Nagłówek `Last-Modified` korzysta ze znacznika czasu, aby określić, czy strona została zmodyfikowana od momentu jej ostatniego zapisu w pamięci podręcznej.

Możesz skonfigurować właściwość `yii\filters\HttpCache::$lastModified`, aby przesyłać nagłowek `Last-Modified`. Właściwość powinna być typu PHP callable i zwracać uniksowy znacznik czasu informujący o czasie modyfikacji strony. Sygnatura metody jest następująca:

```
/**
 * @param Action $action aktualnie przetwarzany obiekt akcji
 * @param array $params wartość w␣laściwości "params"
 * @return int uniksowy znacznik czasu modyfikacji strony
 */
function ($action, $params)
```

Poniżej znajdziesz przykład wykorzystania nagłówka `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');
            },
        ],
    ];
}
```

Kod ten uruchamia pamięć podręczną HTTP wyłącznie dla akcji `index`, która powinna wygenerować nagłówek HTTP `Last-Modified` oparty o datę ostatniej aktualizacji postów. Przeglądarka, wyświetlając stronę `index` po raz pierwszy, otrzymuje jej zawartość wygenerowaną przez serwer; każda kolejna wizyta, przy założeniu, że żaden post nie został zmodyfikowany w międzyczasie, skutkuje wyświetleniem wersji strony przechowywanej w pamięci podręcznej po stronie klienta, zamiast generować ją ponownie przez serwer. W rezultacie, renderowanie zawartości po stronie serwera i przesyłanie jej do klienta jest pomijane.

10.4.2 Nagłowek

Nagłowek "Entity Tag" (lub w skrócie `ETag`) wykorzystuje skrót hash jako reprezentację strony. W momencie, gdy strona się zmieni, jej hash również automatycznie ulega zmianie. Porównując hash przechowywany po stronie klienta z hashem wygenerowanym przez serwer, mechanizm pamięci podręcznej ustala, czy strona się zmieniła i powinna być ponownie przesłana.

Możesz skonfigurować właściwość `yii\filters\HttpCache::$etagSeed`, aby przesłać nagłowek `ETag`. Właściwość powinna być typu PHP callable i zwracać ziarno do wygenerowania hasha ETag. Sygnatura metody jest następująca:

```
/**
 * @param Action $action aktualnie przetwarzany obiekt akcji
 * @param array $params wartość w␣łaściwości "params"
 * @return string ␣łańcuch znaków użyty do generowania hasha ETag
 */
function ($action, $params)
```

Poniżej znajdziesz przykład użycia nagłówka `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]);
            },
        ],
    ];
}
```

Kod ten uruchamia pamięć podręczną HTTP wyłącznie dla akcji `view`, która powinna wygenerować nagłówek HTTP `ETag` oparty o tytuł i zawartość przeglądanego posta. Przeglądarka, wyświetlając stronę `view` po raz pierwszy, otrzymuje jej zawartość wygenerowaną przez serwer; każda kolejna wizyta, przy założeniu, że ani tytuł, ani zawartość posta nie została zmodyfikowana w międzyczasie, skutkuje wyświetleniem wersji strony przechowywanej w pamięci podręcznej po stronie klienta, zamiast generować ją ponownie przez serwer. W rezultacie, renderowanie zawartości po stronie serwera i przesyłanie jej do klienta jest pomijane.

ETagi pozwalają na bardziej skomplikowane i precyzyjne strategie przechowywania w pamięci podręcznej niż nagłówki `Last-Modified`. Dla przykładu, ETag może być zmieniony dla strony w momencie, gdy użyty na niej będzie nowy szablon wyglądu.

Zasobożerne generowanie ETagów może przekreślić cały zysk z użycia `HttpCache` i wprowadzić niepotrzebny narzut, ponieważ muszą być one określane przy każdym żądaniu. Z tego powodu należy używać jak najprostszych metod generujących.

> **Uwaga:** Aby spełnić wymagania RFC 7232[1], `HttpCache` przesyła zarówno nagłówek `ETag`, jak i `Last-Modified`, jeśli oba są skonfi-

[1] https://datatracker.ietf.org/doc/html/rfc7232#section-2.4

gurowane. Jeśli klient wysyła nagłówek `If-None-Match` razem z `If-Modified-Since`, tylko pierwszy z nich jest brany pod uwagę.

10.4.3 Nagłówek

Nagłówek `Cache-Control` określa ogólną politykę obsługi pamięci podręcznej stron. Możesz go przesłać konfigurując właściwość `yii\filters\HttpCache::$cacheControlHeader` z wartością nagłówka. Domyślnie przesyłany jest następujący nagłówek:

```
Cache-Control: public, max-age=3600
```

10.4.4 Ogranicznik pamięci podręcznej sesji

Kiedy strona używa mechanizmu sesji, PHP automatycznie wysyła związane z pamięcią podręczną nagłówki HTTP, określone w `session.cache_limiter` w ustawieniach PHP INI. Mogą one kolidować z funkcjonalnością `HttpCache`, a nawet całkowicie ją wyłączyć - aby temu zapobiec, `HttpCache` blokuje to automatyczne wysyłanie. Jeśli jednak chcesz zmienić to zachowanie, powinieneś skonfigurować właściwość `yii\filters\HttpCache::$sessionCacheLimiter`. Powinna ona przyjmować wartość zawierającą łańcuch znaków `public`, `private`, `private_no_expire` i `nocache`. Szczegóły dotyczące tego zapisu znajdziesz w dokumentacji PHP dla session_cache_limiter()[2].

10.4.5 Korzyści dla SEO

Boty silników wyszukiwarek zwykle respektują ustawienia nagłówków pamięci podręcznej. Niektóre automaty mają limit ilości stron zaindeksowanych w pojedynczej domenie w danej jednostce czasu, dlatego też wprowadzenie nagłówków dla pamięci podręcznej może w znaczącym stopniu przyspieszyć cały proces indeksacji, poprzez redukcję ilości stron, które trzeba przeanalizować.

[2] https://www.php.net/manual/pl/function.session-cache-limiter.php

Rozdział 11

Webserwisy z wykorzystaniem REST

Error: not existing file: rest-quick-start.md

Error: not existing file: rest-resources.md

Error: not existing file: rest-controllers.md

11.1 Routing

Po przygotowaniu klas zasobów i kontrolerów dostęp do nich można uzyskać w ten sam sposób, jak w przypadku zwykłej aplikacji, używając URL np. `http://localhost/index.php?r=user/create`.

W praktyce zwykle chcemy skorzystać z opcji "ładnych" URLi i metod HTTP. Przykładowo żądanie `POST /users` może oznaczać wywołanie akcji `user/create`, co możemy uzyskać w łatwy sposób konfigurując komponent aplikacji `urlManager` w pliku konfiguracyjnym jak poniżej:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

Porównując to z menadżerem URLi dla aplikacji Web, główną nowością tutaj jest użycie `UrlRule` do routingu RESTfulowych zasobów API. Ta specjalna klasa zasad URL stworzy cały zestaw potomnych zasad URL obsługujących routing i tworzenie URLi dla wyznaczonego kontrolera. Dla przykładu, kod powyżej jest zgrubnym odpowiednikiem następujących zasad:

```
[
    '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',
]
```

I poniższe punkty końcowe API są obsługiwane przez tę zasadę:

- `GET /users`: lista wszystkich użytkowników strona po stronie;
- `HEAD /users`: pokazuje streszczenie informacji listy użytkowników;
- `POST /users`: tworzy nowego użytkownika;
- `GET /users/123`: zwraca szczegóły na temat użytkownika 123;
- `HEAD /users/123`: zwraca streszczenie informacji o użytkowniku 123;
- `PATCH /users/123` i `PUT /users/123`: aktualizuje użytkownika 123;
- `DELETE /users/123`: usuwa użytkownika 123;
- `OPTIONS /users`: pokazuje obsługiwane metody dla punktu końcowego `/users`;
- `OPTIONS /users/123`: pokazuje obsługiwane metody dla punktu końcowego `/users/123`.

