Symfony
widoki i twig

v3.1

Plan

- Widoki
- Twig
- Podstawowa składania Twiga
- Załączanie innych szablonów
- Linki i assety
- Tłumaczenia

Widoki

Widoki

Widok

- Widok jest drugą częścią modelu MVC, który poznamy.
- Ideą widoku jest oddzielenie kodu wyświetlającego informacje na stronie od kodu, który zajmuje się tworzeniem tych informacji.
- Oddzielenie widoku pozwala na łatwą podmianę wyglądu naszej aplikacji.

Szablony

- Widoki w Symfony implementujemy dzięki systemowi szablonów.
- Szablony mogą być pisane w różnych językach, ale preferowany jest **Twig**.
- Pliki **Twig** mają rozszerzenie **`.twig`**: **`nazwa_pliku.html.twig`**

Czym jest Twig?

Twig to nowoczesny silnik szablonów dla PHP.

Jest rozwijany i utrzymywany firmę SensioLabs – twórcę Symfony.

Twig jest udostępniany na licencji BSD.

Strona główna projektu:

- http://twig.sensiolabs.org

Kod źródłowy projektu:

- http://github.com/twigphp/Twig

Dlaczego Twig?

Szybki
Kompilowany do czystego, zoptymalizowanego kodu PHP.

Bezpieczny
Ma tryb sandbox, dzięki czemu umożliwia budowanie aplikacji pozwalających użytkownikowi na samodzielne tworzenie/edytowanie szablonów.

Elastyczny
Ma interfejsy pozwalające deweloperowi na definiowanie własnych tagów, filtrów, makr/funkcji, operatorów, a nawet zredefiniowanie jego domyślnej składni.

Dlaczego Symfony wybrało język Twig?

- czytelna składnia,
- zawiera uproszczenia składniowe dla typowych operacji,
- nie tylko służy do tworzenia szablonów HTML, lecz także dowolnych innych formatów tekstowych,
- zaawansowane mechanizmy, m.in. dziedziczenie, bloki, automatyczny escaping wyświetlanych danych,
- stworzone przez tą samą firmę.

Podpinanie szablonu do kontrolera

W akcji kontrolera możemy na dwa sposoby wskazać, jaki szablon ma być użyty:

- użyć metody **`render()`** należącej do bazowej klasy kontrolera,
- użyć adnotacji **`@Template()`**.

Metoda render()

- Pierwszy sposób z użyciem metody **`render()`**.
- Zwraca ona obiekt typu **Response**, który następnie musimy zwrócić ze swojej akcji.

W kontrolerze

```
public function indexAction() {
    return $this->render('hello/greetings/index.html.twig', [
        'name' => $name
    ]);
}
```

Metoda render()

Metoda **`render()`** przyjmuje dwa parametry:

- ścieżkę do pliku szablonu,
- tablicę z danymi, które mają być przekazane do szablonu (opcjonalnie).

Tablica musi – jako klucze – zawierać napisy, dzięki którym następnie w szablonie będziemy mogli odnosić się do zmiennej.

Gdzie trzymamy szablony?

Szablony możemy trzymać w dwóch miejscach:

- **app/Resources/views** – w tym miejscu powinniśmy trzymać bazowe szablony, które nadają wygląd całej naszej aplikacji,
- **path_to_bundle/Resources/views** – tutaj powinniśmy trzymać szablony specyficzne dla naszego bundla.

W katalogach powyższych tworzymy analogiczne katalogi do nazw kontrolera, którego widok chcemy nadpisać dla przykładu:

app/Resources/views/authors/index.html.twig

Ścieżka do szablonu 1/2

Do szablonu odnosimy się różnie, zależnie od miejsca trzymania naszego szablonu.

