Kontrolery w Symfony

v3.1

Plan

- Kontroler - podstawy użycia
- Podstawowy Routing – uzyskany przy pomocy adnotacji
- Kontroler – bardziej zaawansowane użycie
- Routing – konfiguracja dla całej aplikacji

Kontroler

podstawy użycia

Kontroler

- Kontroler jest podstawową klasą, którą będziemy implementować, pisząc projekty w Symfony.
- Jego zadaniem jest otrzymanie zapytania HTTP i wygenerowanie odpowiedzi.
- Kontrolery powinny implementować całą logikę biznesową naszej aplikacji (jeden kontroler zazwyczaj przypada na jeden model ale nie jest to regułą).

Generowanie kontrolera

Kontroler możemy w łatwy sposób wygenerować za pomocą konsoli:

```
php app/console generate:controller
```

Następnie generator przeprowadza nas przez proces tworzenia nowego kontrolera. Musimy podać:

- **nazwę kontrolera** (poprzedzoną nazwą Bundla, do którego zostanie dodany),
- **format routingu** (wybieramy adnotacje),
- **format templatów** (wybieramy Twig),
- **możemy dodawać akcje** (na razie nie dodajemy).

Nazwa kontrolera

Nazwa kontrolera powinna spełniać następujące warunki:

- kończyć się słowem **`Controller`** (jeżeli używamy generatora, to on sam doda je na koniec),
- używać **`camelCase`**,
- składać się z maksymalnie czterech członów,
- mieć taką samą bazę (początek nazwy) jak model, którym będzie sterował.

Kontroler

Generowanie kontrolera

Po wygenerowaniu kontrolera pojawił się nam nowy plik, który znajduje się w tej lokalizacji:

/src/nazwaBundla/Controller

Każdy kontroler musi znajdować się w osobnym pliku!

Namespace kontrolera

```
namespace Bundle_Name\Controller;
```

Jeżeli nie dodamy **namespace** do kontrolera, to cała aplikacja przestanie nam działać.

Każdy kontroler musi znajdować się w odpowiednim namespace!

Kontroler

Generowanie kontrolera

Po wygenerowaniu kontrolera pojawił się nam nowy plik, który znajduje się w tej lokalizacji:

/src/nazwaBundla/Controller

Każdy kontroler musi znajdować się w osobnym pliku!

Namespace kontrolera

```
namespace Bundle_Name\Controller;
```

Ta linia musi znajdować się na początku pliku z kontrolerem. Oczywiście wpisujemy swój Bundle.

Jeżeli nie dodamy **namespace** do kontrolera, to cała aplikacja przestanie nam działać.

Każdy kontroler musi znajdować się w odpowiednim namespace!

Klasa bazowa

- Kontroler w Symfony powinien (ale nie musi) dziedziczyć po klasie bazowej o nazwie **`Controller`**.
- Dzięki niej mamy dostęp do wielu wbudowanych funkcjonalności, które będą nam przydatne.
- Pamiętaj o jej dodaniu, jeżeli piszesz klasę sam (generator doda ją automatycznie).

Akcje kontrolera

Akcje są metodami, które będą wywoływane przez Symfony w następujących przypadkach:

- gdy jakiś użytkownik będzie chciał wejść na przypisany do nich adres URL,
- gdy zostaną wywołane przez inną akcję.

Zazwyczaj jedna akcja skupia się na jednej funkcjonalności. Dla kontrolera użytkowników to:

- stworzenie nowego użytkownika,
- wczytanie wszystkich użytkowników,
- wczytanie użytkownika o podanym **`id`**.

Akcje muszą spełniać następujące założenia:

- być publiczne,
- kończyć się słowem **`Action`**,
- zwracać obiekt typu **`Response`** (bardzo ważne!)

Symfony samo przeskanuje nasz plik i wczyta wszystkie akcje.

Zwracanie obiektu Response

- Akcja – tak jak wcześniej mówiliśmy – musi zwracać obiekt typu **Response**.
- Najprostszy obiekty **Response** do konstruktora przyjmuje string reprezentujący kod HTML strony, która ma być wyświetlona.
- Niebawem dowiemy się, jak generować bardziej skomplikowane odpowiedzi, i jak używać szablonów **Twig** do generowania odpowiedzi.

