To err is human - and to blame it on a computer is even more so.

Robert Orben

10 Design patterns for web applications

10.1 Introduction

This chapter deals with two extra design patterns for web applications: **front controller** and **two step design**. To conclude we will combine them into a complete mini-framework fully using MVC.

Also a short introduction will be given for apache's mod_rewrite module and .htaccess files to configure them.

More on front-controller and it's use in other programming languages: `https://en.wikipedia.org/wiki/Front_Controller_pattern`.

10.2 Front-controller

A front-controller is a dispatching controller to which all requests are made. In most cases this will be the `index.php` file.

So every request is made to index.php and with this request extra parameters for the specific controller, action, ... are provided.

This pattern gives the flexibility to boot up the application (setting config variables, start sessions, initialise database connections, ...) and then reroute the request to the specific controller with the action to performed.

10.2.1 Simple way

The simpliest way to achieve this, is by providing a case switch in the front-controller and including the requested controller. A default controller is set when no controller is provided through the `$_GET parameter`.

Within the controller file itself, a new case switch can be used to select the requested action.

Listing 10.1 simple front-controller

```php
<?php

// booting up, setting variables and session
session_start();
date_default_timezone_set ('Europe/Brussels');

define('DEFAULT_CONTROLLER', 'home');

// getting the requested controller from $_GET['controller']
$controller = (isset($_GET['controller'])) ? $_GET['controller'] : DEFAULT_CONTROLLER;

// getting the requested action frm $_GET['action']
$action = (isset($_GET['action'])) ? $_GET['action'] : DEFAULT_ACTION;

switch ($controller) {
        case 'home':
                require_once('ex10.1.home.php');
                break;

        case 'product':
                require_once('ex10.1.product.php');
                break;

        default:
                die("controller $controller could not be found");
                break;
}

?>
```

Have a look at: `https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.1.php` for a complete example.

Source files:
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.1.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.1.home.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.1.product.phps

10.2.2 Object Oriented front-controller

A better approach is to use classes to create controller objects and to create methods which can be used as actions. By doing this, no switch statement is needed and therefore our front-controller is more flexible then the previous example.

Listing 10.2 simple object oriented front-controller

```php
<?php

// booting up, setting variables and session
session_start();
date_default_timezone_set ('Europe/Brussels');

define('DEFAULT_CONTROLLER', 'home');
define('DEFAULT_ACTION', 'index');

class FrontController
{
  public static function run()
  {
        // getting the requested controller from $_GET['controller']
        $controller = (isset($_GET['controller'])) ? $_GET['controller'] : DEFAULT_CONTROLLER;

        // setting the object and including the controller objectfile
        $object = ucfirst(strtolower($controller));
        $file = 'ex10.2.' . strtolower($object) . '.php';
        require_once($file);

        // getting the requested action frm $_GET['action']
        $method = (isset($_GET['action'])) ? $_GET['action'] : DEFAULT_ACTION;

        // creating the controller as an instance of the controller object
        $controller = new $object();

        // executing the method
        $controller->$method();
  }
}

// execution of the front-controller to dispatch
FrontController::run();

?>
```

Because an object oriented approach is chosen, view files are used for rendering the actual html.

Have a look at: https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.2.php for a complete example.

Source files:
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.2.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.2.home.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.2.home.view.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.2.product.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.2.product.index.view.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.2.product.detail.view.phps

10.3 front-controller, without $_GET variables

Using a front-controller like above is a good way to start, but the urls created this way aren't very readable. It would be nicer if you could use an url like this: `http://example.com/controller/action/parame`

Two extra steps are needed to achieve this: 1. Tell the webserver it needs to use `index.php` as the starting point of our application. 2. Get all the parts out of the url and map them to a controller, action and parameters.

The first step is possible through the use of apache's mod_rewrite. For the second step, some php code is needed.

10.3.1 mod_rewrite

In the apache webserver, you can use the mod_rewrite module to remap a url and to reroute the actual request.

Configuration is done in an .htaccess file which is placed in the root folder of your website. Rules in an .htaccess file are also applied to the subfolders of the folder where this .htaccess is put.

Take a look at the example below.

