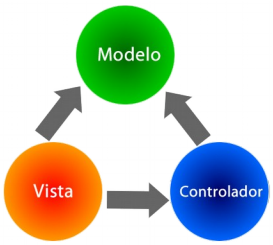
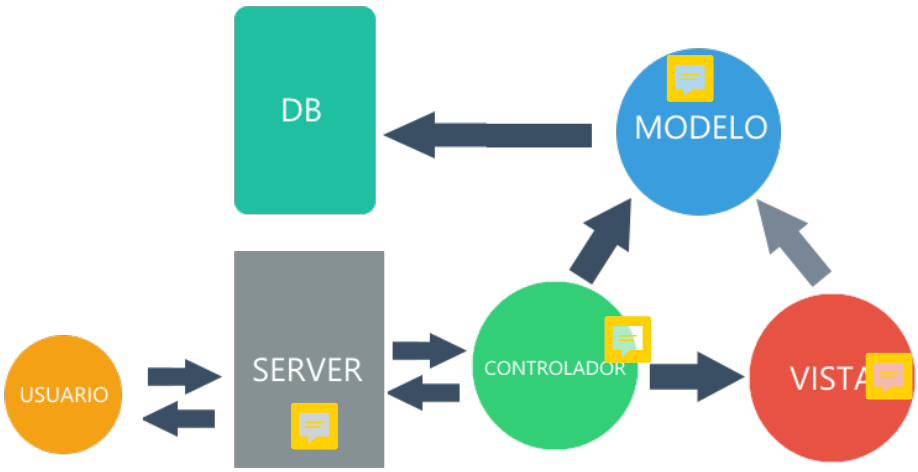
MVC



Ciclo: DAW
Módulo: DWES
Curso: 2020-2021
Autor: César Guijarro Rosaleny

Introducción

Modelo-vista-controlador (**MVC**) es un patrón de arquitectura de software, que separa los **datos** y la **lógica de negocio** de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello *MVC* propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.



Vamos a ver como aplicar este patrón a la programación web.

Preparando la aplicación

Hasta ahora hemos visto como hacer webs accediendo a los archivos a través de la url. Es decir, si el usuario escribe *www.miweb.com/peliculas.php* nosotros le enviamos el contenido de *peliculas.php* que tenemos almacenado en nuestro servidor.

La mayoría de webs utilizan urls amigables en lugar del esquema anterior. Por ejemplo, para acceder a la página anterior el usuario escribiría *www.miweb.com/peliculas*, y en lugar de acceder a una película de la forma *www.miweb.com/peliculas?id=5* lo haría escribiendo en el navegador *www.miweb.com/peliculas/5*. De esta forma es más fácil controlar las rutas y qué archivos servir a cada una.

La idea es definir una serie de rutas válidas y qué controlador y acción ejecutar para cada una de esas rutas. El controlador accederá al modelo adecuado para sacar los datos y montará la vista con esos datos. De esta forma separamos la obtención de los datos de su presentación, y podemos cambiar la forma de obtener los datos sin necesidad de modificar la vista. Incluso podríamos, si usásemos algún sistema de plantillas, cambiar el lenguaje de programación que utilizamos en el servidor y las vistas no se verían afectadas.

Crear la estructura de nuestra web

Lo primero que haremos será crear la estructura de carpetas y archivos que vamos a usar en nuestra web.

Vamos a crear la carpeta *mvc* en *apache_htdocs/DWS/ejemplos/T4*. Dentro de esa carpeta tendremos, por ahora, un sólo archivo llamado *index.php* con el siguiente contenido:

```
<?php

echo "Hola mundo";
```

Además, crearemos las siguientes carpetas:

- **bddd**, archivos donde estarán los datos en forma de arrays
- **controllers**, donde tendremos nuestros controladores
- **core**, donde estarán las clases base de nuestra aplicación
- **css**, donde estarán nuestros archivos css
- **imgs**, carpeta con nuestras imágenes
- **models**, donde estarán nuestros modelos
- **routes**, donde estará nuestro controlador de rutas
- **views**, carpeta con nuestras vistas

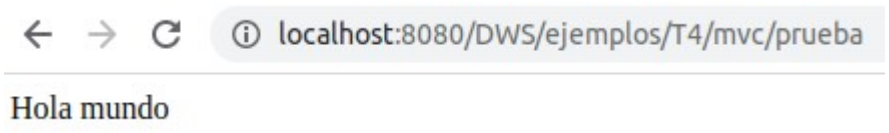
Redirigir las peticiones

Lo primero que haremos es redirigir todas las peticiones a nuestra web a nuestro *index.php*. El proceso cambiará dependiendo de que servidor HTTP se utilice.

