Programowanie Aplikacji Internetowych

Laboratorium nr 2

Wzorzec MVC w PHP

Poniżej znajdują się zadania, które należy wykonać w ramach laboratoriów, a następnie sporządzić sprawozdanie w formie archiwum .zip. Plik archiwum powinien mieć nazwę zgodną ze wzorem: PAI_Lab<nr_laboratorium>_<pierwsza_litera_imienia>.<nazwisko_bez_polskich_znaków>.zip, np. PAI_Lab2_J.Kowalski.zip. W archiwum powinny znajdować się wszystkie pliki z poniższych zadań.

Zadanie 1 Podstawy MVC

Tworzymy następując strukturę plików (pustych) i katalogów:

```
\projekt
| - \app
|       | - \controllers
|       | - \core
|       |       | - App.php
|       |       \ - Controller.php
|       | - \models
|       | - \views
|       | - init.php
|       | - .htaccess
| - \public
|       | - .htaccess
|       \ - index.php
```

Zacznijmy od pliku */projekt/public/index.php*. Otwieramy ten plik i wpisujemy:

```
<?php
error_reporting(E_ALL);             // komunikaty diagnostyczne – bez względu
ini_set('display_errors', 1);       // na ustawienia serwera
require_once '../app/init.php';

$app = new App();
```

Powyższy kod importuje zawartość pliku */projekt/app/init.php* i tworzy instancję klasy *App*.

Czas na wpisanie kodu do pliku */projekt/app/init.php:*

```
<?php

require_once 'core/App.php';
require_once 'core/Controller.php';
```

W pliku tym importujemy zawartość pliku */projekt/app/core/App.php* oraz */projekt/app/core/Controller.php*.

Katalog */projekt/app/* przechowuje wrażliwe dla naszego systemu pliki dlatego warto ograniczyć do niego dostęp z poziomu przeglądarki poprzez dodanie odpowiednich instrukcji w pliku */projekt/app/.htaccess*:

```
Options -Indexes
```

Natomiast plik */projekt/public/.htaccess* odgrywa tutaj kluczową rolę z punktu widzenia działania systemu. Umieszczane w nim instrukcje włączają moduł *Rewrite* dzięki czemu będziemy mieli *przyjazne linki*:

```
Options -MultiViews
RewriteEngine On

RewriteBase /projekt/public
```

```
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f

RewriteRule ^(.+)$ index.php?url=$1 [QSA,L]
```

Pierwsza linijka jest opcjonalna, a wręcz może generować problemy jeżeli serwer jest niepoprawnie skonfigurowany. Dlatego w razie problemów należy usunąć ją w pierwszej kolejności. Wyłacza ona możliwość tworzenia wielu wersji strony. Kolejna linijka włącza moduł Rewrite, a kolejne najważniejsze dla niego parametry. RewriteBase określa ścieżkę do katalogu gdzie znajduje się aktualnie edytowany plik .htaccess. Jest to istotne ponieważ będzie on pod tym adresem zawsze szukany i stanowi katalog bazowy. Kolejne dwie linijki określają warunki istnienia pliku i katalogu, który domyślnie byłby wskazywany w pasku adresu. Jeżeli takowy nie będzie istniał wykonana zostanie reguła zdefiniowana w ostatniej linijce. Adres url jest przekazywany do pliku index.php jako wartość zmiennej.

W pliku */projekt/app/core/App.php* umieszczamy następujący kod:

```
<?php

class App
{
	protected $controller = 'home';	/* http://www.cos.pl/home */

	protected $method = 'index';	/* http://www.cos.pl/home/index */

	protected $params = [];	/* http://www.cos.pl/home/index/5/ */

	public function __construct()
	{
		$url = $this->parseUrl();
		var_dump($url);
	}

	protected function parseUrl()
	{
		if (isset($_GET['url']))
			return	explode('/',	filter_var(rtrim($_GET['url'],	'/'),
FILTER_SANITIZE_URL));
	}
}
```

W pliku tym znajduje się główny kod implementujący obsługę przyjaznych adresów. Funkcja *parseUrl()* spełnia tutaj krytyczne zadania interpretacji ciągu znaków wpisanych w pasek adresu przeglądarki. Funkcja *rtrim()* z parametrem '/' usuwa ten znak z końca ciągu adresu. Funkcja filter_var(), z parametrem FILTER_SANITIZE_URL, daje nam pewność, że z adresu zostaną usunięte wszystkie znaki nie zgodne z przyjętą konwencją ścieżek URL. Funkcja *explode()*, z parametrem '/', rozbija ciąg znaków na tablicę ciągów znaków po każdym wystąpieniu znaku '/'.

