3. Silnik MVC

3.1. Opis rozdziału

W tym rozdziale opisano jak stworzyć prosty silnik pozwalający na wygodne pisanie aplikacji na podstawie wzorca MVC. Silnik ten będzie automatycznie podłączał pliki aktualnie wymagane do poprawnego wyświetlenia strony oraz będzie operował na własnym formacie adresu URL do uzyskania którego będziemy musieli użyć pliku *.htaccess*. Ostatecznie przy pomocy tego silnika można napisać także własną stronę, blog, forum, portal czy cokolwiek się chce.

3.2. Organizacja plików

Zanim zaczniemy pisać skrypt silnika dobrze jest ustalić jakąś architekturę katalogów aby pracowało się wygodnie i wszystko było poukładane w logiczny sposób. Skoro ma to być silnik działający jak pseudo-framework MVC to trzeba uwzględnić podział katalogów aby w jednym znajdowały się pliki silnika w innym zaś modele w kolejnym widoki, a w następnym kontrolery. Do tego katalog z plikami konfiguracyjnymi, zasobami oraz bibliotekami czy skryptami JavaScript. Tak więc struktura katalogów będzie wyglądać mniej więcej tak:

- *_config* – pliki konfiguracyjne,
- *_controllers* – kontrolery,
- *_engine* – silnik,
- *_js* – skrypty JavaScript,
- *_libs* – dodatkowe biblioteki,
- *_models* – modele,
- *_resources* – zasoby,
- *_views* – widoki.

W katalogu z zasobami będziemy umieszczać kolejne tematyczne katalogi, które będą zawierać grafiki np.: *_units*, *_builds*, *_avatars*, *_images*.

3.3. Przekierowanie, plik *.htaccess*

Pliku *.htaccess* użyjemy do przekierowania na plik *index.php* w katalogu głównym z grą wszystkich zapytań do nieistniejących katalogów dzięki czemu uzyskamy możliwość wpisania w adresie URL własnego formatu adresu, który będzie używany przez silnik do obsługi architektury MVC ale o tym później, kod jaki należy umieścić w pliku *.htaccess* to:

Kod 3.3.1. Plik *.htaccess*

```
RewriteEngine on

RewriteBase /

RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.+)?$ index.php [QSA,L]
```

Oczywiście w przypadku kiedy gra jest w podkatalogu należy dopisać do `RewriteBase` ścieżkę do katalogu. Kiedy mamy już przekierowane wszystkie zapytania do nie istniejących katalogów na plik *index.php*, możemy przystąpić do pisania silnika aplikacji.

3.4. Plik inicjujący

W katalogu *_engine* stworzymy pliki silnika zaczynając od pliku inicjującego czyli *init.php*. W pliku umieszczamy kilka stałych globalnych opisujących nasz system takich jak autor czy wersja ponadto tworzymy zmienne zawierające adres hosta oraz ścieżkę do katalogu głównego. Następnie odczytujemy dane konfiguracyjne z pliku konfiguracyjnego i przy pomocy funkcji magicznej `__autoload` będziemy przyłączać kolejne pliki wymagane do wyświetlenia strony.

Kod 3.4.1. Plik init.php

```php
<?php

define('APP_PATH', $_SERVER['DOCUMENT_ROOT']);

define('APP_HOST', 'http://'.$_SERVER['HTTP_HOST']);

define('APP_AUTHOR', 'TKu');

define('APP_VERSION', '1.0');

define('APP_NAME', 'Podziemne Królestwo');

require_once(APP_PATH.'_config/configuration.php');

function __autoload($class_name) {
	require_once(APP_PATH.'_'.str_replace('_', '/', strtolower($class_name)).'.php');
}

session_start();

?>
```

Ważne aby stałe `APP_PATH` oraz `APP_HOST` wskazywały na adres katalogu głównego z grą. Zdarza się, że `$_SERVER['DOCUMENT_ROOT']` jest zakończona znakiem ukośnika, a zdarza się, że nie jest to zależne od serwera więc dobrze tutaj przy pomocy echa sprawdzić czy stałą ta zawiera ten znak na końcu, a w razie jego braku najzwyczajniej dodać go do stałej, dzięki temu potem unikniemy problemu ze źle ładowanymi plikami. Ważne jest aby zmodyfikować stałe `APP_PATH` oraz `APP_HOST` jeśli chcemy aby nasza gra znajdowała się w podkatalogu dodając do tych stałych brakujący kawałek adresu. Po dodaniu stałych ładowany jest plik konfiguracyjny zawierający stałe zawierające dane potrzeb np. do połączenia się z bazą danych.