Przez ścieżkę pliku odnosimy się do szablonów znajdujących się w następującej lokalizacji:

- **app/Resources/views/**

```
$this->render('hello/greetings/index.html.twig', []);

$this->render('hello/index.html.twig', ['name' => $name]);
```

Ścieżka do szablonu 1/2

Do szablonu odnosimy się różnie, zależnie od miejsca trzymania naszego szablonu.

Przez ścieżkę pliku odnosimy się do szablonów znajdujących się w następującej lokalizacji:

- **app/Resources/views/**

```
$this->render('hello/greetings/index.html.twig', []);

$this->render('hello/index.html.twig', ['name' => $name]);
```

Odnosi się do **app/Resources/views/hello/greetings/index.html.twig**

Ścieżka do szablonu 1/2

Do szablonu odnosimy się różnie, zależnie od miejsca trzymania naszego szablonu.

Przez ścieżkę pliku odnosimy się do szablonów znajdujących się w następującej lokalizacji:

- **app/Resources/views/**

```
$this->render('hello/greetings/index.html.twig', []);

$this->render('hello/index.html.twig', ['name' => $name]);
```

Odnosi się do **app/Resources/views/hello/greetings/index.html.twig**

Odnosi się do **app/Resources/views/hello/index.html.twig** i przekazuje do niej jedną zmienną o nazwie **name**

Ścieżka do szablonu 2/2

Do plików znajdujących się w naszym bundlu odwołujemy się przez budowanie ścieżki:

- **BundleName:Blog:index.html.twig**

```
$this->render('AcmeBlogBundle:Blog:index.html.twig', []);
$this->render('AcmeBlogBundle::layout.html.twig', ['name' => $name]);
```

Ścieżka do szablonu 2/2

Do plików znajdujących się w naszym bundlu odwołujemy się przez budowanie ścieżki:

- **BundleName:Blog:index.html.twig**

```
$this->render('AcmeBlogBundle:Blog:index.html.twig', []);
$this->render('AcmeBlogBundle::layout.html.twig', ['name' => $name]);
```

Odnosi się do **AcmeBlogBundle/Resources/views/Blog/index.html.twig**

Ścieżka do szablonu 2/2

Do plików znajdujących się w naszym bundlu odwołujemy się przez budowanie ścieżki:

- **BundleName:Blog:index.html.twig**

```
$this->render('AcmeBlogBundle:Blog:index.html.twig', []);
$this->render('AcmeBlogBundle::layout.html.twig', ['name' => $name]);
```

Odnosi się do **AcmeBlogBundle/Resources/views/Blog/index.html.twig**

Odnosi się do **AcmeBlogBundle/Resources/views/layout.html.twig** i przekazuje do niej zmienną o nazwie **name**

Adnotacja @Template

Szablon możemy też wskazać przez użycie adnotacji **@Template**, którą musimy podlinkować.

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
```

Adnotacja @Template

- Następnie możemy użyć naszej adnotacji.
- Jeżeli chcemy przekazać jakieś dane do widoku, to musimy zwracać z naszej akcji tablicę z danymi (klucze muszą być napisami, dzięki którym będziemy mogli dostać się do tych zmiennych w widoku).

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Template("SensioBlogBundle:Post:show.html.twig")
 */
public function showAction($id) {
    return ['post' => $post];
}

//analogia do
public function showAction2($id){
    $this->render('SensioBlogBundle:Post:show.html.twig', ['post' => $post]);
}
```

Zadania

Podstawowa składnia Twiga

Podstawowa składnia Twiga

W plikach **Twiga** umieszczamy zwykły kod HTML.
Oprócz tego możemy używać trzech typów znaczników **Twiga**, które są wyszczególnione w tabeli.

`{{ ... }}`	Służy do wyświetlania wartości wyrażenia.
`{% ... %}`	Służy do wykonywania wyrażeń, m.in. warunki logiczne, pętle, bloki.
`{# ... #}`	Bloki komentarzy.