Tworzymy pusty plik */projekt/app/core/Controller.php*:

```
<?php
```

W tym momencie można uruchomić aplikację wpisują w przeglądarkę adres *http://localhost/projekt/public/product/create*.

Wyświetli się informacja o zmiennej $url utworzonej w konstruktorze klasy App.

Rozbudujmy dalej nasz kod aplikacji (*/projekt/app/core/App.php*):

```
<?php

class App
{
	protected $controller = 'home';	/* http://www.cos.pl/home */
```

```
	protected $method = 'index';  /* http://www.cos.pl/home/index */

	protected $params = [];       /* http://www.cos.pl/home/index/5/ */

	public function __construct()
	{
		$url = $this->parseUrl();

		if (file_exists('../app/controllers/' . $url[0] . '.php'))
		{
			$this->controller = $url[0];
			unset($url[0]);
		}
		require_once '../app/controllers/' . $this->controller . '.php';
		$this->controller = new $this->controller;

		if(isset($url[1]))
		{
			if(method_exists($this->controller, $url[1]))
			{
				$this->method = $url[1];
				unset($url[1]);
			}
		}

		$this->params = $url ? array_values($url) : [];
		call_user_func_array([$this->controller,     $this->method],     $this-
>params);

	}

	protected function parseUrl()
	{
		if (isset($_GET['url']))
			return    explode('/',    filter_var(rtrim($_GET['url'],    '/'),
FILTER_SANITIZE_URL));
	}
}
```

Dodany został kod rozpoznający kontroler, metodę i jej parametry. Stwórzmy w pliku */projekt/app/core/Controller.php* pustą klasę kontrolera:

```
<?php

class Controller
{

}
```

Teraz czas na pierwszy kontroler (domyślny) *projekt/app/controllers/home.php*:

```
<?php

class Home extends Controller
{
	public function index()
	{
		echo 'home/index';
	}

	public function test($p1 = '', $p2 = '')
	{
		echo $p1 . ' : ' . $p2;
	}
}
```

Przetestujmy go wpisując w adres przeglądarki *http://localhost/projekt/public/home/* oraz *http://localhost/projekt/public/home/test/1/3.*

Rozbudowując aplikację pozostaje nam zaimplementować obsługę modeli i widoków. W tym celu dodajmy kod do pliku */projekt/app/core/Controller.php*:

```
<?php

class Controller
{
     protected function model($model)
     {
        require_once '../app/models/' . $model . '.php';
        return new $model();
     }

      protected function view($view, $data = [])
     {

     }
}
```

Kod modelu umieszczamy w przykładowej klasie */projekt/app/models/product.php*:

```
<?php

class Product extends Model
{

}
```

Natomiast kod obsługi bazy danych i inne helpery w */projekt/app/core/Model.php*

```
<?php

class Model
{


}
```

W pliku */projekt/app/init.php* należy dopisać*:*

```
<?php

require_once 'core/App.php';
require_once 'core/Controller.php';
require_once 'core/Model.php';
```

Instancję danego modelu tworzymy w następujący sposób (*projekt/app/controllers/home.php*):

```
<?php

class Home extends Controller
{
      public function index()
      {
            echo 'home/index';
      }

      public function create()
      {
            $product = $this->model('product');
      }

      public function test($p1 = '', $p2 = '')
      {
            echo $p1 . ' : ' . $p2;
      }
}
```

Zadanie 2 Symfony 2

Na wstępie należy zainstalować PHP w najnowszej wersji oraz serwer bazy danych MySQL. Po instalacji sprawdzamy, czy wszystko działa poprawnie wpisując w konsoli:

```
php -version
```