Listing 10.3 example .htaccess to redirect every request to index.php

```
1 RewriteEngine On # activates the rewrite engine
2 RewriteCond %{REQUEST_FILENAME} -s [OR]
3 RewriteCond %{REQUEST_FILENAME} -l [OR]
4 RewriteCond %{REQUEST_FILENAME} -d
5 RewriteRule ^.*$ - [NC,L]
6 RewriteRule ^.*$ index.php [NC,L]
```

The execution can be read like this:
line 1: activate the rewrite engine.
line 2: if the requested filename is a file with a size greater then 0 OR
line 3: if the requested filename is a symbolic link OR
line 4: if the requested filename is a directory
line 5: apply this rule if the condition above is met, case insensitive [NC] and stop if the rule is applied [L]
line 6: apply this rule if none of the conditions above are met

So, if this file is placed in the root folder, every request to a non-existing file or folder is redirected to index.php

If you want to know more, there are plenty of ressources on the internet on creating a .htaccess file. You can e.g. take a look at this one:
`http://net.tutsplus.com/tutorials/other/a-deeper-look-at-mod_rewrite-for-apache/`

10.3.2 getting the controller, action and parameters

Based on the URI to which the request is made, the controller, action and parameters are derived.

E.g. a request URI `http://example.com/controller/action/parameter1/parameter2` contains two main parts after the rewriting via mod_rewrite:

First: the name and location of the called script, which is the index.php because every request is rewritten to it. The value of this file is automatically stored in `$_SERVER['SCRIPT_NAME']`.

Second: the complete URI of the request, stored in `$_SERVER['REQUEST_URI']`.

Keep this in mind and read the piece of code below.

Listing 10.4 translating a URI into controller, action and parameters

```
1  <?php
2
3  define('DEFAULT_CONTROLLER', 'home');
4  define('DEFAULT_ACTION', 'index');
5
6  // strip the controllerfile out of the scriptname
7  $scriptprefix = str_replace('index.php', '', $_SERVER['SCRIPT_NAME']);
8  $uri = str_replace('index.php', '', $_SERVER['REQUEST_URI']);
9
10 // get the part of the uri, starting from the position after the scriptprefix
11 $path = substr($uri, strlen($scriptprefix));
12
13 // trim the path for /
14 $path = trim($path, '/');
15
16 // explode the $path into three parts to get the controller, action and parameters
17 $parts = explode("/", $path, 3);
18
19 // check if the part exists, if not assign the default value
20 $controller = isset($parts[0]) ? strtolower($parts[0]) : DEFAULT_CONTROLLER;
21 $action = isset($parts[1]) ? strtolower($parts[1]) : DEFAULT_ACTION;
22 $parameters = isset($parts[2]) ? strtolower($parts[2]) : '';
23
24 ?>
```

10.4 front-controller, wrapping up

How is it possible to call a specific method in an object and pass a number of parameters to it, and this has to be as flexible as possible?

Rather simple.

First an instance of the controller object is created and then the function `call_user_func_array()` is called, just like in the example below. Have a look at `http://php.net/manual/en/function.call-user-func-array.php` for more on this function.

Listing 10.5 calling functions and dynamically passing parameters

```
1 <?php
2
3 // create an instance of the controller as an object
4 $controller = new $this->controller();
5
6 // call the method based on $this->action and the params
7 call_user_func_array(array($controller, $this->action), $this->params);
8 ?>
```

A mini-framework with a front-controller and a DB connection is available at `https://projectwerk.webontwerp.khleuven.be/projects/dynweb2012examples` subfolder `04framework_frontcontroller`.

10.5 Two step design pattern

Until now, the way to render a view was to include the view file. But what about elements in the layout like e.g. navigation bars or footer layout? Refactoring these to make them reusable is the best way, but there are two different possible approaches:

One is to slice everything up and let the view file call these parts. A second approach is to create a layout file with empty containers and rendering the view in two steps: first fill the containers and in a second step: stuff the layout file.

10.5.1 slicing the layout

The simpliest way to achieve this is by slicing the template html and putting every reusable part in a different view(-function) which can be called by the view that is used by the controller.