Zwracanie obiektu Response

Żeby zwrócić prosty obiekt **Response**, musimy najpierw wczytać jego klasę

```
use Symfony\Component\HttpFoundation\Response
```

Następnie w akcji:

```
return new Response('<html><body>Hello World!</body></html>');
```

Prosty routing

Musimy jeszcze wskazać Symfony, do jakiego URL ma być przypisana nasza akcja.

Służy do tego adnotacja **@Route("url")**, którą trzeba najpierw zaimportować:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route
```

Adnotacja musi znajdować się bezpośrednio przed akcją!

```
/**
 * @Route("/helloWorld/")
 */
```

Dodając znak "/" na końcu ścieżki Symfony poprawnie zinterpretuje adres zarówno z jak i bez "/" wpisany w przeglądarce.

Przykładowa akcja

Przykładowa akcja powinna wyglądać tak:

```
/**
 * @Route("/helloWorld/")
 */
public function helloWorldAction()
{
    return new Response('<html><body>Hello World!</body></html>');
}
```

Przykładowy kontroler

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
    * @Route("/helloWorld/")
    */
    public function helloWorldAction()
    {
        return new Response('<html><body>Hello World!</body></html>');
    }
}
```

Przykładowy kontroler

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
    * @Route("/helloWorld/")
    */
    public function helloWorldAction()
    {
        return new Response('<html><body>Hello World!</body></html>');
    }
}
```

Import niezbędnych klas

Przykładowy kontroler

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
    * @Route("/helloWorld/")
    */
    public function helloWorldAction()
    {
        return new Response('<html><body>Hello World!</body></html>');
    }
}
```

Definicja kontrolera

Przykładowy kontroler

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
    * @Route("/helloWorld/")
    */
    public function helloWorldAction()
    {
        return new Response('<html><body>Hello World!</body></html>');
    }
}
```

Adnotacja do akcji

Przykładowy kontroler

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
    * @Route("/helloWorld/")
    */
    public function helloWorldAction()
    {
        return new Response('<html><body>Hello World!</body></html>');
    }
}
```

Definicja akcji w kontrolerze

Przykładowy kontroler

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
    * @Route("/helloWorld/")
    */
    public function helloWorldAction()
    {
        return new Response('<html><body>Hello World!</body></html>');
    }
}
```

Zwrócenie obiektu **Response** z akcji

Zadania

Podstawowy Routing

Uzyskany przy pomocy adnotacji

Przypisywanie URL do akcji

Wiemy już, jak przypisać podstawowy URL do danej akcji.
Robimy to za pomocą adnotacji **`@Route("url")`**.

Adnotacja ta ma więcej zastosowań:

- może przyjmować parametry do akcji (coś podobnego do **`GET`**),
- nadawać akcji alias (**`name`**), dzięki któremu będziemy mogli się do niej odnosić z innych części kodu.

Nadawanie akcjom aliasów

Adnotacja **@Route** może przyjmować jeszcze parametr, który nada nam alias danej akcji.

```
/**
 * @Route("/hello", name="hello")
 */
```

Jeżeli go nie podamy, to nazwa akcji budowana jest na zasadzie:

NazwaBundla_NazwaKontrolera_NazwaAkcji

Przekazywanie parametrów do akcji

Jednym z najpotężniejszych mechanizmów wprowadzonych przez URI (czyli nowy system adresów) jest możliwość łatwego i intuicyjnego wprowadzania parametrów. Np.:

- www.mypage/post/4 – powinno nas przekierować do postu o id 4,
- www.mypage/post/6 – powinno nas przekierować do postu o id 6.

Przekazywanie parametrów do akcji

Dzięki podaniu parametru w ścieżce URL, czyli tzw. sluga, możemy łatwo przenieść część informacji jako parametr do naszej akcji.