Twig

Przykładowy plik

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>
        <ul id="navigation">
            {% for item in navigation %}
                <li>
                    <a href="{{ item.href }}">
                    {{ item.caption }}
                    </a>
                </li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Zmienne

- Akcja przekazuje do szablonu zmienne, które następnie są dostępne z poziomu szablonu (przez tablicę).
- Zmiennymi mogą być nie tylko typy proste, lecz także tablice i obiekty.
- Jeśli zmienna z szablonu nie została przekazana, wtedy **Twig** zwraca **`null`**.

Twig

Przykładowy plik

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>
        <ul id="navigation">
            {% for item in navigation %}
                <li>
                <a href="{{ item.href }}">
                {{ item.caption }}
                </a>
                </li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Zmienna **page_title**

Zmienne

- Akcja przekazuje do szablonu zmienne, które następnie są dostępne z poziomu szablonu (przez tablicę).
- Zmiennymi mogą być nie tylko typy proste, lecz także tablice i obiekty.
- Jeśli zmienna z szablonu nie została przekazana, wtedy **Twig** zwraca **null**.

Twig

Przykładowy plik

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>
        <ul id="navigation">
            {% for item in navigation %}
                <li>
                    <a href="{{ item.href }}">
                    {{ item.caption }}
                    </a>
                </li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Tablica obiektów **navigation**

Zmienne

- Akcja przekazuje do szablonu zmienne, które następnie są dostępne z poziomu szablonu (przez tablicę).
- Zmiennymi mogą być nie tylko typy proste, lecz także tablice i obiekty.
- Jeśli zmienna z szablonu nie została przekazana, wtedy **Twig** zwraca **null**.

Twig

Przykładowy plik

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>
        <ul id="navigation">
            {% for item in navigation %}
                <li>
                    <a href="{{ item.href }}">
                    {{ item.caption }}
                    </a>
                </li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Obiekt **item**

Zmienne

- Akcja przekazuje do szablonu zmienne, które następnie są dostępne z poziomu szablonu (przez tablicę).
- Zmiennymi mogą być nie tylko typy proste, lecz także tablice i obiekty.
- Jeśli zmienna z szablonu nie została przekazana, wtedy **Twig** zwraca **null**.

Zmienne w Twigu

- W celu odwołania się do elementów w tablicach używamy nawiasów kwadratowych.
- W celu odwołania się do własności obiektów lub ich metod używamy znaku kropki.

```
{{ foo.bar }}

{{ foo2['bar'] }}
```

Zmienne w Twigu

- W celu odwołania się do elementów w tablicach używamy nawiasów kwadratowych.
- W celu odwołania się do własności obiektów lub ich metod używamy znaku kropki.

```
{{ foo.bar }}

{{ foo2['bar'] }}
```

Wyświetlenie atrybutu **bar** z obiektu **foo**

Zmienne w Twigu

- W celu odwołania się do elementów w tablicach używamy nawiasów kwadratowych.
- W celu odwołania się do własności obiektów lub ich metod używamy znaku kropki.

```
{{ foo.bar }}

{{ foo2['bar'] }}
```

Wyświetlenie atrybutu **bar** z obiektu **foo**

Wyświetlenie zawartości komórki o kluczu **bar** z tablicy **foo2**

Filtry

Zmienne w Twigu mogą być modyfikowane przez tzw. filtry.

Filtry od zmiennych oddzielamy znakiem `|`. Można stosować wiele filtrów dla zmiennej, zostaną one nałożone w kolejności.

```
{{ variable | escape | title }}
```

Filtr można też nałożyć na blok kodu, stosując tag **`filter`**:

```
{% filter upper %}
    This text becomes uppercase
{% endfilter %}
```

Filtry

Zmienne w Twigu mogą być modyfikowane przez tzw. filtry.