En nuestro caso, al usar Apache, tenemos que utilizar un archivo especial llamado .**htaccess** (fíjate bien que el nombre empieza por punto). Crea el archivo .htaccess en la carpeta raiz con el siguiente contenido:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . ./index.php [L]
```

Si accedes ahora a *apache_htdocs/DWS/ejemplos/T4/mvc* deberías ver la frase "*Hola mundo*". Es más, si accedes a cualquier ruta, por ejemplo */mvc/prueba* (a partir de ahora me ahorraré la primera parte de la url para simplificar los apuntes) deberías ver también la frase.



Una vez hemos conseguido que todas las peticiones a nuestra web cargue *index.php* vamos con nuestro enrutador.

Enrutador

Como hemos dicho, tenemos que poder definir rutas válidas y servir una respuesta a cada una de ellas. Podríamos hacer nuestro propio enrutador leyendo la url y quedándonos con la última parte para ver que archivo cargar, pero vamos a utilizar Composer para facilitarnos la vida.

Inicializa Composer dejando las opciones por defecto y después instala la librería nikic/fast-route. Esta librería nos permite definir rutas y las respuestas adecuadas para cada una de ellas.

```
cesar@Asus:~/Proyectos/www/DWS/ejemplos/T4/mvc$ composer require nikic/fast-route
Cannot create cache directory /home/cesar/.composer/cache/repo/https---repo.packagis
t.org/, or directory is not writable. Proceeding without cache
Cannot create cache directory /home/cesar/.composer/cache/files/, or directory is no
t writable. Proceeding without cache
Using version ^1.3 for nikic/fast-route
./composer.json has been updated
Cannot create cache directory /home/cesar/.composer/cache/repo/https---repo.packagis
t.org/, or directory is not writable. Proceeding without cache
Cannot create cache directory /home/cesar/.composer/cache/files/, or directory is no
t writable. Proceeding without cache
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing nikic/fast-route (v1.3.0): Downloading (100%)
Writing lock file
Generating autoload files
cesar@Asus:~/Proyectos/www/DWS/ejemplos/T4/mvc$
```

En la documentación de la librería tienes un ejemplo de como usarla. Lo primero será crear el archivo *web.php* dentro de la carpeta *routes*. En este archivo estará el código para capturar las rutas y servir la respuesta adecuada.

No te preocupes si no entiendes todo el código del ejemplo, poco a poco veremos como utilizarla.

Añade el siguiente código a *web.php*:

```php
<?php
$dispatcher = FastRoute\simpleDispatcher(function(FastRoute\
RouteCollector $r) {
	$basedir = '/DWS/ejemplos/T4/mvc';
	$r->addRoute('GET', $basedir . '/', 'Hola mundo');
	$r->addRoute('GET', $basedir . '/universo', 'Hola universo');
});

// Fetch method and URI from somewhere
$httpMethod = $_SERVER['REQUEST_METHOD'];
$uri = $_SERVER['REQUEST_URI'];

// Strip query string (?foo=bar) and decode URI
if (false !== $pos = strpos($uri, '?')) {
	$uri = substr($uri, 0, $pos);
}

$uri = rawurldecode($uri);

$routeInfo = $dispatcher->dispatch($httpMethod, $uri);

switch ($routeInfo[0]) {
	case FastRoute\Dispatcher::NOT_FOUND:
		// ... 404 Not Found
		echo "404 – Ruta no válida";
		break;
	case FastRoute\Dispatcher::METHOD_NOT_ALLOWED:
		$allowedMethods = $routeInfo[1];
		// ... 405 Method Not Allowed
		break;
	case FastRoute\Dispatcher::FOUND:
		echo $routeInfo[1];
		break;
}
```

Vamos a explicar un poco como funciona la librería. En la primera función anónima es donde definimos nuestras rutas válidas usando la fórmula *$r→addRoute(método http, ruta, acción);*. Añadimos la variable *$basedir* para no tener que repetir la primera parte de la url en las rutas (dependiendo de donde hayas creado el proyecto tendrás que cambiar el valor).

Hemos definido dos rutas válidas usando el mismo método HTTP (por ahora siempre usaremos GET, ya veremos en temas posteriores como usar otros métodos): la ruta raíz (/) y universo (/universo). La acción de las rutas sera la frase que tenga que salir por pantalla.

Las siguientes líneas se utilizan para sacar el método y la URI del cliente. Por último se almacena en el array *$routeInfo* la siguiente información:

- *$routeInfo[0]*: Si se ha encontrado la ruta y si el método es válido o inválido.
- *$routeInfo[1]*: La acción a ejecutar.
- *$routeInfo[2]*: Variables de las rutas, lo veremos más adelante

Por ahora mostramos el mensaje “*404 – Ruta no válida*” si la ruta es inválida o el mensaje de la acción si la ruta es válida.

Ahora tenemos que cargar el autoload del Composer en nuestro *index.php* e incluir nuestro archivo de rutas:

```
<?php

require 'vendor/autoload.php';

include ('./routes/web.php');
```

De esta forma, da igual la ruta a la acceda el cliente, siempre cargará nuestro autoloader y nuestro archivo *routes/web.php*.

Si todo ha funcionado bien, deberías ver el mensaje "*Hola mundo*" cuando accedas a la ruta /*mvc* y el mensaje "*Hola universo*" cuando accedas a /*mvc/universo*. Si intentas acceder a otra ruta, por ejemplo /*mvc/prueba* debería mostrarte el mensaje "*404 – Ruta no válida*".

Controladores

Vamos a añadir ahora los controladores. Por ahora solo definiremos un controlador principal con tres acciones.

Crea el archivo *Main.php* en la carpeta *controllers* con el siguiente contenido:

```
<?php

class Main {

	function mundo() {
		echo "Hola mundo";
	}

	function universo() {
		echo "Hola universo";
	}

	function error() {
		echo "404 - Ruta no válida";
	}
}
```

Modifica el archivo *web.php* de la carpeta *route* para indicar el controlador y la acción a cargar en cada una de las rutas de la siguiente manera:

```
$r->addRoute('GET', $basedir . '/', 'main@mundo');
$r->addRoute('GET', $basedir . '/universo', 'main@universo');
```

Lo que hacemos es indicar en un string que controlador usar y que acción ejecutar separado por el símbolo arroba (@).

El siguiente paso es hacer que nuestro enrutador cargue la acción del controlador adecuado.

Como hemos dicho, en la variable *$routeInfo[1]* se almacena la acción a ejecutar, que en este caso tendrá el aspecto *controlador@acción*. Lo único que tenemos que hacer es obtener ese controlador y esa acción separando el string mediante el método **explode()** con el carácter @. Utilizamos el método **ucfirst()** para poner la primera letra del controlador en mayúscula.

Después incluimos el archivo, creamos un objeto de la clase y ejecutamos la función que toca.

```
case FastRoute\Dispatcher::FOUND:
        $handler = $routeInfo[1];
        $partes = explode('@', $handler);
        $controllerName = ucfirst($partes[0]);
        $action = $partes[1];
        include ('./controllers/' . $controllerName . '.php');
        $controller = new $controllerName();
        $controller->$action();
        break;
}
```

Para el caso en que no encuentre la ruta, podemos llamar directamente a la función *error()* del controlador:

```
case FastRoute\Dispatcher::NOT_FOUND:
        $controllerName = 'Main';
        $action = 'error';
        include ('./controllers/' . $controllerName . '.php');
        $controller = new $controllerName();
        $controller->$action();
        break;
```

Mejorando la inclusión de controladores

Vamos a utilizar los namespaces y el autoload del Composer para no tener que incluir los archivos de los controladores antes de instanciar la clase en el archivo *web.php*.

Para añadir la autocarga nuestras clases al auloader del Composer, tenemos que modificar el archivo *composer.json* añadiendo la sección *autoload*. Esta propiedad permite configurar la carga automática de clases. Los tres valores permitidos para esta propiedad son el cargador de clases que sigue el estándar PSR-4, el mapa de clases (classmap) y los archivos individuales (files). El método recomendado es el cargador PSR-4 porque es el más flexible (no hace falta por ejemplo regenerar el cargador de clases cuando se añade una nueva clase al proyecto).

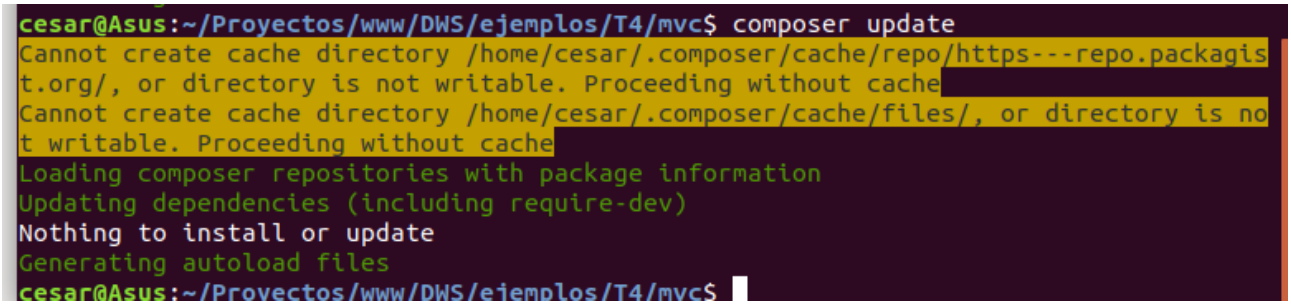
Edita el archivo y añade el siguiente código:

```
{
	"name": "cesar/mvc",
	"authors": [
		{
			"name": "César Guijarro",
			"email": "cguijarro@fpmislata.com"
		}
	],
	"require": {
		"nikic/fast-route": "^1.3"
	},
	"autoload": {
		"psr-4": {"controllers\\": "controllers"}
	}
}
```

Utilizamos la clave psr-4 para definir el mapeo entre namespaces y rutas del sistema de archivos (estas rutas son relativas respecto al directorio raíz del proyecto).

Ten en cuenta que los namespaces deben acabar siempre en \\ para evitar problemas con la carga de clases. En nuestro caso añadimos el namespace *controllers*\\ y le indicamos que los archivos está en la carpeta *controllers*.

Una vez guardado el archivo, tendremos que usar la orden **composer update** para que se reflejen los cambios.



Si abrimos el archivo *vendor/autload_psr4.php* podemos ver como Composer ha actualizado su autoload para indicarle en que carpeta están los archivos del namspace *controllers*.

```php
<?php

// autoload_psr4.php @generated by Composer

$vendorDir = dirname(dirname(__FILE__));
$baseDir = dirname($vendorDir);

return array(
    'controllers\\' => array($baseDir . '/controllers'),
    'FastRoute\\' => array($vendorDir . '/nikic/fast-route/src'),
);
```

Después añade el namespace *controllers* al archivo */controllers/Main.php*.

```
<?php

namespace controllers;

class Main {

	function mundo() {
		echo "Hola mundos";
	}
...
```

A partir de ahora ya podemos ahorrarnos el *include* del archivo de los controladores en el fichero *web.php* añadiendo el namespace a la variable *$controllerName* (tenemos que escapar el carácter \ del namespace con el carácter de escape de PHP, que en este caso es la misma barra invertida, de ahí que escribamos *controllers*\\ en el string):

```
case FastRoute\Dispatcher::FOUND:
	$handler = $routeInfo[1];
	$partes = explode('@', $handler);
	$controllerName = '\\controllers\\' . ucfirst($partes[0]);
	$action = $partes[1];
	$controller = new $controllerName();
	$controller->$action();
	break;
}
```

Vistas

El siguiente paso es incorporar las vistas a nuestra web. Para eso vamos a instalar un sistema de plantillas PHP mediante Composer: **league/plates:v4.0.0-alpha**.

Instala la librería como siempre (**asegurate de instalar la versión 4, si no algunas de las funciones posteriores no funcionarán**):

```
cesar@Asus:~/Proyectos/www/DWS/ejemplos/T4/mvc$ composer require league/plates:v
4.0.0-alpha
Cannot create cache directory /home/cesar/.composer/cache/repo/https---repo.pack
agist.org/, or directory is not writable. Proceeding without cache
Cannot create cache directory /home/cesar/.composer/cache/files/, or directory i
s not writable. Proceeding without cache
./composer.json has been updated
Cannot create cache directory /home/cesar/.composer/cache/repo/https---repo.pack
agist.org/, or directory is not writable. Proceeding without cache
Cannot create cache directory /home/cesar/.composer/cache/files/, or directory i
s not writable. Proceeding without cache
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 0 installs, 1 update, 0 removals
  - Updating league/plates (3.3.0 => v4.0.0-alpha): Downloading (100%)
Writing lock file
Generating autoload files
cesar@Asus:~/Proyectos/www/DWS/ejemplos/T4/mvc$
```

Plates es un sistema nativo de plantillas PHP, lo que quiere decir que no tenemos que aprender otra sintaxis y estructuras de control. Es rápido y fácil de usar, y soporta herencia de plantillas y envío de datos a las plantillas entre otras cosas.

Para empezar a usar el sistema de plantillas debemos instanciar la clase Engine. Ésto lo haremos en nuestro *index.php* para tener las plantillas accesibles en todo el código.

```php
<?php

require 'vendor/autoload.php';

$templates = League\Plates\Engine::create('./views');

include ('./routes/web.php');
```

Al instanciar la clase, usamos el método *create()*, al cual le pasamos la ruta donde estarán nuestras plantillas (en nuestro caso la carpeta *views*).

Vamos a crear ahora nuestra primera plantilla. Será la plantilla *padre* de la que heredarán el resto, por lo tanto contendrá el esqueleto de nuestras páginas con el menú y un *div* donde estará el contenido.

Usaremos Bootstrap para facilitar el maquetado de nuestra web.

Vamos a hacer una webs de libros. La página principal tendrá el menú y una imagen como portada. El menú tendrá sólo un enlace que será *libros*, donde sacaremos un listado con los libros que tenemos almacenados.

A su vez, cada libro será un enlace a su ficha individual, donde mostraremos su título, autor, editorial y precio.

Las plantillas de Plates tienen como extensión *phtml*, por lo que nuestros archivos serán del estilo *nombre_plantilla.phtml*.

Crea el archivo *main.phtml* en la carpeta *views* con este contenido:

```
<!DOCTYPE html>
<head>
	<meta charset="utf-8">
	<title><?=$v($title)?></title>
	<meta name="description" content="">
	<meta name="viewport" content="width=device-width, initial-scale=1">
	<!-- Bootstrap CSS -->
	<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.
css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/
iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
	<link rel="stylesheet" href="./css/estilos.css">
</head>
<body>
	<div class="alert alert-secondary d-flex">
		<a href="/DWS/ejemplos/T4/mvc/libros" class="btn btn-
dark">Libros</a>  
	</div>
	<div class="container">
		<?=$v->section('content')?>
	</div>
</body>
</html>
```

Aparte de los componentes típicos de un archivo html, tenemos un enlace a Bootstrap, otro a nuestra hoja de estilos almacenada en la carpeta *css* (que no tenemos todavía) y dos divs principales: nuestro menú *Libros* y el contenido en *div.containter*.

Fíjate además que tenemos dos bloques de código PHP. Uno será el título de nuestra página (*$v()$title*) y el otro el contenido (*$v→section('content')*).

Ahora vamos a crear nuestro *index.pthml* con este código:

```
<?php
        $v->layout('main', ['title' => 'Biblioteca'])
?>

<img src="./imgs/portada.png" class="mx-auto d-block"/>
```

En la primera línea (el código PHP) indicamos que plantilla vamos a usar como base, pasándole además el título como parámetro. El resto del código será el cuerpo de nuestra página (sustituirá al bloque *$v→section('content')* de la plantilla *main.phtml*).

Por último, guarda la imagen *portada.png* en la carpeta *imgs*.

Haz otra plantilla llamada *error.phtml* para mostrar la página de error, cambiando la imagen por un texto que sea algo parecido a "*404 – Página no encontrada*".

Ya estamos listos para mostrar nuestras plantillas en los controladores, pero antes vamos a hacer algo para facilitarnos la tarea. La mayoría de los controladores mostrarán una vista en algún momento, con lo que tendremos que prepararlos para usarlas individualmente.

Para mejorar el código vamos a crear una clase padre *Controller* donde engancharemos la clase usada por las plantillas, y haremos que nuestros controladores hereden de esa clase, con lo que siempre las tendremos accesibles.

Crea el archivo *Controller.php* en la carpeta *core* con el siguiente código:

```
<?php
namespace core;

use League\Plates\Engine as PlatesEngine;

class Controller {

	protected $templates;

	public function __construct(PlatesEngine $templates) {
		$this->templates = $templates;
	}
}
```

Como ves, importamos la clase *Engine* del sistema de plantillas y creamos una propiedad para almacenarla.

Ahora sólo tenemos que hacer que nuestro controlador *Main* herede de dicha clase, pero antes tenemos que añadir el namespace *core* (donde está nuestra clase padre *Controller*) al autoload del Composer como hemos hecho antes con el namespace *controllers*.

```
"autoload": {
	"psr-4": {
			"controllers\\": "controllers",
			"core\\": "core"
	}
}
```

```
cesar@Asus:~/Proyectos/www/DWS/ejemplos/T4/mvc$ composer update
Cannot create cache directory /home/cesar/.composer/cache/repo/https---repo.pack
agist.org/, or directory is not writable. Proceeding without cache
Cannot create cache directory /home/cesar/.composer/cache/files/, or directory i
s not writable. Proceeding without cache
Loading composer repositories with package information
Updating dependencies (including require-dev)
Nothing to install or update
Generating autoload files
cesar@Asus:~/Proyectos/www/DWS/ejemplos/T4/mvc$
```

Ya podemos modificar nuestro controlador *Main* para que herede de la clase principal *Controller* (acuérdate de importar el namespace y la clase con **use**). Además, sustituiremos las funciones anteriores (*hola()*, *universo()* y *error()*) por dos nuevas, *index()* y *error()*:

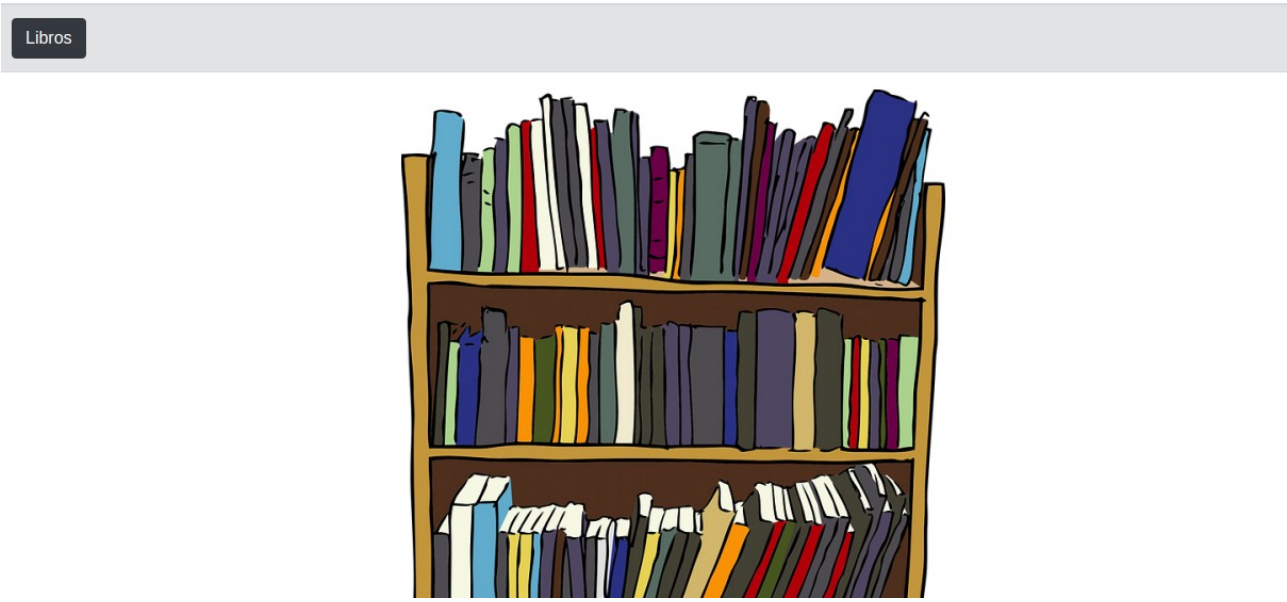
```php
<?php