```
/**
* @Route("/helloWorld/{sentence}")
*/
public function helloWorldAction($sentence)
{
    return new Response(
        "<html><body>Hello World! " +
        "Your sentence is : $sentence</body></html>"
    );
}
```

Tworzenie slugów

Slugi w adresie URL przekazanym do **`@Route`** muszą spełniać następujące zasady:

- być otoczone nawiasami klamrowymi **`{}`**,
- mieć taką samą nazwę jak parametr naszej akcji (ale bez znaku **`$`**),
- oddzielone przy pomocy znaku ukośnika (`/`) lub kropki (`.`),
- musi ich być co najmniej tyle samo co parametrów akcji (możemy mieć nieużywane slugi).

Przykłady użycia slugów

```php
/**
* @Route("/helloWorld/{sentence}")
*/
public function helloWorldAction($sentence){
    return new Response(
        "<html><body>Hello World! " +
        "Your sentence is : $sentence</body></html>"
    );
}
```

```php
/**
 * @Route("/hello/{userName}/{userSurname}")
 */
public function helloAction($userName, $userSurname){
    return new Response(
        "<html><body>Welcome $userName $userSurname</body></html>"
    );
}
```

Wartości domyśle slugów

- Jeżeli używamy slugów, to muszą one być wypełnione, inaczej routing nie dopasuje naszej ścieżki do danej akcji.
- Dla routingu Symfony adres **`/helloWord/2`** jest innym adresem niż **`/helloWord/`**
- Jeżeli chcemy żeby slug miał przypisaną jakąś domyślną wartość, to musimy użyć adnotacji **`defaults`** podanej w prawej kolumnie.

```
@Route("/helloWorld/{sentence}",
defaults={"sentence" = "Default sentence"})

@Route("/post/{id}", defaults={"id" = 1})

@Route("/post/{categoryId}/{postId}",
defaults={"postId" = 1, "categoryId" = 10})
```

Wymagania dotyczące slugów

Możemy też dać wymagania dotyczące slugów, na przykład:

- chcemy, żeby **`id`** było wartością liczbową,
- chcemy, żeby **`userName`** był napisem,

Robimy to przez rozszerzenie naszej adnotacji o człon **`requirements:`**

```
@Route("/hello/{userName}", requirements={"userName"="[a-zA-Z]+"})

@Route("/post/{id}", requirements={"id"="\d+"})
```

Wymagania dotyczące slugów

Możemy też dać wymagania dotyczące slugów, na przykład:

- chcemy, żeby **`id`** było wartością liczbową,
- chcemy, żeby **`userName`** był napisem,

Robimy to przez rozszerzenie naszej adnotacji o człon **`requirements:`**

```
@Route("/hello/{userName}", requirements={"userName"="[a-zA-Z]+"})

@Route("/post/{id}", requirements={"id"="\d+"})
```

- **`userName`** musi składać się z liter alfabetu.

Wymagania dotyczące slugów

Możemy też dać wymagania dotyczące slugów, na przykład:

- chcemy, żeby **`id`** było wartością liczbową,
- chcemy, żeby **`userName`** był napisem,

Robimy to przez rozszerzenie naszej adnotacji o człon **`requirements:`**

```
@Route("/hello/{userName}", requirements={"userName"="[a-zA-Z]+"})

@Route("/post/{id}", requirements={"id"="\d+"})
```

- **`userName`** musi składać się z liter alfabetu.
- **`id`** musi składać się z cyfr.

Wymagania dotyczące slugów

Do napisania wymagań używamy wyrażeń regularnych. Najpopularniejsze z nich to:

\d+	wartość numeryczna
[a-zA-Z]+	wartość alfabetyczna (tylko litery)
(fr|en|pl)	wartość **fr**, **en** lub **pl**

Więcej o regexpach w PHP możecie przeczytać tutaj:

- http://webcheatsheet.com/php/regular_expressions.php

Przypisanie metody HTTP

- Nasza akcja może być przypisana tylko do wybranej (lub kilku wybranych) metody HTTP. Używamy do tego adnotacji **`@Method()`**
- Jeżeli nie podamy tej adnotacji, to akcja jest przypisana do wszystkich metod!

```
/**
 * @Route("/posts/{id}")
 * @Method("GET")
 */

/**
 * @Route("/users/{id}")
 * @Method({"GET","POST"})
 */
```

Przypisanie metody HTTP

- Nasza akcja może być przypisana tylko do wybranej (lub kilku wybranych) metody HTTP. Używamy do tego adnotacji **`@Method()`**
- Jeżeli nie podamy tej adnotacji, to akcja jest przypisana do wszystkich metod!