Filtry od zmiennych oddzielamy znakiem **|**. Można stosować wiele filtrów dla zmiennej, zostaną one nałożone w kolejności.

```
{{ variable | escape | title }}
```

Spowoduje nałożenie na zmienną **variable**, dwóch filtrów **escape** oraz **title**

Filtr można też nałożyć na blok kodu, stosując tag **filter**:

```
{% filter upper %}
    This text becomes uppercase
{% endfilter %}
```

Filtry wbudowane w Twig

- **abs,**
- **batch,**
- **capitalize,**
- **convert_encoding,**
- **date,**
- **date_modify,**
- **default,**
- **escape,**
- **first,**
- **format,**
- **join,**
- **json_encode,**
- **keys,**
- **last,**
- **length,**
- **lower,**

Filtry wbudowane w Twig

- **merge,**
- **nl2br,**
- **number_format,**
- **raw,**
- **replace,**
- **reverse,**
- **round,**
- **slice,**
- **sort,**
- **split,**
- **striptags,**
- **title,**
- **trim,**
- **upper,**
- **url_encode.**

Szczegóły znajdziecie tutaj: http://twig.sensiolabs.org/doc/filters/index.html

Funkcje w Twigu

Twig ma też zestaw funkcji pełniących funkcję pomocniczą.
Funkcji – w odróżnieniu od filtrów – nie nakłada się na zmienne w celu modyfikacji ich wartości.

- **attribute,**
- **block,**
- **constant,**
- **cycle,**
- **date,**
- **dump, include,**
- **max,**
- **min,**
- **parent,**
- **random,**
- **range,**
- **source,**
- **template_from_string**

Szczegóły znajdziecie tutaj: http://twig.sensiolabs.org/doc/functions/index.html

Instrukcje warunkowe

W Twigu można korzystać z instrukcji warunkowych **`if/elseif/else`**.

Służą do tego odpowiednie tagi:

```
{% if v > 1 %}
    {# zmienna v większa od 1 #}
{% elseif v == 1 %}
    {# zmienna v równa 1 #}
{% else %}
    {# w pozostałych przypadkach, tj. zmienna v mniejsza od 1 #}
{% endif %}
```

Operatory logiczne

- b-and,
- b-xor,
- b-or,
- or,
- and,
- ==,
- !=,
- <,
- >,
- >=,
- <=,
- in,
- matches,
- starts with,
- ends with,
- is,
- not

```
{% if v is null or (v is not null and v <= 0) %}
    {# ... #}
{% elseif v not in [1, 2, 3] %}
    {# ... #}
{% elseif v matches '/wyrazenie_regularne/' %}
    {# ... #}
{% elseif v starts with 'a' %}
    {# ... #}
{% endif %}
```

Operatory matematyczne

- +
- -
- *
- /
- %
- ** - Operator potęgowania
- // - Operator dzielenia zwracający liczbę całkowitą (podłogę)

Pętla for

Pętla **`for`** jest jedyną implementowaną w **Twigu**.

Jej działanie jest zbliżone do pętli **`foreach`** w PHP. Da się dzięki niej przeiterować jakąś kolekcję (np. tablicę)

```
{% for val in arr %}
    {# ... #}
{% endfor %}
```

Albo jeżeli interesują nas też klucze:

```
{% for key, val in arr %}
    {{ key }} : {{ val }}
{% endfor %}
```

Pętla for

Jednym z usprawnień pętli w **Twigu** jest warunek **else**.

```
{% for val in arr %}
    {# ... #}
{% else %}
    {# ... #}
{% endfor %}
```

Pętla for

Jednym z usprawnień pętli w **Twigu** jest warunek **else**.

```
{% for val in arr %}
    {# ... #}
{% else %}
    {# ... #}
{% endfor %}
```

Jeżeli nie mamy po czym iterować, np. tablica jest pusta.