namespace controllers;
use core\Controller;

class Main extends Controller{

	function index() {
		echo $this->templates->render('index');
	}

	function error() {
		echo $this->templates->render('error');
	}
}
```

Como ves, para mostrar una plantilla tenemos que escribir el código "*echo $this→templates→render(nombre_plantilla)".*

Por último, modificamos nuestro archivo de rutas para cambiar las acciones de las rutas y añadir la variable *$templates* creada en *index.php* al constructor del controlador:

```php
<?php
$dispatcher = FastRoute\simpleDispatcher(function(FastRoute\
RouteCollector $r) {
	$basedir = '/DWS/ejemplos/T4/mvc';
	$r->addRoute('GET', $basedir . '/', 'main@index');
});

// Fetch method and URI from somewhere
$httpMethod = $_SERVER['REQUEST_METHOD'];
$uri = $_SERVER['REQUEST_URI'];

// Strip query string (?foo=bar) and decode URI
if (false !== $pos = strpos($uri, '?')) {
	$uri = substr($uri, 0, $pos);
}

$uri = rawurldecode($uri);
$routeInfo = $dispatcher->dispatch($httpMethod, $uri);

switch ($routeInfo[0]) {
	case FastRoute\Dispatcher::NOT_FOUND:
		// ... 404 Not Found
		$controllerName = '\\controllers\\Main';
		$action = 'error';
		$controller = new $controllerName($templates);
		$controller->$action();
		break;
	case FastRoute\Dispatcher::METHOD_NOT_ALLOWED:
		$allowedMethods = $routeInfo[1];
		// ... 405 Method Not Allowed
		break;
	case FastRoute\Dispatcher::FOUND:
		$handler = $routeInfo[1];
		$partes = explode('@', $handler);
		$controllerName = '\\controllers\\' . ucfirst($partes[0]);
		$action = $partes[1];
		$controller = new $controllerName($templates);
		$controller->$action();
		break;
}
```

Si todo funciona como toca, al acceder a la ruta principal nos mostrará el menú con la imagen de la librería como portada:



Y la página de error si accedemos a cualquier otra página:

También podemos crear nuestro propio css para modificar a nuestro gusto la web. Sólo tenemos que guardarlo en la carpeta *css* con el nombre *estilos.css* (podemos modificar el nombre de la hoja de estilos si cambiamos el enlace en nuestra plantilla *main.phtml*).

Modelos

Como siempre, vamos a simular nuestra base de datos con un archivo PHP que contendrá un *array* con los datos de los libros. Crea el archivo *libros.php* en la carpeta *bbdd* y añade los siguientes datos:

```php
<?php

return array(
	array(
		'id' => 1,
		'titulo' => 'El nombre de la rosa',
		'autor' => 'Umberto Eco',
		'editorial' => 'Debolsillo',
		'precio' => 11.35
	),
	array(
		'id' => 2,
		'titulo' => 'Sapiens (de animales a dioses)',
		'autor' => 'Yuval Noah Harari',
		'editorial' => 'Debate',
		'precio' => 18.90
	),
	array(
		'id' => 3,
		'titulo' => 'American gods',
		'autor' => 'Neil Gaiman',
		'editorial' => 'Roca editorial de libros',
		'precio' => 18.95
	)
);
```

Teniendo en cuenta que la mayoría de modelos van a necesitar traer todos los datos de la tabla y buscar por *id*, vamos a crear una clase padre *Model.php* en la carpeta *core* con esos dos métodos para que todos nuestros modelos hereden de ella y no tener que repetir las funciones.

Además, la clase tendrá una propiedad *$table* para asociar el archivo PHP donde están los datos. También aprovecharemos para crear el método mágico *__callStatic()* y poder llamar a los métodos de forma estática.