```
/**
 * @Route("/posts/{id}")
 * @Method("GET")
 */

/**
 * @Route("/users/{id}")
 * @Method({"GET","POST"})
 */
```

Jeżeli podajemy więcej niż jeden argument, to używajmy zapisu podobnego do tablicy w PHP

Przypisanie metody HTTP

Dzięki temu możemy napisać osobne akcje do różnych metod tego samego adresu (nie musimy już tego ręcznie sprawdzać).

```
/**
 * @Route("/posts/{id}")
 * @Method("GET")
 */

/**
 * @Route("/posts/{id}")
 * @Method("POST")
 */
```

Przedrostek dla kontrolera

- Jeżeli wszystkie akcje w danym kontrolerze mają taki sam przedrostek, możemy go dodać jako adnotacje do całej klasy kontrolera.
- Adnotacja musi znajdować się wtedy bezpośrednio przed klasą.
- Adnotacja ta może zawierać slugi (musimy wtedy dodać ten atrybut do wszystkich akcji tego kontrolera).

```php
/**
 * @Route("/user/{id}")
 */

class UserController extends Controller
{
    /**
     * @Route("/")
     */
    public function showAction($id)
    {
        /* ... */
    }
}
```

Zadania

Kontroler

Bardziej zaawansowane użycie

Obiekt Request

Do naszej akcji może być przekazywany obiekt typu **`Request.`** Obiekt ten reprezentuje zapytanie HTTP wysłane do naszego serwera.

Obiekt ten trzyma w sobie dane na temat:

- GET,
- POST,
- Ciasteczek,
- Sesji.

Przekazywanie obiektu Request do akcji

Aby przekazać obiekt **Request** do naszej akcji, musi być on **pierwszym parametrem** tej metody i mieć wymuszony typ obiektu. Musimy oczywiście zaimportować klasę:

```
use Symfony\Component\HttpFoundation\Request;

public function helloWorldAction(Request $req, $slug1, $slug2){
}
```

Obiekt Response

- Jedynym wymaganiem akcji jest zwrócenie z siebie obiektu klasy **Response**.
- Jeżeli tego nie zrobimy, Symfony zwróci błąd.
- Obiekt **Response** jest abstrakcją nad zapisem tekstowym, który jest odsyłany do przeglądarki użytkownika.

Obiekt Response

Istnieją różne klasy dziedziczące po klasie **Response**, które możemy zwracać:

- **Response** – (klasa bazowa) podajemy jej napis reprezentujący kod HTML,
- **JsonResponse** – klasa reprezentująca zwracanie danych JSON (http://symfony.com/doc/2.8/components/http_foundation.html#creating-a-json-response),
- **BinaryFileResponse** – klasa reprezentująca zwracanie plików (http://symfony.com/doc/2.8/components/http_foundation.html#serving-files),
- **Templatki.** - np. szablony **Twig**

Dostęp do $_GET

Żeby otrzymać dane z GET-a, powinniśmy skorzystać z obiektu **Request**, który przekazaliśmy do naszej akcji.

Przy używaniu ścieżek typu URI, nie powinniśmy używać parametrów GET.

- Powinniśmy korzystać ze **slugów**.

Istnieje możliwość pobrania parametru GET z użyciem obiektu **Request**

Służy do tego następująca metoda:

```
public function someAction(Request $request)
{
    $request->query->get('name');
}
```

Przykładowy adres mógłby wyglądać dla tej akcji

/helloWorld/?name=Aleks

Dostęp do $_GET

Żeby otrzymać dane z GET-a, powinniśmy skorzystać z obiektu **Request**, który przekazaliśmy do naszej akcji.

Przy używaniu ścieżek typu URI, nie powinniśmy używać parametrów GET.

- Powinniśmy korzystać ze **slugów**.

Istnieje możliwość pobrania parametru GET z użyciem obiektu **Request**

Służy do tego następująca metoda:

```
public function someAction(Request $request)
{
    $request->query->get('name');
}
```

$_GET['name']

Przykładowy adres mógłby wyglądać dla tej akcji

/helloWorld/?name=Aleks

Dostęp do $_POST

Żeby otrzymać dane z POST-a, powinniśmy skorzystać z obiektu **Request**, który przekazaliśmy do naszej akcji

Służy do tego następująca metoda:

```php
public function someAction(Request $request)
{
    $request->request->get('page');
}
```

Dostęp do $_POST

Żeby otrzymać dane z POST-a, powinniśmy skorzystać z obiektu **Request**, który przekazaliśmy do naszej akcji

Służy do tego następująca metoda:

```
public function someAction(Request $request)
{
    $request->request->get('page');
}
```

$_POST['page']

Dostęp do sesji

Do zarządzania sesją istnieje osobny obiekt, który trzymany jest w naszym obiekcie **`Request`**.
Możemy go wczytać za pomocą:

```
public function someAction(Request $request)
{
    $session = $request->getSession();
    /* ... */
}
```

Nie musimy się już przejmować startowaniem sesji. Symfony robi to za nas.

Zapisywanie danych do sesji

Gdy mamy obiekt sesji, możemy łatwo zapisać do niego dane.

```
$session = $request->getSession();
$session->set('foo', 'bar');
```

Zapisywanie danych do sesji

Gdy mamy obiekt sesji, możemy łatwo zapisać do niego dane.

```
$session = $request->getSession();
$session->set('foo', 'bar');
```

Podajemy nazwę jako napis, a następnie podajemy wartość.

Wczytywanie danych z sesji

Gdy mamy obiekt sesji, możemy również łatwo wczytać z niego dane:

```
$session = $request->getSession();
$fooSession = $session->get('foo');
```

Możemy też podać wartość domyślną.

```
$session = $request->getSession();
$fooSession = $session->get('foo', "default_value");
```

Wczytywanie danych z sesji

Gdy mamy obiekt sesji, możemy również łatwo wczytać z niego dane:

```
$session = $request->getSession();
$fooSession = $session->get('foo');
```

Podajemy nazwę jako napis

Możemy też podać wartość domyślną.

```
$session = $request->getSession();
$fooSession = $session->get('foo', "default_value");
```

Wczytywanie danych z sesji

Gdy mamy obiekt sesji, możemy również łatwo wczytać z niego dane:

```
$session = $request->getSession();
$fooSession = $session->get('foo');
```

Podajemy nazwę jako napis

Możemy też podać wartość domyślną.

```
$session = $request->getSession();
$fooSession = $session->get('foo', "default_value");
```

Jeżeli w sesji nie ma klucza **foo**, to zostanie nam zwrócony podany napis

Dostęp do ciasteczek

Ciasteczka są reprezentowane przez specjalną klasę **`Cookie`**.

Żeby używać ciasteczek, musimy załączyć klasę:

```
use Symfony\Component\HttpFoundation\Cookie;
```

W kolejnym kroku możemy stworzyć ciasteczko:

```
$cookie = new Cookie("cookieName", $value, time() + (3600 * 48));
```

Dostęp do ciasteczek

Ciasteczka są reprezentowane przez specjalną klasę **`Cookie`**.

Żeby używać ciasteczek, musimy załączyć klasę:

```
use Symfony\Component\HttpFoundation\Cookie;
```

W kolejnym kroku możemy stworzyć ciasteczko:

```
$cookie = new Cookie("cookieName", $value, time() + (3600 * 48));
```

Podajemy w kolejności:

- nazwę ciasteczka,
- wartość, jaką ma przyjąć,
- czas życia ciasteczka.

Zapamiętywanie ciasteczek

Aby zapamiętać ciasteczko, musimy je dodać do obiektu **Response**, który będziemy zwracać z naszej akcji.

```
$cookie = new Cookie("cookieName", $value, time() + (3600 * 48));
$resp = new Response('<div>Hello Cookie</div>');
$resp->headers->setCookie($cookie);

return $resp;
```

Wczytywanie ciasteczek

Żeby wczytać ciasteczka musimy odwołać się do obiektu **`Request:`**

Dostaniemy tablicę z wszystkimi ciasteczkami, gdzie klucz to nazwa ciasteczka, a zawartość to wartość ciasteczka.