Możesz skonfigurować opcje `only` i `except`, aby wskazać listę akcji, które mają być odpowiednio: tylko obsługiwane lub pominięte. Przykładowo,

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'except' => ['delete', 'create', 'update'],
],
```

Dodatkowo można dodać opcję `patterns` lub `extraPatterns`, aby zredefiniować istniejące wzorce lub dodać nowe obsługiwane przez tę zasadę. Dla przykładu, aby dodać obsługę nowej akcji `search` dla punktu końcowego `GET /users/search`, skonfiguruj opcję `extraPatterns` jak następuje,

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'extraPatterns' => [
        'GET search' => 'search',
    ],
]
```

Na pewno zwróciłeś uwagę na to, że ID kontrolera `user` występuje tu w formie mnogiej jako `users` dla URLi punktu końcowego. Dzieje się tak, ponieważ `UrlRule` automatycznie przechodzi na formę mnogą dla ID kontrolerów podczas tworzenia potomnych zasad URL. Zachowanie to można wyłączyć ustawiając `pluralize` na `false`.

> **Informacja:** forma mnoga ID kontrolerów jest tworzona poprzez metodę `pluralize()`. Uwzględnia ona specjalne zasady tworzenia form mnogich. Dla przykładu, od słowa `box` zostanie utworzona liczba mnoga `boxes` a nie `boxs`.

W przypadku, gdy mechanizm automatycznego tworzenia formy mnogiej nie spełnia Twoich oczekiwań, możesz również skonfigurować właściwość `controller`, aby bezpośrednio określić w jaki sposób nazwa użyta w punkcie końcowym URLi ma być zmapowana na ID kontrolera. Dla przykładu, poniższy kod mapuje nazwę `u` na ID kontrolera `user`.

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['u' => 'user'],
]
```

Error: not existing file: rest-response-formatting.md

Error: not existing file: rest-authentication.md

11.2 Limit użycia

W celu zapobiegnięcia nadużyciom, powinno się rozważyć wprowadzenie *limitu użycia* swojego API. Może to być na przykład ograniczenie do maksymalnie 100 zapytań do API dla każdego użytkownika w czasie 10 minut. Jeśli użytkownik przekroczy ten limit w zadanym czasie, należy zwrócić odpowiedź ze statusem 429 (oznaczającym "Zbyt dużo zapytań").

Aby ustalić limit użycia, `klasa identyfikująca użytkownika` powinna zaimplementować `RateLimitInterface`. Interfejs ten wymaga dodania trzech metod:

- `getRateLimit()`: zwraca maksymalną liczbę zapytań i okres czasu (np. `[100, 600]` oznacza maksymalnie 100 zapytań do API w czasie 600 sekund).
- `loadAllowance()`: zwraca liczbę pozostałych dozwolonych zapytań z limitu i uniksowy znacznik czasu wskazujący datę ostatniego sprawdzenia limitu.
- `saveAllowance()`: zapisuje liczbę pozostałych dozwolonych zapytań i aktualny uniksowy znacznik czasu.

Do celów obsługi powyższych metod można wykorzystać dwie dodatkowe kolumny w bazie danych użytkowników dla liczby dokonanych połączeń i znacznika czasu. Po ustaleniu tych wartości, metody `loadAllowance()` i `saveAllowance()` mogą być poprawnie zaimplementowane do odczytu i zapisu tych wartości dla aktualnego zautoryzowanego użytkownika. Aby zwiększyć wydajność tego mechanizmu, należy rozważyć użycie pamięci podręcznej lub bazy typu NoSQL.

Po zaimplemetowaniu wymaganego interfejsu, Yii automatycznie użyje `RateLimiter`, skonfigurowanego jako filtr akcji dla `Controller`, aby pilnować limitu użycia API. Mechanizm rzuci wyjątek `TooManyRequestsHttpException`, kiedy limit zostanie przekroczony.

Po dodaniu limitu, każda odpowiedź będzie domyślnie zawierała następujące nagłówki HTTP, zawierające informacje o aktualnym użyciu limitu:

- `X-Rate-Limit-Limit`, maksymalna liczba zapytań w zadanym okresie czasu,
- `X-Rate-Limit-Remaining`, liczba pozostałych dozwolonych zapytań z limitu w aktualnym okresie czasu,
- `X-Rate-Limit-Reset`, liczba sekund, którą należy odczekać, aby uzyskać ponownie maksymalną liczbę zapytań z limitu.

Wysyłanie powyższych nagłówków można wyłączyć konfigurując `enableRateLimitHeaders` w klasie kontrolera REST jak w poniższym przykładzie.

```
public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['rateLimiter']['enableRateLimitHeaders'] = false;
    return $behaviors;
}
```

11.3 Wersjonowanie

Cechą dobrego API jest jego *wersjonowanie*: zmiany i nowe funkcjonalności powinny być implementowane w nowych wersjach API, zamiast ciągłych modyfikacji jednej już istniejącej. W przeciwieństwie do aplikacji Web, nad którymi ma się pełną kontrolę zarówno po stronie klienta, jak i serwera, nad API zwykle nie posiada się kontroli po stronie klienta. Z tego powodu niezwykle istotnym jest, aby zachować pełną wsteczną kompatybilność (BC = backward compatibility), kiedy to tylko możliwe. Jeśli konieczne jest wprowadzenie zmiany, która może nie spełniać BC, należy wprowadzić ją w nowej wersji API, z kolejnym numerem. Istniejące klienty mogą wciąż używać starej, działającej wersji API, a nowe lub uaktualnione klienty mogą otrzymać nową funkcjonalność oferowaną przez kolejną wersję API.

> **Wskazówka:** Zapoznaj się z Wersjonowaniem semantycznym[1], aby uzyskać więcej informacji na temat nazewnictwa numerów wersji.

Jedną z często spotykanych implementacji wersjonowania API jest dodawanie numeru wersji w adresach URL API. Dla przykładu `https://example.com/v1/users` oznacza punkt końcowy `/users` API w wersji 1.

Inną metodą wersjonowania API, która zyskuje ostatnio popularność, jest umieszczanie numeru wersji w nagłówkach HTTP żądania. Zwykle używa się do tego nagłówka `Accept`:

```
// poprzez parametr
Accept: application/json; version=v1
// poprzez dostarczany typ zasobu
Accept: application/vnd.company.myapp-v1+json
```

Obie metody mają swoje wady i zalety i wciąż prowadzone są dyskusje na ich temat. Poniżej prezentujemy strategię wersjonowania, która w praktyczny sposób łączy je obie:

- Umieść każdą główną wersję implementacji API w oddzielnym module, którego ID odpowiada numerowi głównej wersji (np. `v1`, `v2`). Adresy URL API będą zawierały numery głównych wersji.
- Wewnątrz każdej głównej wersji (i w związku z tym w każdym odpowiadającym jej module), użyj nagłówka HTTP `Accept`, aby określić pomniejszy numer wersji i napisz warunkowy kod odpowiadający temu numerowi.

Każdy moduł obsługujący główną wersję powinien zawierać klasy zasobów i kontrolerów odpowiednie dla tej wersji. W celu lepszego rozdzielenia zadań kodu, możesz trzymać razem zestaw podstawowych wspólnych klas zasobów i kontrolerów w jednym miejscu i rozdzielać go na podklasy w każdym module wersji. W podklasie implementujesz bazowy kod, taki jak `Model::fields()`.

Struktura Twojego kodu może wyglądać jak poniższa:

[1] https://semver.org/lang/pl/