```php
<?php
namespace core;

class Model {

	protected $table;

	protected function getAll() {
		$result = include('./bbdd/' . $this->table);
		return $result;
	}

	protected function getById($id) {
		$result = array();
		$rows = include('./bbdd/' . $this->table);
		$expression = "[?id == `" . $id . "`]";
		$result = JmesPath\search($expression, $rows)[0];
		return $result;
	}

	public static function __callStatic($name, $arguments) {
		return (new static)->$name(...$arguments);
	}
}
```

Ten en cuenta que tendrás que instalar la librería JmesPath como siempre con Composer para poder utilizarla.

De esta forma, ya podemos crear nuestros modelos heredando de la clase padre especificando la tabla a la que hará referencia.

Empezamos creando el modelo *Libro.php* en la carpeta *models*. Acuérdate de añadir el namespace *models* e importar *core\Model*.

```php
<?php

namespace models;
use core\Model;

class Libro extends Model{

	protected $table = 'libros.php';

}
```

Vamos a crear el controlador *Libro.php* que será el encargado de gestionar todas las peticiones que tengan que ver con la tabla *Libro*, pero antes vamos a añadir el namespace *models* al autoloader del Compose como hemos hecho antes.

```
"autoload": {
	"psr-4": {
		"controllers\\": "controllers",
		"core\\": "core",
		"models\\": "models"
	}
}
```

```
cesar@Asus:~/Proyectos/www/DWS/ejemplos/T4/mvc$ composer update
Cannot create cache directory /home/cesar/.composer/cache/repo/https---repo.pack
agist.org/, or directory is not writable. Proceeding without cache
Cannot create cache directory /home/cesar/.composer/cache/files/, or directory i
s not writable. Proceeding without cache
Loading composer repositories with package information
Updating dependencies (including require-dev)
Nothing to install or update
Generating autoload files
cesar@Asus:~/Proyectos/www/DWS/ejemplos/T4/mvc$
```

Ya podemos crear nuestro controlador *Libros.php*.

```
<?php

namespace controllers;
use core\Controller;
use models\Libro as modelLibro;

class Libro extends Controller{

	function getAll() {
		$libros = modelLibro::getAll();
		echo $this->templates->render('libros_listado', ['libros' =>
$libros]);
	}

}
```

Fíjate en el código porque hay un par de detalles importantes. Primero, tienes que crear un alias cuando importas la clase y el namespace *models\Libro* ya que nuestro controlador también es una clase llamada *Libro* y daría un error por duplicidad de nombres de clase.

El controlador sólo tendrá (por ahora) el método *getAll()*, el cual llamará el método *getAll()* del modelo *Libro* (aunque se llamen igual, son métodos diferentes) para traer los datos de todos los libros.

Antes hemos visto como mostrar una plantilla por pantalla sin pasarle datos. En esta ocasión la plantilla *libros_listado* (que crearemos a continuación) necesita los datos de los libros que tiene que mostrar. La forma de pasarle datos a una plantilla en Plates es en forma de *array* como segundo parámetro del método *render()*.

Vamos a crear ahora la plantilla *libros_listado.phtml*.

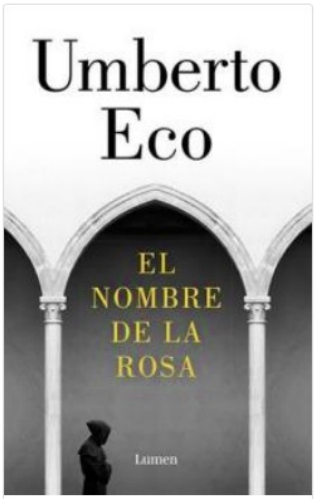
```
<?php
	$v->layout('main', ['title' => 'Libros'])
?>