Kod 3.4.2. Plik configuration.php

```php
<?php

defined('APP_PATH') or die();

define('DB_HOST', '');
define('DB_NAME', '');
define('DB_USER', '');
define('DB_PASS', '');

?>
```

Ciekawą funkcją jest funkcja magiczna `__autoload` która jest uruchamiana przed wywołaniem konstruktora klasy co jest dla nas bardzo miłym udogodnieniem bo dzięki temu będziemy mogli przed stworzeniem obiektu dodać plik zawierający jego kod. Plusem tej operacji jest fakt, że nie będziemy musieli w naszym kodzie przyłączać plików tylko będą ładowane automatycznie. W tej funkcji jest tylko jedna linijka, która pobiera z parametru nazwę wywołanej

klasy następnie przekształca ją na ścieżkę do pliku w którym znajduje się jej kod. Jak to się dzieje? Jest to bardzo proste gdyż nazwy naszych klas będą posiadały specyficzny format dzięki któremu będziemy mogli przekształcić nazwę klasy na ścieżkę do pliku w którym się znajduje. Na końcu uruchamiamy sesję.

3.5. Nazewnictwo klas, a nazwy plików

To jak nazwiemy nasze klasy ma znaczenie i jest ściśle powiązane z architektura naszych katalogów. Zasada jest bardzo prosta. Załóżmy że chcemy utworzyć kontroler *test*, a skoro wiemy, że kontrolery muszą się znajdować w katalogu *_controllers* oraz rzecz jasna plik musi posiadać rozszerzenie **.php* wtedy ścieżka do pliku wygląda następująco *_controllers/test.php*. Teraz na podstawie tego adresu możemy utworzyć nazwę klasy czyli znak podkreślenia usuwamy znak ukośnika zastępujemy podkreśleniem usuwamy rozszerzenie pliku, a pierwsze litery nazw katalogów i plików zamieniamy na duże po czym uzyskujemy nazwę klasy `Controllers_Test`. Nasz magiczna funkcja potrafi teraz w drugą stronę przekształcić tą nazwę na ścieżkę do pliku, tj. najpierw zamieniamy wszystkie duże litery na małe, podmieniamy podkreślenie na ukośnik i dodajemy podkreślenie przed nazwą folderu i rozszerzenie pliku. Należy się trzymać tej zasady bo inaczej silnik będzie wywalał błędy i nie będzie uruchamiał strony.

3.6. Format adresu URL, określenie kontrolera i akcji

Zanim przystąpimy do stworzenia parsera zastanówmy się jak chcemy określać który kontroler i akcje chcemy uruchomić. W tym przykładzie zrobimy w taki sposób, że w adresie URL podamy w postaci ścieżki do katalogu nazwę kontrolera oraz akcję jaką ma wywołać czyli: *adres_strony.pl/kontroler/akcja*

3.7. Parser adresu URL

Zakładając, że możemy teraz opracować własny format adresu URL i parser który go przygotuje do uruchamiania odpowiednich kontrolerów i akcji możemy także zrezygnować z klasycznego przesyłania danych metodą GET, a do odczytu tych danych podobnie jak danych przesłanych metodą POST użyjemy własnej funkcji. Tak więc chcąc przekazać w adresie więcej

informacji niż tylko nazwa kontrolera i akcji dopisujemy nasze parametry w postaci dalszej ścieżki czyli: *adres_strony.pl/kontroler/akcja/parametr1/parametr2* wartości parametrów jak i dane przekazane metodą POST zostaną dodane do tablicy z której będziemy je odczytywać. Co za tym idzie zrezygnujemy z odczytu danych w postaci `$_GET[]` czy `$_POST[]`.

Zaczynamy od stworzenia pustej klasy z polami oraz nagłówkiem konstruktora klasy, a następnie dodamy pozostałe funkcje.

Kod 3.7.1. Klasa parsera adresu URL

```php
<?php

defined('APP_PATH') or die();

class Engine_UrlParser {

        private $_controller;
        private $_action;
        private $_arg = array();
        private $_args = 0;

        public function __construct() {
        }

}

?>
```

Pola klasy będą wypełnione na podstawie danych pobranych z adresu URL, a oznaczają:

- `$_controller` – nazwa kontrolera,
- `$_action` – nazwa akcji,
- `$_arg` – tablica parametrów przekazanych w adresie oraz metodą POST,
- `$_args` – licznik parametrów pobranych z adresu, bez zliczania danych przesłanych metodą POST.