```
public function helloAction(Request $req) {
    $cookies = $req->cookies->all();
    $cookieValue = $cookies["cookieName"];
    /* ... */
}
```

Przekierowywanie do innej akcji

Kontroler potrafi w łatwy sposób przekierować do innej akcji. Służy do tego następująca metoda:

```
$response = $this->redirect($url);
```

Przy czym zmienna **$url** reprezentuje wygenerowaną ścieżkę dostępu:

```
$url = $this->generateUrl('homepage');
```

Metoda ta zwraca obiekt **Response**, który następnie trzeba zwrócić z naszej akcji. Można tez skrócić zapis i użyć:

```
return $this->redirect($url);
```

Jeśli akcja, do której przekierowujemy posada slug'i to używając metody **generateUrl** również możemy przekazać dane, o czym w kolejnych slajdach.

Przekierowywanie do innej akcji

Kontroler potrafi w łatwy sposób przekierować do innej akcji. Służy do tego następująca metoda:

```
$response = $this->redirect($url);
```

Przy czym zmienna **$url** reprezentuje wygenerowaną ścieżkę dostępu:

```
$url = $this->generateUrl('homepage');
```

Tworzy link do akcji o nazwie **homepage**

Metoda ta zwraca obiekt **Response**, który następnie trzeba zwrócić z naszej akcji. Można tez skrócić zapis i użyć:

```
return $this->redirect($url);
```

Jeśli akcja, do której przekierowujemy posada slug'i to używając metody **generateUrl** również możemy przekazać dane, o czym w kolejnych slajdach.

Przekierowywanie do innej akcji

Jako trzeci argument możemy podać kod przekierowania.
Domyślnie jest to 302 (chwilowe przeniesienie):

```
return $this->redirect($url, [], 301);
```

Przekierowywanie do innej strony

Z użyciem metody **redirect** możemy też przekierować do innej strony

```
$response = $this->redirect('http://google.com');
```

Zwraca ona obiekt **Repsonse**, który następnie trzeba zwrócić z naszej akcji. Można tez skrócić zapis i użyć:

```
return $this->redirect('http://google.com');
```

Generowanie URL w kontrolerach

- W kontrolerze mamy możliwość wygenerowania ścieżek URL do innych akcji/kontrolerów.
- Musimy znać tylko nazwę tej akcji.
- Nazwa akcji to jej alias (patrz slajd o nadawaniu aliasów) albo wygenerowana automatycznie nazwa.
- Musimy też podać wszystkie slugi, które ten adres powinien przyjąć.

Generowanie URL w kontrolerach

Do generowanie adresów URL służy nam metoda odziedziczona po kontrolerze:

```
$url = $this->generateUrl(
    'action_name',
    ['slug' => 'slug_value']
);
```

Metoda ta zwraca napis, który zawiera ścieżkę zależną (czyli bez nazwy naszej strony).

Dzięki niej podczas przekierowywania do innej akcji z użyciem metody redirect możemy skorzystać ze slug'ów .

Generowanie URL w kontrolerach

Jeżeli chcemy dostać bezwzględną ścieżkę (czyli z pełną nazwą naszego serwera), to musimy podać do metody dodatkowy argument:

```
$url = $this->generateUrl(
     'action_name',
     ['slug' => 'slug_value'],
     UrlGeneratorInterface::ABSOLUTE_URL
);
```

Zadania

Routing

Konfiguracja dla całej aplikacji

Debugowanie i wizualizacja ścieżek

W większych aplikacjach łatwo zapomnieć, jakie ścieżki odpowiadają danym kontrolerom.

Na szczęcie z pomocą przychodzi konsola Symfony:

```
php app/console debug:router
```

Komenda ta wyświetli nam wszystkie ścieżki URI, jakie są w naszej aplikacji (łącznie z nazwami kontrolerów, przypisanymi metodami, ich aliasami i slugami).

Możemy też wyświetlić wszystkie informacje o jednej ze ścieżek.