Zadania

Załączanie innych szablonów

Include

Najprostszym sposobem użycia innego szablonu jest metoda **include** (możemy przekazać zmienne).

```
{% for post in posts %}
    {{ include('post/show.html.twig', { 'post': post})}}
{% endfor %}
```

W pliku: **app/Resources/views/post/show.html.twig**:

```
<h2>{{ post.title }}</h2>
<h3 class="byline">by {{ post.authorName }}</h3>
<p>
    {{ post.body }}
</p>
```

Include

Najprostszym sposobem użycia innego szablonu jest metoda **include** (możemy przekazać zmienne).

```
{% for post in posts %}
    {{ include('post/show.html.twig', { 'post': post})}}
{% endfor %}
```

Taki zapis spowoduje dołączenie dla każdego elementu petli szablonu z innego pliku i wypełnienie go odpowiednimi danymi

W pliku: **app/Resources/views/post/show.html.twig**:

```
<h2>{{ post.title }}</h2>
<h3 class="byline">by {{ post.authorName }}</h3>
<p>
    {{ post.body }}
</p>
```

Załączanie kontrolerów

W bardziej skomplikowanym przypadku możemy w widoku wywołać akcje kontrolera. W tym celu używamy standardowej notacji:

- **BundleName:ControllerName:Action**

```
<div id="sidebar">
    {{ render(controller(
        'AppBundle:Article:recentArticles', {
        'max': 3 }
       ))
    }}
</div>
```

Załączanie kontrolerów

W bardziej skomplikowanym przypadku możemy w widoku wywołać akcje kontrolera. W tym celu używamy standardowej notacji:

- **BundleName:ControllerName:Action**

```
<div id="sidebar">
    {{ render(controller(
          'AppBundle:Article:recentArticles', {
          'max': 3 }
        ))
    }}
</div>
```

Nazwa akcji

Załączanie kontrolerów

W bardziej skomplikowanym przypadku możemy w widoku wywołać akcje kontrolera. W tym celu używamy standardowej notacji:

- **BundleName:ControllerName:Action**

```
<div id="sidebar">
    {{ render(controller(
        'AppBundle:Article:recentArticles', {
        'max': 3 }
       ))
    }}
</div>
```

Nazwa akcji

Możemy przekazać do akcji zmienne

Dziedziczenie szablonów

To najpotężniejsze narzędzie w **Twigu**. Nie jest to dziedziczenie znane z PHP!

Dziedziczenie pozwala tworzyć szablony zawierające bloki, które mogą zostać nadpisane przez szablony potomne.

Wykorzystujemy w tym celu takie tagi jak:

- **`extends`,**
- **`block`.**

Dziedziczenie szablonów

Szablon bazowy

```
<!DOCTYPE html>
<html>
    <head>
        {% block head %}
            <link rel="stylesheet" href="style.css" />
            <title>{% block title %}{% endblock %} | My Webpage</title>
        {% endblock %}
    </head>
    <body>
        <div id="content">{% block content %}{% endblock %}</div>
        <div id="footer">
            {% block footer %}
            © Copyright 2011 by <a href="http://...">Something</a>.
            {% endblock %}
        </div>
    </body>
</html>
```

Dziedziczenie szablonów

W szablonie znajdują się tagi **`block`**, a każdemu z nich nadajemy nazwę.

Dzięki temu w szablonie, który będzie dziedziczył po naszym szablonie bazowym, będziemy mogli wypełniać tylko te bloki, na których nam zależy.

Jeżeli nie nadpiszemy jakiegoś bloku, to zostanie tam zawartość z szablonu bazowego.

Dziedziczenie szablonów

Szablon dziedziczący z szablonu bazowego może wyglądać następująco.

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css" href="style_2.css"></style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

Dziedziczenie szablonów

Szablon dziedziczący z szablonu bazowego może wyglądać następująco.

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css" href="style_2.css"></style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

Informujemy z jakiego szablonu dziedziczymy

Dziedziczenie szablonów

Szablon dziedziczący z szablonu bazowego może wyglądać następująco.

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css" href="style_2.css"></style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

Nadpisujemy blok **`title`** odpowiednią wartością

Dziedziczenie szablonów

Szablon dziedziczący z szablonu bazowego może wyglądać następująco.