Konstruktor jak widać to metoda magiczna i w niej odczytujemy adres URL i odczytujemy interesujące nas dane.

Kod 3.7.2. Konstruktor klasy parsera

```
$temp = explode('/', substr(strtolower($_SERVER['REQUEST_URI']), 1));
$this->_controller = $temp[0];
if(!empty($temp[1])) $this->_action = $temp[1];

if(sizeof($temp) > 2) {
        for($i = 2; $i < sizeof($temp); $i++) {
                if($temp[$i]){
                        $this->_arg['arg'.$this->_args++] = $temp[$i];
                }
        }
}

if($_POST) while(list($key, $value) = each($_POST)) $this->_arg[$key] = $value;

$_REQUEST    = array();
$_POST       = array();
$_GET        = array();
```

Na początku przetwarzamy nasz adres URL aby można było z niego wyciągnąć interesujące nas informacje. Informacje jakie nas interesują odczytamy z adresu URI czyli części adresu, który znajduje się po nazwie domeny naszej strony czyli *http://www.adres_strony.pl*, ta cześć adresu zaczyna się od ukośnika i zawierać będzie pseudo katalogowy format adresu, który stworzyliśmy czyli */kontroler/akcja/parametr* jeśli zostaną określone. Odcinamy od adresu URI jeden znak którym w tym przypadku jest ukośnik, bez odcięcia tego znaku będziemy generować złą tablicę czyli w pierwszym polu tablicy nie będziemy mieli żadnej wartości, oczywiście w przypadku gdy gra jest w podkatalogu musimy odciąć tyle znaków ile jest w ścieżce do katalogu z grą z odcięciem ukośnika po ścieżce. Następnie rozbijamy adres gdzie separatorem jest ukośnik, w pieszym polu tablicy z rozbitym adresem będziemy mieli nazwę kontrolera w następnym nazwę akcji, a reszta to dodatkowe parametry. Kontroler i akcję przypisujemy do odpowiednich zmiennych. Jeśli w adresie podano więcej danych są one dodawane do tablicy `$_arg` gdzie indeksem jest zlepek `arg` oraz numer parametru liczony od zera. Dla przykładu dla adresu:

adres_strony.pl/kontroler/akcja/parametr0/parametr1

parser podzieli dane tak że:

Kod 3.7.3. Zawartość pól klasy parsera po wywołaniu przykładowego adresu

```
$_controller = 'kontroler';
$_action = 'akcja';
$_arg['arg0'] = 'parametr0';
$_arg['arg1'] = 'parametr1';
```

Po odczytaniu wszystkich parametrów z adresu odczytujemy dane przesłane metodą POST i zapisujemy je do tablicy `$_arg`, a indeksy pól ustawiamy takie same jak indeksy w tablicy `$_POST`. Przykładowo jeśli parser odczyta z tablicy POST indeks `name` o wartości `Tomasz` to do tablicy `$_arg` zostanie dodane pole `name` o wartości `Tomasz`:

Kod 3.7.4. Zawartość pól klasy parsera po odebraniu wartości z tablicy POST

```
$_arg['name'] = 'Tomasz';
```

Na koniec czyścimy tablice `$_REQUEST`, `$_POST`, `$_GET` bo nie będą przez nas używane i nie chcemy aby ktoś z nich korzystał.

Następnie dodajemy kilka funkcji dzięki którym będziemy mogli odczytać pobrane z adresu wartości z pól klasy parsera:

Kod 3.7.5. Funkcje zmieniające/odczytujące zawartości pól klasy parsera

```
public function getController() { //odczytanie nazwy kontrolera
        return $this->_controller;
}

public function setController($value) { //nadanie nazwy kontrolera
        $this->_controller = $value;
}

public function getAction() { //odczytanie nazwy akcji
        return $this->_action;
}

public function setAction($value) { //nadanie nazwy akcji
        $this->_action = $value;
}

public function getArg($id) { //odczytanie wartości parametru
        if(gettype($id) == 'integer') $arg = 'arg'.$id;
        else $arg = $id;
        if( isset($this->_arg[$arg]) ) return $this->_arg[$arg];
        else return false;
}

public function setArg($id, $value) { //zmiana wartości parametru
        if(gettype($id) == 'integer') $arg = 'arg'.$id;
        else $arg = $id;
        $this->_arg[$arg] = $value;
}

public function getSizeOfArgs() { //odczytanie ilości parametrów przekazanych w adresie
        return $this->_args;
}
```

3.8. FrontController

FrontController jest uruchamiany jako pierwsza klasa, która ma za zadanie sprawdzić czy istnieje kontroler, który chcemy wywołać i tworzy instancje klasy wywoływanego kontrolera, a w razie jego braku uruchamia domyślny kontroler.