```
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
```

Konfiguracja Twojej aplikacji mogłaby wyglądać następująco:

```
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']],
            ],
        ],
    ],
];
```

Rezultatem powyższego kodu będzie skierowanie pod adresem `https://example.com/v1/users`

do listy użytkowników w wersji 1, podczas gdy `https://example.com/v2/users` pokaże użytkowników w wersji 2.

Dzięki podziałowi na moduły, kod różnych głównych wersji może być dobrze izolowany, ale jednocześnie wciąż możliwe jest ponowne wykorzystanie wspólnego kodu poprzez wspólną bazę klas i dzielonych zasobów.

Aby prawidłowo obsłużyć pomniejsze numery wersji, możesz wykorzystać funkcjonalność negocjatora zawartości dostarczaną przez behavior `contentNegotiator`. Ustawi on właściwość `yii\web\Response::$acceptParams`, kiedy już zostanie ustalone, który typ zasobów wspierać.

Przykładowo, jeśli żądanie jest wysłane z nagłówkiem HTTP `Accept: application/json; version=v1`, po negocjacji zawartości `yii\web\Response::$acceptParams` będzie zawierać wartość `['version' => 'v1']`.

Bazując na informacji o wersji w `acceptParams`, możesz napisać obsługujący ją warunkowy kod w miejscach takich jak akcje, klasy zasobów, serializatory, itp., aby zapewnić odpowiednią funkcjonalność.

Ponieważ pomniejsze wersje z definicji wymagają zachowania wstecznej kompatybilności, w kodzie nie powinno znaleźć się zbyt wiele miejsc, gdzie numer wersji będzie sprawdzany. W przeciwnym wypadku możliwe, że konieczne będzie utworzenie kolejnej głównej wersji API.

11.4 Obsługa błędów

Podczas obsługi żądania RESTfulowego API, w przypadku wystąpienia błędu w zapytaniu użytkownika lub gdy stanie się coś nieprzewidywanego z serwerem, możesz po prostu rzucić wyjątkiem, aby powiadomić użytkownika, że coś poszło nieprawidłowo. Jeśli możesz zidentyfikować przyczynę błędu (np. żądany zasób nie istnieje), powinieneś rozważyć rzucenie wyjątkiem razem z odpowiednim kodem statusu HTTP (np. `NotFoundHttpException` odpowiada statusowi o kodzie 404). Yii wyśle odpowiedź razem z odpowiadającym jej kodem i treścią statusu HTTP. Yii dołączy również do samej odpowiedzi zserializowaną reprezentację wyjątku. Przykładowo:

```
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
}
```

Poniższa lista zawiera kody statusów HTTP, które są używane przez Yii REST framework:

- `200`: OK. Wszystko działa w porządku.
- `201`: Zasób został poprawnie stworzony w odpowiedzi na żądanie `POST`. Nagłówek `Location` zawiera URL kierujący do nowoutworzonego zasobu.
- `204`: Żądanie zostało poprawnie przetworzone, ale odpowiedź nie zawiera treści (jak w przypadku żądania `DELETE`).
- `304`: Zasób nie został zmodyfikowany. Można użyć wersji przetrzymywanej w pamięci podręcznej.
- `400`: Nieprawidłowe żądanie. Może być spowodowane przez wiele czynników po stronie użytkownika, takich jak przekazanie nieprawidłowych danych JSON, nieprawidłowych parametrów akcji, itp.
- `401`: Nieudana autentykacja.
- `403`: Autoryzowany użytkownik nie ma uprawnień do danego punktu końcowego API.
- `404`: Żądany zasób nie istnieje.
- `405`: Niedozwolona metoda. Sprawdź nagłówek `Allow`, aby poznać dozwolone metody HTTP.
- `415`: Niewspierany typ mediów. Żądany typ zawartości lub numer wersji jest nieprawidłowy.
- `422`: Nieudana walidacja danych (dla przykładu w odpowiedzi na żądanie `POST`). Sprawdź treść odpowiedzi, aby poznać szczegóły błędu.
- `429`: Zbyt wiele żądań. Żądanie zostało odrzucone z powodu osiagnięcia limitu użycia.
- `500`: Wewnętrzny błąd serwera. To może być spowodowane wewnętrznymi błędami programu.

11.4.1 Modyfikowanie błędnej odpowiedzi

Czasem wymagane może być dostosowanie domyślnego formatu błędnej odpowiedzi. Dla przykładu, zamiast używać różnych statusów HTTP dla oznaczenia różnych błędów, można serwować zawsze status 200 i dodawać prawidłowy kod statusu HTTP jako część struktury JSON w odpowiedzi, jak pokazano to poniżej:

```
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
```

```
    }
}
```

Aby osiągnąć powyższy efekt, należy skonfigurować odpowiedź na event `beforeSend` dla komponentu `response` w aplikacji:

```
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;
                }
            },
        ],
    ],
];
```

Powyższy kod zreformatuje odpowiedź (zarówno typu sukces jak i błąd), kiedy `suppress_response_code` zostanie przekazane jako parametr `GET`.

Rozdział 12

Narzędzia wspomagające tworzenie aplikacji

Rozdział 13

Testowanie

13.1 Testowanie

Testowanie jest istotnym elementem produkcji każdego oprogramowania. Niezależnie, czy jesteśmy tego świadomi, czy też nie, testy przeprowadzamy nieustannie. Dla przykładu, kiedy napiszemy klasę w PHP, możemy debugować ją krok po kroku lub po prostu użyć wyrażeń jak echo lub die, aby sprawdzić, czy implementacja działa zgodnie z naszym początkowym planem. W przypadku aplikacji web wprowadzamy testowe dane w formularzach, aby upewnić się, że strona odpowiada tak, jak powinna.

Proces testowania może zostać zautomatyzowany, dzięki czemu za każdym razem, kiedy musimy coś sprawdzić, wystarczy wywołać kod, który zrobi to za nas. Kod, który weryfikuje zgodność wyniku z planowaną odpowiedzią, jest nazywany testem, a proces jego tworzenia i późniejszego wykonania jest nazywany testowaniem zautomatyzowanym, co jest głównym tematem tych rozdziałów.

13.1.1 Tworzenie kodu z testami

Tworzenie kodu opartego na testach (Test-Driven Development, TDD) i opartego na zachowaniach (Behavior-Driven Development, BDD) jest podejściem deweloperskim opierającym się na opisywaniu zachowania fragmentu kodu lub też całej jego funkcjonalności jako zestawu scenariuszy lub testów przed napisaniem właściwego kodu i dopiero potem stworzeniu implementacji, która pozwoli na poprawne przejście testów, spełniających zadane kryteria.

Proces tworzenia funkcjonalności wygląda następująco:

- Stwórz nowy test, opisujący funkcjonalność do zaimplementowania.
- Uruchom nowy test i upewnij się, że zakończy się błędem. To właściwe zachowanie, ponieważ nie ma jeszcze implementacji funkcjonalności.
- Napisz prosty kod, który przejdzie poprawnie nowy test.

- Uruchom wszystkie testy i upewnij się, że wszystkie zakończą się poprawnie.
- Ulepsz kod, sprawdzając czy testy wciąż są zdane.

Po zakończeniu proces jest powtarzany dla kolejnej funkcjonalności lub ulepszenia. Jeśli istniejąca funkcjonalność ma być zmodyfikowana, testy powinny być również zmienione.

> **Wskazówka**: Jeśli czujesz, że tracisz czas, przeprowadzając dużo krótkich i prostych iteracji, spróbuj objąć testowym scenariuszem więcej działań, aby sprawdzić więcej kodu, przed ponownym uruchomieniem testów. Jeśli debugujesz zbyt dużo, spróbuj zrobić dokładnie na odwrót.

Powodem tworzenia testów przed jakąkolwiek implementacją, jest możliwość skupienia się na tym, co chcemy osiągnąć, zanim przystąpimy do "w jaki sposób to zrobić". Zwykle prowadzi to do stworzenia lepszej warstwy abstrakcji i łatwiejszej obsługi testów w przypadku poprawek funkcjonalności.

Podsumowując, zalety takiego projektowania są następujące:

- Pozwala na skupienie się na pojedynczej rzeczy na raz, dzięki czemu pozwala na lepsze planowanie i implementacje.
- Obejmuje testami więcej funkcjonalności w większym stopniu, co oznacza, że jeśli testy zakończyły się poprawnie jest spore prawdopodobieństwo, że wszystko działa poprawnie.

Na dłuższą metę przynosi to zwykle efekt w postaci mnóstwa oszczędzonego czasu i problemów.

> **Wskazówka**: Jeśli chcesz dowiedzieć się więcej na temat reguł ustalania wymagań dla oprogramowania i modelowania istoty tego rozdziału, warto zapoznać się z domenowym podejściem do tworzenia aplikacji (Domain Driven Development, DDD)[1].

13.1.2 Kiedy i jak testować

Podejście typu "testy najpierw" opisane powyżej, ma sens w przypadku długofalowych i relatywnie skomplikowanych projektów i może być przesadne w przypadku prostszych. Przesłanki, kiedy testy są odpowiednie są następujące:

- Projekt jest już duży i skomplikowany.
- Wymagania projektowe zaczynają być skomplikowane. Projekt wciąż się powiększa.
- Projekt jest planowany jako długoterminowy.
- Koszty potencjalnych błędów są zbyt duże.

[1] https://pl.wikipedia.org/wiki/Domain-Driven_Design

Nie ma nic złego w tworzeniu testów obejmujących zachowania istniejących implementacji.

- Projekt jest oparty starszym kodzie i stopniowo przepisywany.
- Projekt, nad którym masz pracować, nie ma w ogóle testów.

W niektórych przypadkach jakakolwiek forma automatycznego testu może być nadmiarowa:

- Projekt jest prosty i nie jest rozbudowywany.
- Projekt jest jednorazowym zadaniem, które nie będzie rozwijane.

Pomimo tego, jeśli masz na to czas, automatyzacja testowania jest również dobrym pomysłem.

13.1.3 Biblioteka

- Test Driven Development: By Example - Kent Beck. (ISBN: 0321146530)

13.2 Przygotowanie środowiska testowego

> Uwaga: Ta sekcja jest w trakcie tworzenia.

Yii 2 jest oficjalnie zintegrowany z `Codeception`[2] - frameworkiem testowym, pozwalającym na utworzenie testów następujących typów:

- Testy jednostkowe - sprawdzające czy pojedyncza jednostka kodu działa poprawnie;
- Testy funkcjonalne - weryfikujące scenariusze działań z perspektywy użytkownika poprzez emulację przeglądarki;
- Testy akceptacyjne - weryfikujące scenariusze działań z perspektywy użytkownika w przeglądarce.

Yii dostarcza gotowy do użycia zestaw testów wszystkich trzech typów zarówno dla szablonu projektu `yii2-basic`[3] jak i `yii2-advanced`[4].

W celu uruchomienia testów koniecznie jest zainstalowanie Codeception[5]. Instalację można wykonać lokalnie - dla konkretnego pojedynczego projektu - lub globalnie - na komputerze deweloperskim.

Poniższe komendy służą do instalacji lokalnej:

```
composer require "codeception/codeception=2.0.*"
composer require "codeception/specify=*"
composer require "codeception/verify=*"
```

Do instalacji globalnej należy dodać dyrektywę `global`:

[2] https://github.com/Codeception/Codeception
[3] https://github.com/yiisoft/yii2-app-basic
[4] https://github.com/yiisoft/yii2-app-advanced
[5] https://github.com/Codeception/Codeception

```
composer global require "codeception/codeception=2.0.*"
composer global require "codeception/specify=*"
composer global require "codeception/verify=*"
```

Jeśli nigdy wcześniej nie używałeś Composera do globalnych pakietów, uruchom komendę `composer global status`. W odpowiedzi powinieneś uzyskać:

```
Changed current directory to <directory>
```

Następnie dodaj `<directory>/vendor/bin` do zmiennej systemowej `PATH`. Od tej pory będziesz mógł użyć `codecept` z linii komend globalnie.

> Uwaga: instalacja globalna Codeception pozwala na użycie go we wszystkich projektach na komputerze deweloperskim oraz na wykonywanie komendy `codecept` globalnie bez konieczności wskazywania ścieżki. Taka instalacja może jednak nie być pożądana, kiedy, dla przykładu, dwa różne projekty wymagają różnych wersji Codeception. Dla uproszczenia wszystkie komendy powłoki odnoszące się do uruchamiania testów użyte w tym przewodniku są napisane przy założeniu, że Codeception został zainstalowany globalnie.

13.3 Testy jednostkowe

> Uwaga: Ta sekcja jest w trakcie tworzenia.

Test jednostkowy weryfikuje poprawność działania pojedynczej jednostki kodu. W programowaniu zorientowanym obiektowo najbardziej podstawową jednostką kodu jest klasa. Test jednostkowy musi sprawdzić, czy każda z metod interfejsu klasy działa poprawnie tj. czy przy zadanych różnych parametrach wejściowych metoda zwraca spodziewane rezultaty. Testy jednostkowe są zazwyczaj tworzone przez osoby, które piszą klasy poddawane tym testom.

Testy jednostkowe w Yii są oparte o PHPUnit oraz, opcjonalnie, Codeception, zatem zalecane jest, aby zapoznać się z ich dokumentacją:

- PHPUnit (dokumentacja zaczyna się w rozdziale 2)[6].
- Testy jednostkowe Codeception[7].

13.3.1 Uruchamianie testów jednostkowych dla podstawowego i zaawansowanego szablonu projektu

Prosimy o zapoznanie się z instrukcjami dostępnymi w plikach `apps/advanced/tests/README.md` i `apps/basic/tests/README.md`.

[6] https://phpunit.de/manual/current/en/writing-tests-for-phpunit.html

[7] https://codeception.com/docs/05-UnitTests

13.3.2 Testy jednostkowe frameworka

Jeśli chcesz przeprowadzić testy jednostkowe na frameworku Yii przejdź do sekcji "Od czego zacząć projektowanie w Yii 2[8]".

13.4 Testy funkcjonalne

Uwaga: Ta sekcja jest w trakcie tworzenia.

- Testy funkcjonalne Codeception[9]

13.4.1 Uruchamianie testów funkcjonalnych dla podstawowego i zaawansowanego szablonu projektu

Prosimy o zapoznanie się z instrukcjami dostępnymi w plikach `apps/advanced/tests/README.md` i `apps/basic/tests/README.md`.

13.5 Testy akceptacyjne

Uwaga: Ta sekcja jest w trakcie tworzenia.

- Testy akceptacyjne Codeception[10]

13.5.1 Uruchamianie testów akceptacyjnych dla podstawowego i zaawansowanego szablonu projektu

Prosimy o zapoznanie się z instrukcjami dostępnymi w plikach `apps/advanced/tests/README.md` i `apps/basic/tests/README.md`.

[8] https://github.com/yiisoft/yii2/blob/master/docs/internals/getting-started.md
[9] https://codeception.com/docs/04-FunctionalTests
[10] https://codeception.com/docs/03-AcceptanceTests

Error: not existing file: test-fixtures.md

Rozdział 14

Temata specjalne

14.1 Tworzenie własnej struktury aplikacji

> **Uwaga:** Ta sekcja jest w trakcie tworzenia.

Podstawowy[1] i zaawansowany[2] szablon projektu jest w pełni wystarczający w większości przypadków, ale czasem może zajść potrzeba stworzenia własnego szablonu, na bazie którego tworzony będzie projekt.

Szablony projektów w Yii to po prostu repozytoria zawierające plik `composer.json` i zarejestrowane jako paczki Composera. Każde repozytorium może stać sie taką paczką, dzięki czemu można je zainstalować wywołując komendę Composera `create-project`.

Ponieważ stworzenie nowego szablonu projektu od podstaw jest pracochłonne, łatwiej po prostu użyć jednego z gotowych szablonów jako bazy. W tym przykładzie skorzystamy z podstawowego szablonu.

14.1.1 Kopia podstawowego szablonu projektu

Pierwszym krokiem jest wykonanie kopii podstawowego szablonu Yii z repozytorium Git:

```
git clone git@github.com:yiisoft/yii2-app-basic.git
```

Po zakończeniu pobierania plików repozytorium, można skasować folder `.git` wraz z zawartością, ponieważ wprowadzonych przez nas zmian nie zamierzamy wysyłać z powrotem.

14.1.2 Modyfikacja plików

Następnie należy zmodyfikować plik `composer.json`, aby opisywał nasz szablon. Zmień wartości `name` (nazwa), `description` (opis), `keywords` (słowa kluczowe), `homepage` (strona domowa), `license` (licencja), i `support` (wsparcie) na

[1] https://github.com/yiisoft/yii2-app-basic
[2] https://github.com/yiisoft/yii2-app-advanced

takie, które odpowiadają nowemu szablonowi. Zmodyfikuj również `require`, `require-dev`, `suggest` i wszelkie inne opcje zgodnie z wymaganiami.

> **Uwaga:** W pliku `composer.json` użyj parametru `writable` znajdującego się w elemencie `extra`, aby określić uprawnienia dla plików, które zostaną ustawione po utworzeniu aplikacji na podstawie szablonu.

Następnie należy zmodyfikować właściwą strukturę i zawartość aplikacji, aby stanowiły domyślną początkową wersję dla projektów. Na samym końcu zmodyfikuj plik README, aby pasował do szablonu.

14.1.3 Tworzenie paczki

Nowy szablon umieść w odpowiadającym mu repozytorium Git. Jeśli zamierzasz udostępnić go jako open source, Github[3] jest najlepszym miejscem do tego celu. Jeśli jednak nie przewidujesz współpracy z innymi nad swoim szablonem, dowolne repozytorium Git będzie odpowiednie.

Następnie należy zarejestrować swoją paczkę dla Composera. Dla publicznie dostępnych szablonów paczkę należy zarejestrować w serwisie Packagist[4]. Z prywatnymi szablonami sprawa jest trochę bardziej skomplikowana - instrukcję, jak to zrobić, znajdziesz w dokumentacji Composera[5].

14.1.4 Użycie szablonu

Tylko tyle jest wymagane, aby stworzyć nowy szablon projektu Yii. Teraz już możesz rozpocząć pracę nad świeżym projektem, używając swojego szablonu, za pomocą komend:

```
composer global require "fxp/composer-asset-plugin:^1.4.1"
composer create-project --prefer-dist --stability=dev
mojafirma/yii2-app-fajna nowy-projekt
```

[3] https://github.com
[4] https://packagist.org/
[5] https://getcomposer.org/doc/05-repositories.md#hosting-your-own

Error: not existing file: tutorial-console.md

Error: not existing file: tutorial-core-validators.md

Error: not existing file: tutorial-docker.md

Error: not existing file: tutorial-i18n.md

14.2 Wysyłanie poczty

> **Uwaga:** Ta sekcja jest w trakcie tworzenia.

Yii wspiera tworzenie oraz wysyłanie wiadomości email, jednakże silnik dostarcza jedynie funkcjonalność składania treści oraz prosty interfejs. Mechanizm wysyłania wiadomości powinien być dostarczony przez rozszerzenie, ponieważ projekty mogą wymagać różnych implementacji, przez co mechanizm jest zależny od zewnętrznych usług i bibliotek.

Dla większości przypadków możesz używać oficjalnego rozszerzenia yii2-swiftmailer[6].

14.2.1 Konfiguracja

Konfiguracja tego komponentu zależy od rozszerzenia jakie wybrałeś. Generalnie, konfiguracja Twojego komponentu w aplikacji powinna wyglądać tak:

```
return [
    //....
    'components' => [
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
    ],
];
```

14.2.2 Podstawowe użycie

Kiedy komponent 'mailer' zostanie skonfigurowany, możesz użyć następującego kodu do wysłania wiadomości email:

```
Yii::$app->mailer->compose()
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Temat wiadomości')
    ->setTextBody('Zwyk␣la treść wiadomości')
    ->setHtmlBody('<b>Treść HTML wiadomości</b>')
    ->send();
```

W powyższym przykładzie metoda `compose()` tworzy instancję wiadomości email, która następnie jest wypełniana danymi i wysłana. Możesz utworzyć tutaj więcej złożonej logiki jeśli jest to potrzebne:

```
$message = Yii::$app->mailer->compose();
if (Yii::$app->user->isGuest) {
    $message->setFrom('from@domain.com')
```

[6] https://github.com/yiisoft/yii2-swiftmailer

```
} else {
    $message->setFrom(Yii::$app->user->identity->email)
}
$message->setTo(Yii::$app->params['adminEmail'])
    ->setSubject('Temat wiadomości')
    ->setTextBody('Zwyk␣la treść wiadomości')
    ->send();
```

> **Uwaga:** każde rozszerzenie mailingowe posiada dwie główne klasy: 'Mailer' oraz 'Message'. Klasa 'Mailer' zawsze posiada nazwę klasy 'Message'. Nie próbuj instancjować obiektu 'Message' bezpośrednio - zawsze używaj do tego metody `compose()`.

Możesz również wysłać wiele wiadomości na raz:

```
$messages = [];
foreach ($users as $user) {
    $messages[] = Yii::$app->mailer->compose()
        // ...
        ->setTo($user->email);
}
Yii::$app->mailer->sendMultiple($messages);
```

Niektóre rozszerzenia mailingowe mogą czerpać korzyści z tego sposobu, np. używając pojedyńczych wiadomości sieciowych.

14.2.3 Tworzenie treści maila

Yii pozwala na tworzenie treści aktualnej wiadomości email przez specjalne pliki widoków. Domyślnie, pliki te zlokalizowane są w ścieżce '@app/mail'.

Przykładowy widok pliku treści wiadomości email:

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


/* @var $this \yii\web\View instancja komponentu View */
/* @var $message \yii\mail\BaseMessage instancja nowo utworzonej wiadomości
email */

?>
<h2>Ta wiadomość pozwala Ci odwiedzić stronę g␣lówną naszej witryny przez
jedno kliknięcie</h2>
<?= Html::a('Idź do strony g␣lównej', Url::home('http')) ?>
```

W celu wykorzystania tego pliku do utworzenia treści wiadomości, przekaż po prostu nazwę tego widoku do metody `compose()`:

```
Yii::$app->mailer->compose('home-link') // wynik renderingu widoku staje się
treścią wiadomości
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Temat wiadomości')
    ->send();
```

Możesz przekazać dodatkowe parametry do metody `compose()`, które będą dostępne w plikach widoków:

```
Yii::$app->mailer->compose('greetings', [
    'user' => Yii::$app->user->identity,
    'advertisement' => $adContent,
]);
```

Możesz określić różne pliki do zwykłej treści oraz treści HTML:

```
Yii::$app->mailer->compose([
    'html' => 'contact-html',
    'text' => 'contact-text',
]);
```

Jeśli określisz nazwę widoku jako ciąg skalarny, to wynik jego renderowania zostanie użyty jako ciało HTML wiadomości, podczas gdy przy użyciu zwykłego teksu zostanie ono utworzone przez usunięcie wszystkich encji HTML z tego widoku.