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css" href="style_2.css"></style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

W bloku **head** pozostawiamy wartość z szabonu bazowego (**parent()**) oraz dodajemy link

Dziedziczenie szablonów

Szablon dziedziczący z szablonu bazowego może wyglądać następująco.

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css" href="style_2.css"></style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

Nadpisujemy blok **content**

Dziedziczenie szablonów

Jeżeli nie chcemy całkowicie nadpisać całego bloku, a tylko dopisać do niego jakąś treść, używamy metody **`parent():`**

```
{% block head %}
    {{ parent() }}
    <style type="text/css" href="style_2.css"></style>
{% endblock %}
```

Dziedziczenie szablonów

Jeżeli nie chcemy całkowicie nadpisać całego bloku, a tylko dopisać do niego jakąś treść, używamy metody **`parent()`:**

```
{% block head %}
    {{ parent() }}
    <style type="text/css" href="style_2.css"></style>
{% endblock %}
```

Zawartość bloku z szablonu bazowego możemy dodać zarówno przed dodawanym kodem jak i za nim

Dziedziczenie

Możemy podać listę szablonów bazowych.
Pierwszy istniejący zostanie użyty.

```
{% extends ['layout.html', 'base_layout.html'] %}
```

Szablon bazowy może też być warunkowy:

```
{% extends standalone ? "minimum.html" : "base.html" %}
```

Zaawansowane wielokrotne dziedziczenie:
http://twig.sensiolabs.org/doc/tags/use.html

Dziedziczenie

Możemy podać listę szablonów bazowych.
Pierwszy istniejący zostanie użyty.

```
{% extends ['layout.html', 'base_layout.html'] %}
```

Szablon bazowy może też być warunkowy:

```
{% extends standalone ? "minimum.html" : "base.html" %}
```

Tutaj załadowany szablon zależy od wartości zmiennej **`standalone`**

Zaawansowane wielokrotne dziedziczenie:
http://twig.sensiolabs.org/doc/tags/use.html

Zadania

Linki i assety

Linki do innych stron

Żeby utworzyć linki do innych podstron naszego serwisu, możemy użyć dwóch metod wbudowanych w **Twig**:

- **`path()`**,
- **`url()`**.

Metoda **`path()`** przyjmuje nazwę akcji, do której ma kierować, i opcjonalnie tablicę, w której musimy podać wszystkie potrzebne do wygenerowania tego adresu slugi.

```
{% for article in articles %}
    <a href="{{ path('article_show', {'id': article.id}) }}">
        {{ article.title }}
    </a>
{% endfor %}
```

Metoda ta zwraca ścieżkę relatywną (bez domeny).

Przypomnienie – nazwy akcji

Nazwę akcji mogliśmy nadać przy pomocy adnotacji **`@Route()`**.

Jeżeli tego nie zrobiliśmy, to akcja ma nadaną domyślną nazwę:

- **`(nazwaKontrolera_nazwaAkcji)`**

Wszystkie nazwy akcji można zobaczyć dzięki komendzie konsolowej:

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

Linki do zasobów

- Z czasem w naszej aplikacji zasoby będą rosły (pliki JS, CSS, obrazy).
- Do tworzenia dynamicznych linków należy używać metody **asset()**.

```
<img src="{{ asset("bundles/coderslab/images/default_user.jpg") }}">
```

Gdzie trzymać zasoby?

Zasoby możemy trzymać w dwóch miejscach:

- w naszym bundlu (**src\myBundleName\Resources\public)**,
- bezpośrednio w katalogu web (uznawane teraz za najlepszą praktykę).