```
php app/console debug:router <nazwa ścieżki>
```

Debugowanie i wizualizacja ścieżek

Możemy też w łatwy sposób odnaleźć, jaki kontroler jest przypisany do danego URI

URI może mieć wypełnione slugi – router Symfony dopasuje odpowiednie dane.

Wystarczy wpisać:

```
php app/console router:match <URI>
```

Przykład:

```
php app/console route:match /helloWorld/2
```

Konfiguracja routingu

- Cały routing jest wczytywany z jednego pliku: **`app/config/routing.yml`**
- Jeżeli używamy YAML, XML albo PHP do tworzenia routingu, to musimy wpisywać tam, gdzie znajduje się nasz plik konfiguracyjny.
- Jeżeli używamy adnotacji, to musimy wskazać tam, gdzie znajdują się nasze kontrolery.

Dopisywanie bundla do routingu

- Każdy **`bundle`** musi mieć wybrany jeden system dodawania ścieżek – wynika to z ogólnej zasady.
- Różne **`bundle`** w jednym systemie mogą mieć różne typy definiowania ścieżek (np. **`FOSUserBundle`** będzie używał XML, poznamy go w trakcie kursu).

Dopisywanie bundla do routingu

Żeby dodać bundla do routingu, musimy wpisać tam następujący kawałek kodu (dla pliku YAML):

```
bundleName:
resource: '@bundleName/Controller/'
  type: annotation
  prefix: /my_prefix
```

- Jeżeli **`Bundle`** nie zostanie tutaj dodany, to router nie będzie o nim wiedział.
- Jeżeli podamy złą ścieżkę do **`resource`**, to cała aplikacja przestanie nam działać!

Dopisywanie bundla do routingu

Żeby dodać bundla do routingu, musimy wpisać tam następujący kawałek kodu (dla pliku YAML):

```yaml
bundleName:
resource: '@bundleName/Controller/'
  type: annotation
  prefix: /my_prefix
```

- Wpisujemy nazwę bundla bez dodawania na końcu słowa **Bundle**

- Jeżeli **Bundle** nie zostanie tutaj dodany, to router nie będzie o nim wiedział.
- Jeżeli podamy złą ścieżkę do **resource**, to cała aplikacja przestanie nam działać!

Dopisywanie bundla do routingu

Żeby dodać bundla do routingu, musimy wpisać tam następujący kawałek kodu (dla pliku YAML):

```yaml
bundleName:
resource: '@bundleName/Controller/'
  type: annotation
  prefix: /my_prefix
```

- Wpisujemy nazwę bundla bez dodawania na końcu słowa **Bundle**
- Ścieżka do naszych kontrolerów **@bundleName** musi zawierać słowo **Bundle**

- Jeżeli **Bundle** nie zostanie tutaj dodany, to router nie będzie o nim wiedział.
- Jeżeli podamy złą ścieżkę do **resource**, to cała aplikacja przestanie nam działać!

Dopisywanie bundla do routingu

Żeby dodać bundla do routingu, musimy wpisać tam następujący kawałek kodu (dla pliku YAML):

```yaml
bundleName:
resource: '@bundleName/Controller/'
  type: annotation
  prefix: /my_prefix
```

- Wpisujemy nazwę bundla bez dodawania na końcu słowa **Bundle**
- Ścieżka do naszych kontrolerów **@bundleName** musi zawierać słowo **Bundle**
- Typ używanych ścieżek: **annotation, XML, YAML** lub **PHP**

- Jeżeli **Bundle** nie zostanie tutaj dodany, to router nie będzie o nim wiedział.
- Jeżeli podamy złą ścieżkę do **resource**, to cała aplikacja przestanie nam działać!

Dopisywanie bundla do routingu

Żeby dodać bundla do routingu, musimy wpisać tam następujący kawałek kodu (dla pliku YAML):

```yaml
bundleName:
resource: '@bundleName/Controller/'
  type: annotation
  prefix: /my_prefix
```

- Wpisujemy nazwę bundla bez dodawania na końcu słowa **Bundle**
- Ścieżka do naszych kontrolerów **@bundleName** musi zawierać słowo **Bundle**
- Typ używanych ścieżek: **annotation, XML, YAML** lub **PHP**
- Możemy też dodać globalny **prefix** do całego bundla.

- Jeżeli **Bundle** nie zostanie tutaj dodany, to router nie będzie o nim wiedział.
- Jeżeli podamy złą ścieżkę do **resource**, to cała aplikacja przestanie nam działać!

Dopisywanie bundla do routingu

Przykładowy bundle dodany do pliku **routing.yml:**

```
coderslab:
    resource: "@CoderslabBundle/Controller/"
    type: annotation
    prefix: /
```