Kod 3.8.1. FrontController

```
<?php

defined('APP_PATH') or die();

class Engine_FrontController {

        private $ControllerPathPrefix;
        private $ControllerPathSufix;

        public function __construct() {
                $this->ControllerPathPrefix = APP_PATH.'_controllers/';
                $this->ControllerPathSufix = '.php';
        }

        public function ControllerLoad(Engine_UrlParser $parser) {
                if($parser->getController() != '') {
                        if(!file_exists($this->ControllerPathPrefix.$parser->getController().$this-
>ControllerPathSufix)) $parser->setController('home');
                } else $parser->setController('home');

                $ControllerName = 'Controllers_'.ucfirst($parser->getController());

                $Controller = new $ControllerName($parser);
        }

}

?>
```

W tym silniku domyślny kontroler nosi nazwę `home` i musi być zawsze utworzony nawet jeśli ma być pusty w środku. Pusty nie znaczy czysty plik ale plik zawierający podstawową strukturę.

3.9. Plik główny strony

Skoro już mamy pliki silnika to stwórzmy plik *index.php* który będzie uruchamiał te pliki. Plik tworzymy w katalogu głównym strony i umieszczamy w nim kod:

Kod 3.9. ***index.php***

```
<?php

ob_start();

require_once('_engine/init.php');

$Controller = new Engine_FrontController();
$UrlParser = new Engine_UrlParser();
$Controller->ControllerLoad($UrlParser);

?>
```

Na razie nasz silnik będzie wyrzucał błędy ponieważ musimy dodać domyślny kontroler.

3.10. Domyślny kontroler

Stwórzmy klasę która będzie dziedziczona przez nasze klasy z kontrolerami aby nie tworzyć w każdej klasie jednego i tego samego konstruktora który ma za zadanie wywołać funkcję określonej akcji. Tak więc zacznijmy od utworzenia klasy `Controllers_Engine`, która to będzie zawierała konstruktor i domyślną akcję, którą będziemy mogli przeciążyć aby stworzyć inną zawartość tej akcji.

Kod 3.10.1 Rodzic kontrolerów

```
<?php

defined('APP_PATH') or die();

class Controllers_Engine {

	public function __construct(Engine_UrlParser $parser) {
		if($parser->getAction() != '') {
			if(!method_exists($this,$parser->getAction().'Action')) $parser-
>setAction('index');
		} else $parser->setAction('index');
		$ActionName = $parser->getAction().'Action';
		$this->$ActionName($parser);
		$this->parser = $parser;
	}

	//funkcja domyślnej akcji

}

?>
```

Podobnie jak w przypadku FrontController'a konstruktor sprawdza czy istnieje funkcja akcji jaką chcemy uruchomić, a w przeciwnym przypadku uruchamiana jest ustawiana domyślna nazwa akcji tutaj jest to akcja index. Następnie tworzona jest nazwa akcji, która jest zlepkiem dwóch słów z których pierwsze określa nazwę akcji, a drugie to słowo `Action`, w wyniku zlepienia otrzymujemy

nazwę funkcji, która zawiera kod akcji. Wielkość liter ma tutaj znaczenie i nazwa akcji powinna się składać wyłącznie z małych liter, a słowo `Action` powinno się zaczynać od wielkiej litery. Pozostałe funkcje mogą być nazywane jakkolwiek i nie będą one uruchamiane jako akcje przez konstruktor.

Kod 3.10.2. Domyślna akcja kontrolera

```
public function indexAction(Engine_UrlParser $parser) {
        echo 'Dealult constructor index.';
}
```

Domyślna akcja ma za zadanie wypisać na ekranie przykładowy tekst aby sprawdzić czy silnik działa poprawnie i czy załadował konstruktor, ale zanim będziemy mogli przetestować jego działanie musimy stworzyć domyślny konstruktor o podstawowej strukturze.

Domyślny konstruktor to ten który będzie uruchamiany jeśli chcemy uruchomić konstruktor, który nie istnieje. Każdy konstruktor musi dziedziczyć po klasie `Controllers_Engine`, bez czego nie zostanie załadowana odpowiednia akcja.