Bundle vs katalog web

Bundle

Jeżeli zdecydujemy się na trzymanie zasobów w naszym bundlu, to musimy pamiętać, że za każdym razem, kiedy je zmieniamy, musimy wywołać komendę konsolową:

```
php app/console assets:install
```

Komenda ta kopiuje nasze zasoby do katalogu: **/web/bundles/myBundleName.**

Katalog web

- Trzymanie zasobów w katalogu web jest uznane za najlepszą praktykę.
- Pozwala to zminimalizować problemy związane z rozproszeniem naszych zasobów na różne katalogi.
- Pamiętajmy jednak o tym, żeby stworzyć katalogi **/css, /js, /images.**

Zadania

Tłumaczenia

Ustawienia regionalne

Wszelkie teksty jakie umieszczamy w szablonach twiga możemy umiędzynarodowić z wykorzystaniem ustawień regionalnych (ang. locale).

Do uruchomienia tłumaczeń w symfony potrzebujemy kilku kroków:

1. Skonfigurowania usługi **`Translation`**,
2. Zmiany sposobu prezentacji łańcuchów tekstowych
3. Utworzenia pliku z tłumaczeniem dla danego języka

Konfiguracja usługi Translation

Domyślnie usługa **Translation** jest wyłączona.

Aktywacja usługi wymaga usunięcia komentarza w pliku **`config.yml`** w linii zawierającej następujący wpis:

```
#translator: { fallbacks: ["%locale%"] }
```

W pliku tym możemy również zmienić domyślny język używany przez naszą aplikację.

```
parameters:
  locale: en
```

Konfiguracja usługi Translation

Domyślnie usługa **Translation** jest wyłączona.

Aktywacja usługi wymaga usunięcia komentarza w pliku **`config.yml`** w linii zawierającej następujący wpis:

```
#translator: { fallbacks: ["%locale%"] }
```

- usuwamy

W pliku tym możemy również zmienić domyślny język używany przez naszą aplikację.

```
parameters:
  locale: en
```

Konfiguracja usługi Translation

Domyślnie usługa **Translation** jest wyłączona.

Aktywacja usługi wymaga usunięcia komentarza w pliku **`config.yml`** w linii zawierającej następujący wpis:

```
#translator: { fallbacks: ["%locale%"] }
```

- usuwamy

W pliku tym możemy również zmienić domyślny język używany przez naszą aplikację.

```
parameters:
  locale: en
```

en - możemy zmienić

Zmiana sposobu prezentacji tekstu

W widoku możemy użyć funkcji **trans** w następujący sposób tuż po tłumaczonym tekście:

```
{{ 'Hello world' | trans }}
```

Wyrażenie **Hello world** zostanie zamienione na to, odpowiadające aktualnym ustawieniom regionalnym.

Plik z tłumaczeniem

Ostatnią częścią procesu jest utworzenie pliku, który zawierał będzie wyrażenia do podmiany.

Plik ten należy utworzyć w katalogu:

<bundleName>/Resources/translations/

Prawidłowa nazwa pliku dla języka polskiego powinna wyglądać następująco:

messages.pl.xlf

Więcej o sposobie nazywania plików z tłumaczeniami można przeczytać w dokumentacji pod adresem: http://symfony.com/doc/2.8/components/translation.html#using-message-domains

Plik z tłumaczeniem

Zawartość pliku xlf z tłumaczeniami.

```
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext" original="file.ext
    <body>
      <trans-unit id="1">
        <source>Hello world</source>
        <target>Witaj świecie</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Na zakończenie

Po dodaniu nowego pliku z tłumaczeniami należy pamiętać o wyczyszczeniu cache Symfony odpowiednim poleceniem konsolowym.

```
php app/console cache:clear
```

Usługa translacji ma wiele możliwości oraz opcji do zastawania w praktyce:

- prawidłowa odmiana liczby mnogiej
- tłumaczenia ciągów tekstowych zawierających zmienne
- tłumaczenia z użyciem baz danych
- zmiana domyślnego języka aplikacji

Informacje te znajdziemy w dokumentacji: http://symfony.com/doc/2.8/translation.html

Zadania