Następnie instalujemy symfonię na **Linuxie**:

```
php -r "readfile('http://symfony.com/installer');" > symfony.phar
sudo mv symfony.phar /usr/local/bin/symfony
chmod a+x /usr/local/bin/symfony
symfony
```

Na **Windowsie**:

```
php -r "readfile('http://symfony.com/installer');" > symfony.phar
php symfony.phar
```

Tworzymy nowy projekt na Linuxie:

```
symfony new myproject
```

Na **Windowsie**:

```
php symfony.phar new myproject
```

Uruchamiamy aplikację:

```
cd myproject/
php app/console server:run
```

W przeglądarce wpisujemy adres http://localhost:8000/.

Czas stworzyć pierwszy moduł. W tym celu należy utworzyć nowy pakiet:

```
php app/console generate:bundle --namespace=Product/DemoBundle --format=annotation
```

Na każde pytanie odpowiadamy twierdząco lub wybieramy domyślą odpowiedź (jeżeli pytanie wymaga innej odpowiedzi niż yes/no). Kolejnym krokiem będzie konfiguracja bazy danych. W tym celu edytujemy plik my*project/app/config/parameters.yml*:

```
# app/config/parameters.yml
parameters:
    database_driver:    pdo_mysql
    database_host:      localhost
    database_name:      test_project
    database_user:      root
    database_password:  password

# ...
```

Odpowiednio zmieniamy wybrane parametry w taki sposób aby były zgodne z ustawieniami naszego serwera bazy danych.

Czas zająć się modelem aplikacji. Poniższym poleceniem utworzymy bazę danych, której nazwę przed chwilę określiliśmy, na naszym serwerze baz danych:

```
php app/console doctrine:database:create
```

Czas utworzyć naszą klasę (ang. Entity class) odwzorowującą strukturę naszego modelu w pliku *src/Product/DemoBundle/Entity/Product.php*:

```
<?php
// src/Product/DemoBundle/Entity/Product.php
namespace Product\DemoBundle\Entity;
```

```
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $name;

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $price;

    /**
     * @ORM\Column(type="text")
     */
    protected $description;
}
```

Poniższym poleceniem wygenerujemy metody dostępu (setery i getery):

```
php app/console doctrine:generate:entities Product/DemoBundle/Entity/Product
```

Kolejnym poleceniem na podstawie wygenerowanego kodu utworzymy odpowiednią strukturę w bazie danych:

```
php app/console doctrine:schema:update --force
```

Teraz, gdy mamy już utworzony model zarówno w aplikacji, jak i w bazie danych, możemy przejść do napisania pierwszej strony, która będzie wyświetlała listę produktów. W tym celu dodajemy kod do metody indexAction() w klasie wysuniętego kontrolera (ang. front controller) w pliku *src/Product/DemoBundle/Controller/DefaultController.php*:

```
<?php

namespace Product\DemoBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
      /**
       * @Route("/productdemo", name="product_demo_homepage")
       */
      public function indexAction()
      {
            $products = $this->getDoctrine()
                  ->getRepository('ProductDemoBundle:Product')
                  ->findAllOrderedByName();

            return $this->
```

```
                    render('ProductDemoBundle:Default:index.html.twig',
                          array( 'products' => $products ) );
        }
}
```

W powyższym kodzie odwołujemy się do repozytorium, którego jeszcze nie mamy. Utworzy go teraz. W pierwszym kroku musimy dodać adnotację w pliku *src/Product/DemoBundle/Entity/Product.php*:

```
<?php
// src/Product/DemoBundle/Entity/Product.php

namespace Product\DemoBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="Product\DemoBundle\Entity\ProductRepository")
 * @ORM\Table(name="product")
 */
class Product
{
```

Teraz możemy wygenerować repozytorium:

```
php app/console doctrine:generate:entities Product
```

oraz dopisać interesującą nas metodę w pliku *src/Product/DemoBundle/Entity/ProductRepository.php*:

```
<?php
// src/Product/DemoBundle/Entity/ProductRepository.php

namespace Product\DemoBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery(
                'SELECT p FROM ProductDemoBundle:Product p ORDER BY p.name ASC'
            )
            ->getResult();
    }
}
```