Kod 3.10.3. Domyślny pusty konstruktor

```
<?php

defined('APP_PATH') or die();

class Controllers_Home extends Controllers_Engine {

        //akcje i funkcje

}

?>
```

Teraz możemy uruchomić silnik i powinniśmy zobaczyć informację z domyślnej akcji, ale chcemy aby domyślna akcja tego kontrolera wyświetlała to co chcemy np. stronę główna tak więc musimy przeciążyć funkcję `indexAction` dodając ją do klasy naszego kontrolera.

Kod 3.10.4. Własna domyślna akcja kontrolera

```
public function indexAction(Engine_UrlParser $parser) {
        echo '
                <h1>Krasnale</h1>
                Własna strona testowa.
        ';
}
```

Po dodaniu domyślnej akcji do naszego konstruktora zobaczymy nasz komunikat.

3.11. Widok

Widoki czyli zestaw funkcji które będą służyły nam do stworzenia wizualnej prezentacji strony w przeglądarce. Mechanizm tutaj przedstawiony będzie bardzo prosty i nie powinno być problemu z jego zrozumieniem. Zaczniemy od przygotowania klasy rodzica dla widoków która będzie zawierać funkcje, które będą powtarzalne dla każdego widoku czyli między innymi pobieranie treści do umieszczenia w odpowiednich miejscach na stronie, funkcje generujące tagi do znacznika `<head>`, funkcja generująca komunikaty o błędzie, sukcesie czy tylko zwykłą informacja, funkcja która będzie rysować blok w sekcji głównej strony oraz na końcu co najważniejsze funkcja rysująca wszystko na stronie.

Zaczniemy od utworzenia tego pliku z pustą klasą oraz komentarzami w miejsca których będą wstawiane poszczególne funkcje. W katalogu *_views* tworzymy plik *engine.php* w którym umieszczamy kod naszej klasy:

Kod 3.11.1. Pusta klasa rodzica widoków

```php
<?php

defined('APP_PATH') or die();

class Views_Engine {

	//pola klasy

	//konstruktor klasy

	//funkcje zarządzania treścią główna strony

	//funkcje zarządzania treścią znacznika <head>

	//funkcje zarządzania treścią menu górnego

	//funkcje zarządzania treścią panelu bocznego

	//funkcje zarządzania treścią stopki

	//funkcja rysująca widok

	//funkcje generujące tagi dla znacznika <head>

	//funkcje z komunikatami

}

?>
```

Teraz możemy dodawać w odpowiednia miejsca z komentarzami poszczególne kawałki

kodu. Na początek 2 pola do których będziemy wstawiać to co ma zostać wstawione pomiędzy znaczniki <head> oraz to co będzie widoczne w głównej części strony.

Kod 3.11.2. Pola klasy engine

```
private $_container;
private $_head;
private $_top_menu;
private $_aside;
private $_footer;
```

Następnie tworzymy konstruktor który w tym przypadku doda 3 meta tagi do sekcji <head> takie jak tytuł strony, kodowanie strony oraz domyślny arkusz stylów, a także domyślną stopkę strony.

Kod 3.11.3. Konstruktor klasy engine

```
public function __construct($http = 'text/html; charset=UTF-8') {
        $this->setTitleTag('Krasnale');
        $this->setMetaTagHttp('Content-Type', $http);
        $this->setMetaTagLink('stylesheet', '/_views/style.css', 'text/css', '');
        $this->setFooter('&copy <a href="home">Podziemne królestwo</a>');
}
```

Kolejne funkcje pozwalają do dodanie oraz odczytanie wartości z pola z główną treścią strony czyli tą do której będziemy wstawiać wszystkie treści podstron.

Kod 3.11.4. Funkcje zarządzania treścią główna strony

```
public function getContainer() {
        if( isset($this->_container) ) return $this->_container;
        else return false;
}

public function setContainer($value) {
        $this->_container .= $value;
}
```

Oraz funkcje dodającą oraz odczytującą zawartość znacznika <head>.

Kod 3.11.5. Funkcje zarządzania treścią znacznika <head>

```
public function getHead() {
        if( isset($this->_head) ) return $this->_head;
        else return false;
}

public function setHead($value) {
        $this->_head .= $value;
}
```

Funkcja dodająca oraz odczytująca menu górne które trzeba uprzednio wygenerować, lub

dla sprawdzenia można dodać kilka przykładowych odsyłaczy.

Kod 3.11.6 Funkcje zarządzania treścią menu górnego

```
public function getHead() {
        if( isset($this->_head) ) return $this->_head;
        else return false;
}

public function setHead($value) {
        $this->_head .= $value;
}
```

Funkcja dodająca oraz odczytująca treści znajdujące się wewnątrz panelu bocznego.