Wynik renderowania widoku może zostać opakowany w szablon. Szablon możesz ustawić przez właściwość `htmlLayout` lub `textLayout`. Zadziała to w identyczny sposób co w standardowej aplikacji web. Szalony mogą zostać użyte do ustawienia styli CSS, lub innej wspólnej treści:

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

/* @var $this \yii\web\View view component instance */
/* @var $message \yii\mail\MessageInterface the message being composed */
/* @var $content string main view render result */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=<?=
Yii::$app->charset ?>" />
    <style type="text/css">
        .heading {...}
        .list {...}
        .footer {...}
    </style>
    <?php $this->head() ?>
```

```
</head>
<body>
    <?php $this->beginBody() ?>
    <?= $content ?>
    <div class="footer">Z pozdrowieniami, zespół <?= Yii::$app->name
?></div>
    <?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

14.2.4 Załączniki do wiadomości

Możesz dodać załączniki do wiadomości przez użycie metod `attach()` oraz `attachContent()`:

```
$message = Yii::$app->mailer->compose();

// Załącz plik z lokalnego systemu plików:
$message->attach('/path/to/source/file.pdf');

// Utwórz załącznik w locie:
$message->attachContent('Attachment content', ['fileName' => 'attach.txt',
'contentType' => 'text/plain']);
```

14.2.5 Osadzanie obrazków

W treści wiadomości możesz osadzać również obrazki przy użyciu metody `embed()`. Metoda ta zwraca ID załącznika, który powinien zostać później użyty w tagu 'img'. Użycie tej metody jest proste podczas tworzenia treści wiadomości z pliku widoku:

```
Yii::$app->mailer->compose('embed-email', ['imageFileName' =>
'/path/to/image.jpg'])
    // ...
    ->send();
```

Następnie, w pliku widoku możesz użyć następującego kodu:

```
<img src="<?= $message->embed($imageFileName); ?>">
```

14.2.6 Testowanie i debugowanie

Deweloperzy często muszą sprawdzić, czy emaile zostały wysłane przez aplikację lub jaka była ich treść. Możesz tego dokonać w łatwy sposób, używając dostarczonej przez Yii funkcjonalności, którą aktywujesz przez parametr `useFileTransport`. Jeśli zostanie aktywowana, każda wiadomość email będzie zapisywana do lokalnych plików zamiast zostać wysłana. Wszystkie pliki będą zapisane w ścieżce podanej w `fileTransportPath`, która domyślnie ustawiona jest na '@runtime/mail'.

> **Uwaga:** możesz albo zapisywać wiadomości do plików, albo wysyłać je do odbiorców, nie można wykonać tych dwóch czynności na raz.

Plik z wiadomością email może zostać otwarty przez standardowy edytor tekstu, dzięki czemu będziesz mógł przeglądać nagłówki oraz treść wiadomości.

> **Uwaga:** plik wiadomości jest tworzony przy użyciu metody `toString()`, więc jest zależny od aktualnie używanego rozszerzenia mailingowego w Twojej aplikacji.

14.2.7 Tworzenie własnego rozwiązania mailingowego

Aby utworzyć swoje własne rozwiązanie mailingowe, musisz utworzyć dwie klasy: 'Mailer' oraz 'Message'. Możesz rozszerzyć klasy `BaseMailer` i `BaseMessage` jako bazowe klasy do tego rozwiązania. Zawierają one podstawową logikę mechanizmu mailingu, który został opisany w tej sekcji. Oczywiście ich użycie nie jest obowiązkowe, wystarczy zaimplementowanie interfejsów `MailerInterface` oraz `MessageInterface`. Następnie musisz zaimplementować wszystkie abstrakcyjne metody do swoich klas.

Error: not existing file: tutorial-performance-tuning.md

14.3 Współdzielone środowisko hostujące

Współdzielone środowiska hostujące są często ograniczone, jeśli chodzi o możliwości ich konfiguracji i struktury folderów. Pomimo to, wciąż, w większości przypadków, możesz w takim środowisku uruchomić Yii 2.0 po kilku drobnych modyfikacjach.

14.3.1 Wdrożenie podstawowej aplikacji

W standardowym współdzielonym środowisku hostującym jest zwykle tylko jeden główny folder publiczny (webroot), zatem wygodniej jest stosować podstawowy szablon projektu. Korzystając z instrukcji w sekcji Instalowanie Yii, zainstaluj taki szablon lokalnie. Po udanej instalacji, dokonamy kilku modyfikacji, aby aplikacji mogła działać na współdzielonym środowisku.

Zmiana nazwy webroota

Połącz się ze swoim współdzielonym hostem za pomocą np. klienta FTP. Prawdopodobnie zobaczysz listę folderów podobną do poniższej.

```
config
logs
www
```

W tym przykładzie, `www` jest folderem webroot. Folder ten może mieć różne nazwy, zwykle stosowane są: `www`, `htdocs` i `public_html`.

Webroot w naszym podstawowym szablonie projektu nazywa się `web`. Przed skopiowaniem aplikacji na serwer, zmień nazwę lokalnego folderu webroot, aby odpowiadała folderowi na serwerze, czyli z `web` na `www`, `public_html` lub na inną nazwę, która używana jest na serwerze.

Folder root FTP jest zapisywalny

Jeśli masz prawa zapisu w folderze poziomu root, czyli tam, gdzie znajdują się foldery `config`, `logs` i `www`, skopiuj foldery `assets`, `commands` itd. bezpośrednio w to miejsce.

Dodatkowe opcje serwera

Jeśli Twój serwer to Apache, będziesz musiał dodać plik `.htaccess` z poniższą zawartością do folderu `web` (czy też `public_html`, bądź jakakolwiek jest jego nazwa), gdzie znajduje się plik `index.php`:

```
Options +FollowSymLinks
IndexIgnore */*

RewriteEngine on
```

```
# jeśli katalog lub plik istnieje, użyj go bezpośrednio
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# w innym przypadku przekieruj żądanie na index.php
RewriteRule . index.php
```

W przypadku serwera nginx nie powinieneś potrzebować dodatkowego pliku konfiguracyjnego.

Sprawdzenie wymagań

Aby uruchomić Yii, Twój serwer musi spełniać jego wymagania. Minimalnym wymaganiem jest PHP w wersji 5.4. Możesz sprawdzić wszystkie wymagania, kopiując plik `requirements.php` z folderu root do folderu webroot i uruchamiając go w przeglądarce pod adresem `https://example.com/requirements.php`. Nie zapomnij o skasowaniu tego pliku po sprawdzeniu wymagań.

14.3.2 Wdrożenie zaawansowanej aplikacji

Wdrażanie zaawansowanej aplikacji na współdzielonym środowisku jest odrobinę bardziej problematyczne, niż w przypadku podstawowej aplikacji, ponieważ wymaga ona dwóch folderów webroot, czego zwykle nie wspierają serwery środowisk współdzielonych. Będziemy musieli odpowiednio dostosować strukturę folderów.

Przeniesienie skryptów wejściowych do jednego folderu webroot

Na początek potrzebujemy folderu webroot. Stwórz nowy folder i nazwij go tak, jak webroot docelowego serwera, jak opisane zostało to w Zmiana nazwy webroota powyżej, np. `www` czy też `public_html`. Następnie utwórz poniższą strukturę, gdzie `www` jest folderem webroot, który właśnie stworzyłeś:

```
www
    admin
backend
common
console
environments
frontend
...
```

`www` będzie naszym folderem frontend, zatem przenieś tam zawartość `frontend/web`. Do folderu `www/admin` przenieś zawartość `backend/web`. W każdym przypadku będziesz musiał zmodyfikować ścieżki w plikach `index.php` i `index-test.php`.