Ostatecznie już pozostaje nam utworzenie szablonu, który posłuży nam do wygenerowania widoku. W tym celu wstawiamy następujący kod do pliku *src/Product/DemoBundle/Resources/views/Default/index.html.twig*:

```
{% extends '::base.html.twig' %}

{% block body %}
      <table border=1>
            <tr><th>Name</th><th>Price</th><th>Description</th></tr>
            {% for p in products %}
                  <tr>
                        <td>{{ p.name }}</td>
                        <td>{{ p.price }}</td>
                        <td>{{ p.description }}</td>
                  </tr>
            {% endfor %}
      <table>
{% endblock %}
```

W tym momencie można przetestować pierwszy widok. W przeglądarce wpisujemy adres

http://localhost:8000/productdemo/. Tabela jest pusta ponieważ w bazie danych nie ma żadnych produktów. Dlatego teraz dodamy funkcjonalność dodawania nowego produktu. W tym celu należy dodać odpowiednie hiperłącze w pliku *src/Product/DemoBundle/Resources/views/Default/index.html.twig*:

```
{% extends '::base.html.twig' %}

{% block body %}
      <table border=1>
            <tr><th>Name</th><th>Price</th><th>Description</th></tr>
            {% for p in products %}
                  <tr>
                        <td>{{ p.name }}</td>
                        <td>{{ p.price }}</td>
                        <td>{{ p.description }}</td>
                  </tr>
            {% endfor %}
      <table>
      <a href="{{ path('product_demo_add') }}">Add product</a>
{% endblock %}
```

Dopiszmy teraz odpowiednią funkcję akcji w pliku *src/Product/DemoBundle/Controller/DefaultController.php*:

```
...
use Product\DemoBundle\Entity\Product
...
class DefaultController extends Controller
{
      ...
      /**
       * @Route("/productdemo/add", name="product_demo_add")
       */
      public function addAction()
      {
            $product = new Product();
            $product->setName('A Foo Bar');
            $product->setPrice('19.99');
            $product->setDescription('Lorem ipsum dolor');

            $form = $this->createFormBuilder($product)
                  ->add('name', 'text')
                  ->add('price', 'money')
                  ->add('description', 'textarea')
                  ->add('save', 'submit', array('label' => 'Create product'))
                  ->getForm();

            return $this->
              render('ProductDemoBundle:Default:product_form.html.twig',     array(
                  'form' => $form->createView(),
            ) );
      }
}
```

Pozostaje już tylko napisać szablon w pliku *src/Product/DemoBundle/Resources/views/Default/product_form.html.twig*:

```
% extends '::base.html.twig' %}

{% block body %}
      {{ form_start(form) }}
      {{ form_widget(form) }}
      {{ form_end(form) }}
      <a href="{{ path('product_demo_homepage') }}">Cancel</a>
{% endblock %}
```

Uruchommy ponownie aplikację - w przeglądarce wpisujemy adres http://localhost:8000/productdemo/. Dalej kliknij łącze 'Add product'. Na stronie z formularzem po wciśnięciu przycisku 'Add product' nic się nie dzieje. Należy dopisać obsługę formularza w pliku *src/Product/DemoBundle/Controller/DefaultController.php*:

```
...
use Symfony\Component\HttpFoundation\Request;
...
class DefaultController extends Controller
{
	...
	/**
	 * @Route(”/productdemo/add”, name=”product_demo_add”)
	 */
	public function addAction(Request $request)
	{
		...
		$form->handleRequest($request);

		if ($form->isValid()) {
			$em = $this->getDoctrine()->getManager();
			$em->persist($product);
			$em->flush();

			return $this->redirectToRoute('my_demo_homepage');
		}
		...
	}
}
```

Ponownie przetestujmy aplikację – http://localhost:8000/productdemo/.