Kod 3.11.7. Funkcje zarządzania treścią panelu bocznego

```
public function getHead() {
        if( isset($this->_head) ) return $this->_head;
        else return false;
}

public function setHead($value) {
        $this->_head .= $value;
}
```

Oraz funkcja dodająca oraz odczytująca treści ze stopki strony.

Kod 3.11.8. Funkcje zarządzania treścią stopki

```
public function getHead() {
        if( isset($this->_head) ) return $this->_head;
        else return false;
}

public function setHead($value) {
        $this->_head .= $value;
}
```

Kolejna funkcja będzie odpowiadać za narysowanie naszego widoku, poprzez odczytanie wartości z pól funkcji oraz wstawienia ich do pliku z szablonem html naszej strony.

Kod 3.11.9. Funkcja rysująca widok

```
public function view() {
        $head = $this->getHead();
        $container = $this->getContainer();
        $top_menu = $this->getTopMenu();
        $aside = $this->getAside();
        $footer = $this->getFooter();
        include ('_views/template.phtml');
}
```

Następnie zestaw kilku funkcji generujących różne tagi i meta tagi dla znacznika `<head>`.

Kod 3.11.10. Funkcje generujące tagi dla znacznika <head>

```
public function setTitleTag($title) {
	$this->setHead('<title>'.$title.'</title>');
}

public function setMetaTagHttp($name, $content) {
	$this->setHead('<meta http-equiv="'.$name.'" content="'.$content.'"/>');
}

public function setMetaTagName($name, $content) {
	$this->setHead('<meta name="'.$name.'" content="'.$content.'" />');
}

public function setMetaTagProperty($name, $content) {
	$this->setHead('<meta property="'.$name.'" content="'.$content.'" />');
}

public function setMetaTagLink($rel, $href, $type, $title = '') {
	$this->setHead('<link rel="'.$rel.'" href="'.APP_HOST.$href.'" type="'.$type.'" title="'.
$title.'" />');
}

public function setMetaTagScript($src, $title = '') {
	$this->setHead('<script type="text/javascript" src="'.APP_HOST.'/_js/'.$src.'" title="'.
$title.'"></script>');
}

public function siteRefresh($time = 1, $url = '') {
	$this->setContainer($this->setMetaTagHttp('Refresh', $time.'; url='.APP_HOST.$url));
}
```

A na końcu 3 funkcje które będą wyświetlać w odpowiednich kolorach treści komunikatów o błędzie, sukcesie oraz zwykła informacja.

Kod 3.11.11. Funkcje z komunikatami

```
public function error($com) {
	$this->setContainer('<div id="com" class="error">'.$com.'</div>');
}

public function success($com) {
	$this->setContainer('<div id="com" class="success">'.$com.'</div>');
}

public function info($com) {
	$this->setContainer('<div id="com" class="info">'.$com.'</div>');
}
```

Aby wszystko działało jak powinno należy stworzyć plik z szablonem html do którego będzie wstawiona wygenerowana treść. Plik ten został opisany w poprzednim rozdziale i z niego skorzystamy, z małymi modyfikacjami czyli określeniem miejsc w które mają zostać wstawione wygenerowane treści. Tworzymy więc plik *template.phtml* w katalogu *_view* z kodem:

Kod 3.11.12. Szablon HTML

```
<?defined('APP_PATH') or die();?>
<!DOCTYPE html>
	<head>
		<?=$head?>
	</head>
	<body>
		<div id="top">
			<header>
			</header>
			<div id="menu_gorne">
				<?=$top_menu?>
			</div>
			<div id="strona">
				<aside>
					<?=$aside?>
				</aside>
				<div id="tresc">
					<?=$container?>
				</div>
				<div style="clear: both;"></div>
			</div>
			<footer>
				<?=$footer?>
			</footer>
		</div>
	</body>
</html>
```

W miejsca ze zmiennymi zostaną wstawione wygenerowane przez nas wartości, ale ważne jest aby wywołać funkcję rysująca dopiero po zakończeniu generowania treści gdyż treści wygenerowane po wywołaniu tej funkcji nie znajdą się na stronie. Do pełni szczęścia zostaje nam jeszcze dodanie arkusza stylów i tutaj także jak z plikiem z szablonem strony użyjemy arkusza stylów powstałego w poprzednim rozdziale. Kopiujemy więc plik z poprzedniego rozdziału i umieszczamy w katalogu *_view* pod nazwa *style.css*. Musimy jeszcze dodać do arkusza stylów opis dla naszych komunikatów tak więc doklejamy do pliku następującą treść:

Kod 3.11.13. Formatowanie komunikatów

```
#com {
	font-weight: bold;
}

.error {
	color: #FF0000;
}

.success {
	color: #0000FF;
}

.info {
	color: #00FF00;
}
```

Następnie należy skopiować grafiki z poprzedniego rozdziału i wkleić do katalogu *resources/images*, a w pliku CSS zmienić ścieżki do używanych grafik na *../resources/images/plik.png*.