Rozdzielone sesje i ciasteczka

Backend i frontend zostały stworzone z myślą o uruchamianiu ich z poziomu oddzielnych domen. Jeśli uruchamiamy je z poziomu jednej domeny, frontend i backend będą dzielić te same ciasteczka, co może wywołać konflikty. Aby temu zapobiec, zmodyfikuj backendową konfigurację aplikacji `backend/config/main.php` jak poniżej:

```
'components' => [
    'request' => [
        'csrfParam' => '_backendCSRF',
        'csrfCookie' => [
            'httpOnly' => true,
            'path' => '/admin',
        ],
    ],
    'user' => [
        'identityCookie' => [
            'name' => '_backendIdentity',
            'path' => '/admin',
            'httpOnly' => true,
        ],
    ],
    'session' => [
        'name' => 'BACKENDSESSID',
        'cookieParams' => [
            'path' => '/admin',
        ],
    ],
],
```

14.4 Silniki szablonów

Yii domyślnie używa PHP jako języka szablonów, ale nic nie stoi na przeszkodzie, aby skonfigurować wsparcie dla innych silników renderujących widok, takich jak Twig[7] lub Smarty[8], dostępnych w postaci rozszerzeń.

Komponent `view` jest odpowiedzialny za renderowanie widoków. Aby dodać niestandardowy silnik szablonów, należy skonfigurować komponent jak poniżej:

```
[
    'components' => [
        'view' => [
            'class' => 'yii\web\View',
            'renderers' => [
                'tpl' => [
                    'class' => 'yii\smarty\ViewRenderer',
```

[7] https://twig.symfony.com/
[8] https://www.smarty.net/

```
                    //'cachePath' => '@runtime/Smarty/cache',
                ],
                'twig' => [
                    'class' => 'yii\twig\ViewRenderer',
                    'cachePath' => '@runtime/Twig/cache',
                    // Tablica ustawień twig:
                    'options' => [
                        'auto_reload' => true,
                    ],
                    'globals' => ['html' => '\yii\helpers\Html'],
                    'uses' => ['yii\bootstrap'],
                ],
                // ...
            ],
        ],
    ],
]
```

W powyższym przykładzie zarówno Smarty jak i Twig są gotowe do użycia w plikach widoków. Aby dodać te rozszerzenia w projekcie, należy zmodyfikować dodatkowo plik `composer.json` poprzez dopisanie w wymaganiach (`require`):

```
"yiisoft/yii2-smarty": "~2.0.0",
"yiisoft/yii2-twig": "~2.0.0",
```

Po zapisaniu pliku można zainstalować rozszerzenia uruchamiając komendę `composer update --prefer-dist` z konsoli.

Szczegóły na temat każdego z powyższych silników szablonów dostępne są w ich dokumentacjach:

- Przewodnik po Twig[9]
- Przewodnik po Smarty[10]

[9] https://github.com/yiisoft/yii2-twig/tree/master/docs/guide
[10] https://github.com/yiisoft/yii2-smarty/tree/master/docs/guide

Error: not existing file: tutorial-yii-integration.md

14.5 Używanie Yii jako mikroframeworka

Yii może być z powodzeniem wykorzystywane bez dodatkowych funkcjonalności dostarczanych przez prosty i zaawansowany szablon aplikacji. Inaczej mówiąc, Yii już jest samo w sobie mikroframeworkiem. Do pracy z Yii nie jest wymagane, aby struktura folderów była dokładnie taka, jak pokazana w szablonach.

Jest to szczególnie korzystne, kiedy nie potrzebujesz gotowego kodu szablonów, jak w przypadku assetów luc widoków. Jednym z takich przypadków jest budowa JSON API. W tej sekcji pokażemy jak to zrobić.

14.5.1 Instalacja Yii

Stwórz folder dla plików swojego projektu i ustaw go jako aktywną ścieżkę. Komendy używane w przykładach oparte są na składni UNIXowej, ale podobne dostępne są również w Windows.

```
mkdir micro-app
cd micro-app
```

> **Uwaga:** minimalna wiedza na temat użytkowania Composera jest wymagana w celu kontynuacji. Jeśli nie wiesz, jak używać Composera, prosimy o zapoznanie się najpierw z Przewodnikiem po Composerze[11].

Stwórz plik `composer.json` w folderze `micro-app`, używając swojego ulubionego edytora i dodaj co następuje:

```
{
    "require": {
        "yiisoft/yii2": "~2.0.0"
    },
    "repositories": [
        {
            "type": "composer",
            "url": "https://asset-packagist.org"
        }
    ]
}
```

Zapisz plik i uruchom komendę `composer install`. Dzięki temu zainstalujesz framework i wszystkie jego zależności.

[11] https://getcomposer.org/doc/00-intro.md

14.5.2 Tworzenie struktury projektu

Po zainstalowaniu frameworka, czas na utworzenie punktu wejścia dla aplikacji. Punkt wejścia to pierwszy plik, który będzie uruchamiany, podczas startu aplikacji. Ze względów bezpieczeństwa, zalecane jest, aby plik punktu wejścia umieścić w osobnym folderze, który będzie ustawiony jako bazowy folder aplikacji.

Stwórz folder `web` i umieść w nim plik `index.php` z następującą zawartością:

```
<?php

// zakomentuj poniższe dwie linie przy wydaniu aplikacji na środowisku
produkcyjnym
defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

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

$config = require __DIR__ . '/../config.php';
(new yii\web\Application($config))->run();
```

Stwórz również plik `config.php`, który będzie zawierał całą konfigurację aplikacji:

```
<?php
return [
    'id' => 'micro-app',
    // ścieżką bazową aplikacji będzie folder `micro-app`
    'basePath' => __DIR__,
    // w tym miejscu określamy, gdzie aplikacja ma szukać wszystkich
    kontrolerów
    'controllerNamespace' => 'micro\controllers',
    // ustawiamy alias, aby umożliwić auto␣ladowanie klas z przestrzeni nazw
    'micro'
    'aliases' => [
        '@micro' => __DIR__,
    ],
];
```

> **Informacja:** Pomimo że konfiguracja mogłaby być przechowywana w pliku `index.php`, zalecane jest, aby zapisana była osobno. Dzięki temu może być również wykorzystywana dla aplikacji konsolowej, jak pokazano to poniżej.

Twój projekt jest już gotowy do rozpoczęcia kodowania. Od Ciebie również zależy struktura jego folderów, dopóki jak będziesz pamiętać o poprawnych przestrzeniach nazw.

14.5.3 Tworzenie pierwszego kontrolera

Stwórz folder `controllers` i dodaj w nim plik `SiteController.php`, który będzie domyślnym kontrolerem obsługującym żądania bez wskazanej wyraźnie ścieżki.

```
<?php

namespace micro\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actionIndex()
    {
        return 'Hello World!';
    }
}
```

Jeśli chcesz użyć innej nazwy dla tego kontrolera, nie krępuj się - musisz jedynie skonfigurować odpowiednio `yii\base\Application::$defaultRoute`. Dla przykładu, jeśli chcesz go zmienić na `DefaultController`, ustaw `'defaultRoute' => 'default/index'` w konfiguracji.

W tym momencie struktura projektu powinna wyglądać jak poniżej:

```
micro-app/
|- composer.json
|- web/
    |- index.php
|- controllers/
    |- SiteController.php
```

Jeśli nie ustawiłeś jeszcze serwera web, być może zechcesz zerknąć na pliki przykładów konfiguracji serwera web. Inną opcją jest skorzystanie z komendy `yii serve`, która użyje wbudowanego w PHP serwera web. Możesz uruchomić ją z poziomu folderu `micro-app/` za pomocą:

```
vendor/bin/yii serve --docroot=./web
```

Uruchomienie adresu URL aplikacji w przeglądarce powinno zaowocować teraz komunikatem "Hello World!", który jest zwracany w `SiteController::actionIndex()`.