Dodajmy opcję edycji. W tym celu dodajmy odpowiednie hiperłącza w pliku *src/Product/DemoBundle/Resources/views/Default/index.html.twig*:

```
{% extends '::base.html.twig' %}

{% block body %}
	<table border=1>
		<tr><th>Name</th><th>Price</th><th>Description</th></tr>
		{% for p in products %}
			<tr>
				<td>{{ p.name }}</td>
				<td>{{ p.price }}</td>
				<td>{{ p.description }}</td>
				<td><a href=”{{ path('product_demo_edit', {'id' : p.id} ) }}”>Edit</a></td>
			</tr>
		{% endfor %}
	<table>
	<a href=”{{ path('product_demo_create') }}”>Add product</a>
{% endblock %}
```

Dopiszmy odpowiednią funkcję akcji w pliku *src/Product/DemoBundle/Controller/DefaultController.php*:

```
...
class DefaultController extends Controller
{
	...
	/**
	 * @Route(”/productdemo/edit/{id}”, name=”product_demo_edit”)
	 */
	public function editAction(Request $request, $id)
	{
		$em = $this->getDoctrine()->getManager();
```

```
            $product = $em->getRepository('ProductDemoBundle:Product')->find($id);

            if (!$product) {
                  throw $this->createNoFoundException(
                        'No product found for id '.$id
                  );
            }

            $form = $this->createFormBuilder($product)
                  ->add('name', 'text')
                  ->add('price', 'money')
                  ->add('description', 'textarea')
                  ->add('save', 'submit', array('label' => 'Change product'))
                  ->getForm();

            $form->handleRequest($request);

            if ($form->isValid()) {
                  $em = $this->getDoctrine()->getManager();
                  $em->persist($product);
                  $em->flush();

                  return $this->redirectToRoute('my_demo_homepage');
            }

            return $this->
              render('ProductDemoBundle:Default:product_form.html.twig',     array(
                  'form' => $form->createView(),
            ) );
      }
}
```

Ponownie przetestujmy aplikację – http://localhost:8000/productdemo/.

Jedyne czego nam brakuje to możliwości usuwania produktów. Aby dodać tą funkcjonalność dodajmy odpowiednie hiperłącza w pliku *src/Product/DemoBundle/Resources/views/Default/index.html.twig*:

```
...
{% block body %}
      <table border=1>
            <tr><th>Name</th><th>Price</th><th>Description</th></tr>
            {% for p in products %}
                  <tr>
                        <td>{{ p.name }}</td>
                        <td>{{ p.price }}</td>
                        <td>{{ p.description }}</td>
                        <td><a    href="{{    path('product_demo_edit',    {'id'    :
p.id} ) }}">Edit</a></td>
                        <td><a href="{{ path('product_demo_delete', {'id' : p.id} )
}}">Del</a></td>
                  </tr>
            {% endfor %}
      <table>
      <a href="{{ path('product_demo_create') }}">Add product</a>
{% endblock %}
```

oraz dopiszmy odpowiednią funkcję akcji w pliku *src/Product/DemoBundle/Controller/DefaultController.php*:

```
...
class DefaultController extends Controller
{
      ...
      /**
       * @Route("/productdemo/del/{id}", name="product_demo_delete")
       */
```

```
    public function deleteAction($id)
    {
        $em = $this->getDoctrine()->getManager();
        $product = $em->getRepository('ProductDemoBundle:Product')->find($id);

        if ($product) {
            $em->remove($product);
            $em->flush();
        } else {
            throw $this->createNoFoundException(
                'No product found for id '.$id
            );
        }

        return $this->redirectToRoute('my_demo_homepage');
    }
}
```

W ten sposób ukończyliśmy pisanie modułu zarządzania produktem w aplikacji. Pozostaje tylko testowanie i usuwanie błędów - http://localhost:8000/productdemo/.

Zadanie 3 Implementacja modułu zakupów

W ramach samodzielnej pracy należy zaimplementować:

a) moduł rejestracji i logowania klienta,
b) funkcjonalność wyszukiwania produktów,
c) moduł koszyka,
d) historii zakupów,
e) dodać moduł kategorii,
f) itp.