Zanim przetestujemy nasze widoki musimy dodać widok dziedziczący po klasie rodzica z widokami w którym będziemy opisywać widoki dla danego kontrolera. Tak więc dodajmy przykładowy widok domyślny, który będzie współgrał z kontrolerem domyślnym. W katalogu *_view* utwórzmy plik *home.php* z kodem przykładowego widoku.

Kod 3.11.14. Klasa domyślnego widoku

```
<?php

defined('APP_PATH') or die();

class Views_Home extends Views_Engine {

	function testTopMenu() {

		$this -> setTopMenu('
			<ul>
				<li><a href="home">Strona główna</a></li>
				<li><a href="home/reg">Rejestracja</a></li>
				<li><a href="home/owu">OWU</a></li>
				<li><a href="home/kontakt">Kontakt</a></li>
			</ul>
		');

	}

}

?>
```

Klasa musi dziedziczyć po klasie `Views_Engine` aby nie trzeba było definiować funkcji rysujących wypełniających znacznik `<head>` bo w innym wypadku nie miało by sensu tworzenie widoków skoro w każdym pliku miałby być napisany za każdym razem oddzielny szablon HTML. W klasie dodałem przykładową funkcję, która doda menu górne.

Teraz przetestujmy jak uruchomienie widoku, w domyślnym kontrolerze. Wpisując do funkcji z domyślną akcją index taki oto przykładowy kod:

Kod 3.11.15. Testowanie widoku

```
$tpl = new Views_Home();

$tpl -> testTopMenu();

$tpl -> setContainer('Przykładowa treść.');

$tpl -> success('Strona działa :)');

$tpl -> view();
```

Po uruchomieniu strony w przeglądarce powinniśmy już zobaczyć stronę w postaci podobnej do utrzymanej w poprzednim rozdziale z dodanymi w odpowiednie miejsca przykładowymi treściami.

3.12. Model czyli połączenie z bazą danych

Teraz zajmiemy się warstwą modelu naszej aplikacji, która będzie odpowiadać za przetwarzanie danych czyli w tym przypadku głównie operacje na bazie danych, które będą zwracane do konstruktora, a następnie umieszczane w widoku.

Należało by zacząć jak w poprzednich podrozdziałach od stworzenia klasy rodzica, która w tym przypadku będzie odpowiadała za nawiązania połączenia z bazą danych. W klasie tej będzie znajdował się tylko konstruktor. Dodajmy plik *engine.php* do katalogu *_models*, a w nim kod:

Kod 3.12.1. Klasa rodzica dla modelów

```
<?php

defined('APP_PATH') or die();

class Models_Engine {

	public $pdo;

	public function __construct() {
		try {
			$this->pdo = new PDO('mysql:host='.DB_HOST.';dbname='.DB_NAME, DB_USER,
DB_PASS);
		} catch(PDOException $e) {
			echo 'DB_CONNECT_ERROR: '. $e->getMessage();
			return false;
		}
	}

}

?>
```

Klasa ta otwiera połączenie z bazą danych korzystając z biblioteki PDO, a dane wymagane do połączenia z bazą danych znajdują się w pliku *configuration.php* w katalogu *_config*, które trzeba we własnych silnikach ustawić pod własną bazę danych. Następnie trzeba utworzyć potomka tej klasy aby można było zdefiniować dla danego konstruktora wymagane funkcje. Stwórzmy konstruktor dla domyślnego kontrolera pod nazwą Views_Home z kodem:

Kod 3.12.2. Szablon klasy modelu

```
<?php

defined('APP_PATH') or die();

class Models_Home extends Models_Engine {

        public function  temp() {
                $pdo = $this->pdo;
        }

}

?>
```

Na tym etapie dodajmy pustą klasę z szablonem funkcji, bo nie mamy jeszcze stworzonej bazy danych. Stwórzmy więc bazę danych z przykładową tabelą:

Kod 3.12.3. Tworzenie bazy danych oraz przykładowej tabeli

```
CREATE DATABASE `krasnale` DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_polish_ci;
USE krasnale;

CREATE TABLE temp (
  id int(11) NOT NULL AUTO_INCREMENT,
  title varchar(255) NOT NULL,
  text text NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_polish_ci AUTO_INCREMENT=1 ;
```

Następnie dodajmy do naszej przykładowej tabeli kilka rekordów na których będziemy testować działanie naszej klasy z modelem.