<div class="row mx-auto">

<?php
	foreach ($libros as $libro) {
		echo '<div class="col-3">';
		echo '<a href="libros/' . $libro['id'] . '" class="custom-card">';
		echo '<div class="card" style="width: 100%;">';
		echo '<img class="card-img-top" src="./imgs/libros/' .
		$libro['id'] . '.jpg">';
		echo '<div class="card-body">';
		echo '<h5 class="card-title text-center font-weight-bold">' .
		$libro['titulo'] . '</h5>';
		echo '</div></div></a></div>';
}
```

Lo que hacemos es crear una tarjeta de Bootstrap por cada libro que será un enlace a *libros/id* y tendrá la imagen (los nombres de las imágenes son *id_libro.jpg*) y el título del libro. Como siempre, puedes cambiar el estilo a tu gusto.

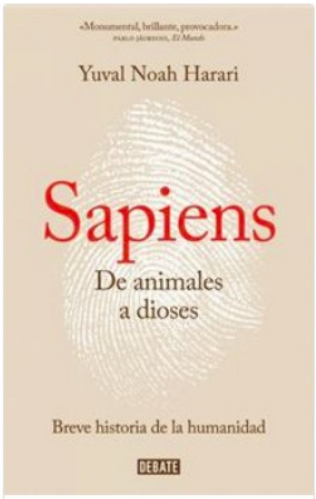
Sólo nos queda añadir la ruta a nuestro archivo de rutas *web.php* y probar que todo funciona como debería.

```
$r->addRoute('GET', $basedir . '/libros', 'Libro@getAll');
```

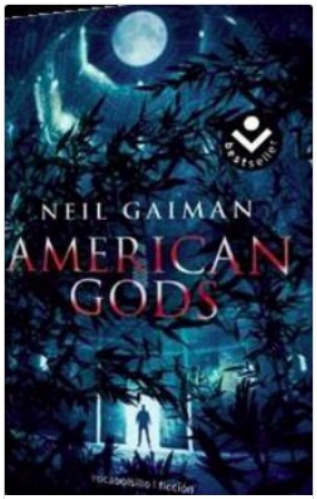
Libros



El nombre de la rosa



Sapiens (de animales a dioses)



American gods

Rutas con variables

Hasta ahora hemos visto como crear rutas estáticas, es decir, rutas que no tienen ningún parámetro que varíe. Ahora tenemos que crear rutas del estilo *libros/id_libro* para mostrar la ficha del libro.

Podríamos crear 3 rutas diferentes (ya que tenemos sólo 3 libros en nuestra tabla): *libros/1*, *libros/2* y *libros/3*, pero obviamente esto se vuelve inviable cuando tenemos tablas con miles o más de registros.

Para solucionar este tema, todos los enrutadores web tienen un método para crear rutas con parámetros que varíen.

En el caso de nuestra librería se hace encerrando el parámetro entre llaves. De esta forma, sólo tenemos que definir una ruta de la forma *libros/{id}* para gestionar cualquier ruta del tipo *libros/1*, *libros/456*, *libros/23…*

También podemos indicar que tipo de parámetro es mediante expresiones regulares, para no permitir, por ejemplo, rutas con un *id* que no sea numérico (en este caso sería *libros/{id: d+}*).

Vamos a crear una ruta de ese tipo para mostrar el detalle de los libros:

```
$r->addRoute('GET', $basedir . '/libros/{id:\d+}', 'Libro@getById');
```

La ruta llamará al método *getById()* del controlador *Libros*, por lo tanto tenemos que crear ese método, pero antes vamos a hacer unos cambios en nuestro archivo de rutas *web.php* para poder pasar parámetros a las funciones de los controladores.

Antes vimos que la variable *$routInfo[]* era un array donde el primer elemento era el resultado de la comparación del patrón con la ruta del navegador, el segundo la acción a ejecutar y el tercero las variables de las rutas. Por lo tanto, para poder leer esa variable sólo tenemos que acceder al 3 elemento del *array* para poder pasársela al método del controlador:

```
case FastRoute\Dispatcher::FOUND:
        $handler = $routeInfo[1];
        $partes = explode('@', $handler);
        $controllerName = '\\controllers\\' . ucfirst($partes[0]);
        $action = $partes[1];
        $controller = new $controllerName($templates);
        $vars = $routeInfo[2];
        $controller->$action($vars);
        break;
```

Ahora ya podemos crear el método *getById()* en nuestro controlador *Libro*:

```
function getById($vars) {
        $libro = modelLibro::getById($vars['id']);
        echo $this->templates->render('libros_ficha',  ['libro' => $libro]);
}
```

La función es casi igual que *getAll()*, sólo que ahora utilizamos el método *getById()* del modelo (como antes, aunque los métodos se llamen igual no son el mismo) para sacar sólo los datos del libro que nos interesa.

Lo único que nos queda es crear nuestra plantilla *libros_ficha.phtml* y comprobar que todo funciona.

```php
<?php
$v->layout('main', ['title' => 'Libros - Ficha']);

echo "<b>Título: </b>" . $libro['titulo'] . "<br />";
echo "<b>Autor: </b>" . $libro['autor'] . "<br />";
echo "<b>Editorial: </b>" . $libro['editorial'] . "<br />";
echo "<b>Precio: </b>" . $libro['precio'] . "€<br />";
?>
```