> **Informacja:** W naszym przykładzie zmieniliśmy domyślną przestrzeń nazw aplikacji `app` na `micro`, aby zademonstrować, że nie ma potrzeby być ograniczonym przez tę nazwę (w przypadku, gdyby ktoś myślał, że jednak jest). Po zmianie jej na inną, należy jedynie zmodyfikować odpowiednio `przestrzeń nazw kontrolerów` i ustawić właściwy alias.

14.5.4 Tworzenie API REST

Aby zademonstrować, jak korzystać z naszego "mikroframeworka", stworzymy proste API REST dla postów.

Aby API mogło zwrócić jakieś dane, najpierw potrzebujemy ich bazy. Dodaj konfigurację połączenia z bazą danych do konfiguracji aplikacji:

```
'components' => [
    'db' => [
        'class' => 'yii\db\Connection',
        'dsn' => 'sqlite:@micro/database.sqlite',
    ],
],
```

> **Informacja:** Używamy w tym przykładzie bazy danych sqlite dla uproszczenia. Aby zapoznać się z innymi opcjami, przejdź do przewodnika po bazach danych.

Następnie tworzymy migrację bazodanową, aby skonstruować tabelę postów. Upewnij się, że posiadasz oddzielny plik konfiguracji, jak zostało to opisane powyżej, ponieważ musimy teraz uruchomić komendę konsolową, jak poniżej. Uruchomienie tych komend utworzy plik migracji i wprowadzi migrację do bazy danych:

```
vendor/bin/yii migrate/create --appconfig=config.php create_post_table
--fields="title:string,body:text"
vendor/bin/yii migrate/up --appconfig=config.php
```

Stwórz folder `models` i plik `Post.php` w tym folderze. Poniżej znajdziesz kod dla modelu:

```
<?php

namespace micro\models;

use yii\db\ActiveRecord;

class Post extends ActiveRecord
{
    public static function tableName()
    {
        return '{{posts}}';
    }
}
```

> **Informacja:** Tak utworzony model jest klasą ActiveRecord, która reprezentuje dane z tabeli `posts`. Zapoznaj się z przewodnikiem po active record, aby uzyskać więcej informacji.

Aby obsłużyć posty w naszym API, dodaj `PostController` w `controllers`:

```
<?php

namespace micro\controllers;

use yii\rest\ActiveController;

class PostController extends ActiveController
{
    public $modelClass = 'micro\models\Post';

    public function behaviors()
    {
        // wy␣lącz rateLimiter, który do pracy wymaga, aby użytkownik by␣l
        zalogowany
        $behaviors = parent::behaviors();
        unset($behaviors['rateLimiter']);
        return $behaviors;
    }
}
```

W tym momencie nasze API obsługuje już następujące adresy URL:

- `/index.php?r=post` - wyświetla listę wszystkich postów
- `/index.php?r=post/view&id=1` - wyświetla post o ID 1
- `/index.php?r=post/create` - tworzy post
- `/index.php?r=post/update&id=1` - aktualizuje post o ID 1
- `/index.php?r=post/delete&id=1` - usuwa post o ID 1

Zapoznaj się z poniższymi wskazówkami, które pomogą Ci w dalszym rozwijaniu Twojej aplikacji:

- Aktualnie API rozpoznaje jedynie urlenkodowane dane formularza na wejściu - aby zmienić je w prawdziwe JSON API, musisz skonfigurować `yii\web\JsonParser`.
- Aby uczynić adresy URL, bardziej przyjaznymi dla użytkownika, musisz skonfigurować ruting. Zobacz przewodnik po rutingu REST, który wyjaśnia, jak to zrobić.
- Dodatkowo przeczytaj też sekcję Dalsze kroki, która podpowie jak zaplanować rozwój projektu.

Rozdział 15

Widżety

Rozdział 16

Klasy pomocnicze

16.1 Klasy pomocnicze

> **Uwaga:** Ta sekcja jest w trakcie tworzenia.

Yii jest wyposażone w wiele klas upraszczających pisanie często wykorzystywanych zadań w kodzie, takich jak manipulowanie ciągami znaków bądź tablicami, generowanie kodu HTML, itp. Te pomocnicze klasy znajdują się w przestrzeni nazw `yii\helpers` i wszystkie są klasami statycznymi (czyli zawierają wyłącznie statyczne właściwości i nie powinny być tworzone ich instancje).

Aby skorzystać z klasy pomocnicznej, należy bezpośrednio wywołać jedną z jej statycznych metod, jak w przykładzie poniżej:

```
use yii\helpers\Html;

echo Html::encode('Test > test');
```

> **Uwaga:** W celu zapewnienia możliwości dostosowania klas pomocniczych do własnych potrzeb, Yii rozdziela każdą z ich wbudowanych wersji na dwie klasy: podstawę (np. `BaseArrayHelper`) i klasę właściwą (np. `ArrayHelper`). Kiedy chcesz użyć klasy pomocnicznej, powinieneś korzystać wyłącznie z jej właściwej wersji i nigdy nie używać bezpośrednio podstawy.

16.1.1 Wbudowane klasy pomocnicze

Poniższe wbudowane klasy pomocnicze dostępne są w każdym wydaniu Yii:

- ArrayHelper
- Console
- FileHelper
- FormatConverter
- Html

- HtmlPurifier
- Imagine (poprzez rozszerzenie yii2-imagine)
- Inflector
- Json
- Markdown
- StringHelper
- Url
- VarDumper

16.1.2 Dostosowywanie klas pomocniczych do własnych potrzeb

Aby zmodyfikować wbudowaną klasę pomocniczną (np. `ArrayHelper`), należy stworzyć nową klasę rozszerzającą odpowiednią podstawę (np. `BaseArrayHelper`) i nazwać ją identycznie jak jej wersja właściwa (np. `ArrayHelper`), łącznie z zachowaniem jej przestrzeni nazw. Ta klasa może następnie zostać użyta do zastąpienia oryginalnej implementacji we frameworku.

Poniższy przykład ilustruje w jaki sposób zmodyfikować metodę `merge()` klasy `ArrayHelper`:

```
<?php

namespace yii\helpers;

class ArrayHelper extends BaseArrayHelper
{
    public static function merge($a, $b)
    {
        // zmodyfikowana wersja metody
    }
}
```

Klasę należy zapisać w pliku o nazwie `ArrayHelper.php`, który może znajdować się w dowolnym odpowiednim folderze, np. `@app/components`.

Następnie dopisujemy poniższą linijkę kodu w skrypcie wejściowym aplikacji po fragmencie dołączającym plik `yii.php`, dzięki czemu autoloader klas Yii załaduje zmodyfikowaną wersję klasy pomocniczej zamiast oryginalnej:

```
Yii::$classMap['yii\helpers\ArrayHelper'] =
'@app/components/ArrayHelper.php';
```

Należy pamiętać o tym, że modyfikowanie klasy pomocniczej jest użyteczne tylko w przypadku, gdy chcemy zmienić domyślny sposób działania jej metody. W przypadku dodawania do aplikacji dodatkowych funkcjonalności, lepszym pomysłem jest stworzenie całkowicie nowej, osobnej klasy pomocniczej.

Error: not existing file: helper-array.md

Error: not existing file: helper-html.md

Error: not existing file: helper-json.md

Error: not existing file: helper-url.md

Rozdział 17

Uwagi do polskiego tłumaczenia przewodnika