Kod 3.12.4. Dodawanie rekordów do tabeli temp

```
INSERT INTO temp( title, TEXT ) VALUES
('Pierwszy tytył',  'Pierwszy tekst'),
('Drugi tytył',  'Drugi tekst'),
('Trzeci tytył',  'Trzeci tekst'),
('Czwarty tytył',  'Czwarty tekst');
```

Na tak przygotowanych przykładowych danych możemy przetestować nasz model poprzez dodanie funkcji, która odczyta dane z bazy danych następnie prześle je do kontrolera, który

uruchomi widok i wypisze odczytane dane w prostym bloku.

Zacznijmy od utworzenia funkcji odczytującej dane z bazy danych, dodajmy do klasy modelu domyślnego funkcję `selectPosts` z kodem:

Kod 3.12.5. Funkcja `selectPosts` odczytująca rekordy z bazy danych

```
public function  selectPosts() {
        $pdo = $this->pdo;
        $temp = array();

        $select = $pdo -> query("SELECT * FROM temp");
        if($select -> rowCount()) {
                $lp = 0;
                while($row = $select -> fetch()) $temp[$lp++] = $row;
                return $temp;
        } else return 0;
        $select -> closeCursor();
}
```

Jest to prosta funkcja odczytująca z tabeli wszystkie rekordy, a razie gdy zapytanie zwróci zerową ilość odczytanych rekordów funkcja zwraca zero co oznacza niepowodzenie, w przeciwnym wypadku zwracana jest tablica z odczytanymi wartościami. Następnie dodajmy do domyślnego widoku funkcje generujące kod postów.

Kod 3.12.6. Funkcje generujące widok z postami

```
function genPosts($posts) {
        if($posts) {
                for($i = 0; $i < count($posts); $i++){
                        $this -> showPost($posts[$i]);
                }
        }
}

function showPost($post) {
        $this -> setContainer('
                <article>
                        <h1>'.$post['title'].'</h1>
                        <p>'.$post['text'].'</p>
                </article>
        ');
}
```

Teraz możemy przetestować jak wszystko ze sobą współgra, dodajmy do domyślnej akcji domyślnego konstruktora kod:

Kod 3.12.7. Test współdziałania modelu, widoku oraz konstruktora

```
public function indexAction(Engine_UrlParser $parser) {

        $tpl = new Views_Home();
        $mdl = new Models_Home();

        $tpl -> testTopMenu();

        $posts = $mdl -> selectPosts();
        if($posts) $tpl -> genPosts($posts); else $tpl -> error('Próba odczytania danych nie
powiodła się!');

        $tpl -> view();

}
```

Funkcja ta najpierw tworzy instancje klas widoku oraz modelu, a następnie przypisujemy do zmiennej wynik działania funkcji modelu odczytującej rekordy gdy zmienna będzie posiadać wartość zero to drukowany jest komunikat, a w przypadku kiedy zmienna będzie posiadała jakieś wartości zostaje uruchomiona funkcja widoku generująca kod HTML z postami. Na końcu rysujemy widok. Po uruchomieniu skryptu w przeglądarce powinniśmy ujrzeć, że zostały wyświetlone rekordy z bazy danych w oddzielnych blokach.

Ten silnik można wykorzystać nie tylko do pisania gier ale także zwykłych stron internetowych. W pliku „*roz3.rar*” znajduje się pełny kod źródłowy skryptów z tego rozdziału.

3.13. Ćwiczenia

1. Dodaj nowy kontroler, widok oraz model.
2. Dodaj do nowego widoku funkcję rysującą formularz, który ma dodawać nowe rekordu do przykładowej tabeli temp z tego rozdziału.
3. Dodaj do nowego modelu funkcję dodającą rekordy do tabeli temp poprzez wcześniej utworzony formularz.
4. Uzupełnij nowy konstruktor tak aby poprawnie odczytał dane przesłane z formularza przekazał je do modelu, który doda te dane do bazy danych oraz wyświetli odpowiedni komunikat o powodzeniu bądź też niepowodzeniu dokonania transakcji.