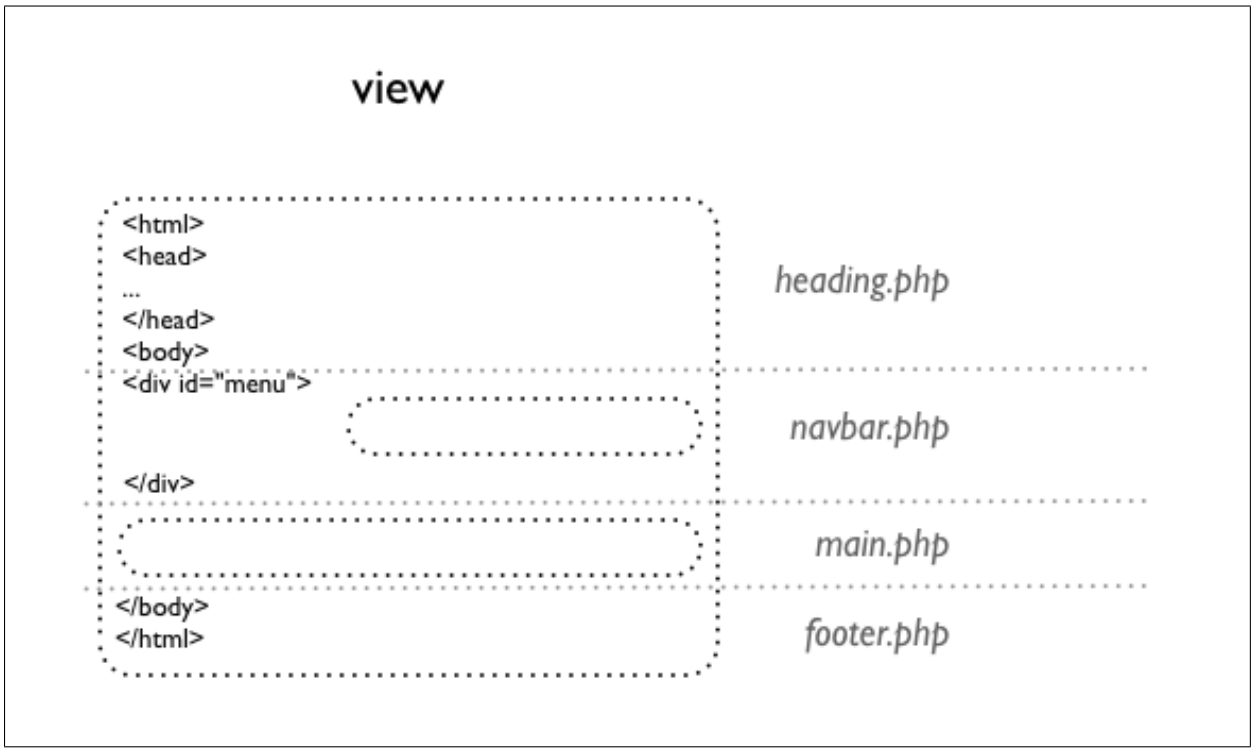

Fig. 10.1 slicing the layout

Listing 10.6 view file with sliced layout

```
1 <?php require_once('ex10.3.layout.header.php'); ?>
2
3 <?php require_once('ex10.3.layout.navbar.php'); ?>
4
5         <h1><?php echo $title; ?></h1>
6
7         <p>Welcome to this sliced example.</p>
8
9 <?php require_once('ex10.3.layout.footer.php'); ?>
```

Source files:
`https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.3.phps`

```
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.3.home.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.3.home.view.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.3.layout.header.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.3.layout.navbar.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.3.layout.footer.phps
```

The layout is built by putting the different layout files together inside the view file.

But the main disadvantage is that there is no flexible way to combine the layout files. E.g. If someone wants to add an extra div-element to the main content, he has to add the opening div-tag to `navbar.php` and the closing div-tag to `footer.php`.

And if someone wants to create a page without the navbar, leaving the navbar.php file out would result in an invalid html-page.

To overcome these disadvantages, the two step design pattern can be used.

10.5.2 Two step design pattern

In a two step design layout, a layout file is used which uses html-tags that open and close in the same file (and thus validates by itself) and has some regions (containers) that can be filled with view files.

A **layout file** is like a backboneframe for your website. In this layout file, regions are defined for partials and one region is defined for the controller view.

e.g.
`https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.4.default.phps`

A **partial** is a view that is rendered independent from the controller, but depends on the layout file. The controller will prepare the data, but the layout will define if and where the partial has to be rendered.

e.g.

```
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.4.partial.header.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.4.partial.navbar.phps
https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.4.partial.footer.phps
```

The **view file** is the view that is defined by the controller. This is the main view file. The data for this view file is passed by the controller.

e.g.
`https://dynweb.webontwerp.khleuven.be/dynweb/examples/ex10.4.home.view.phps`

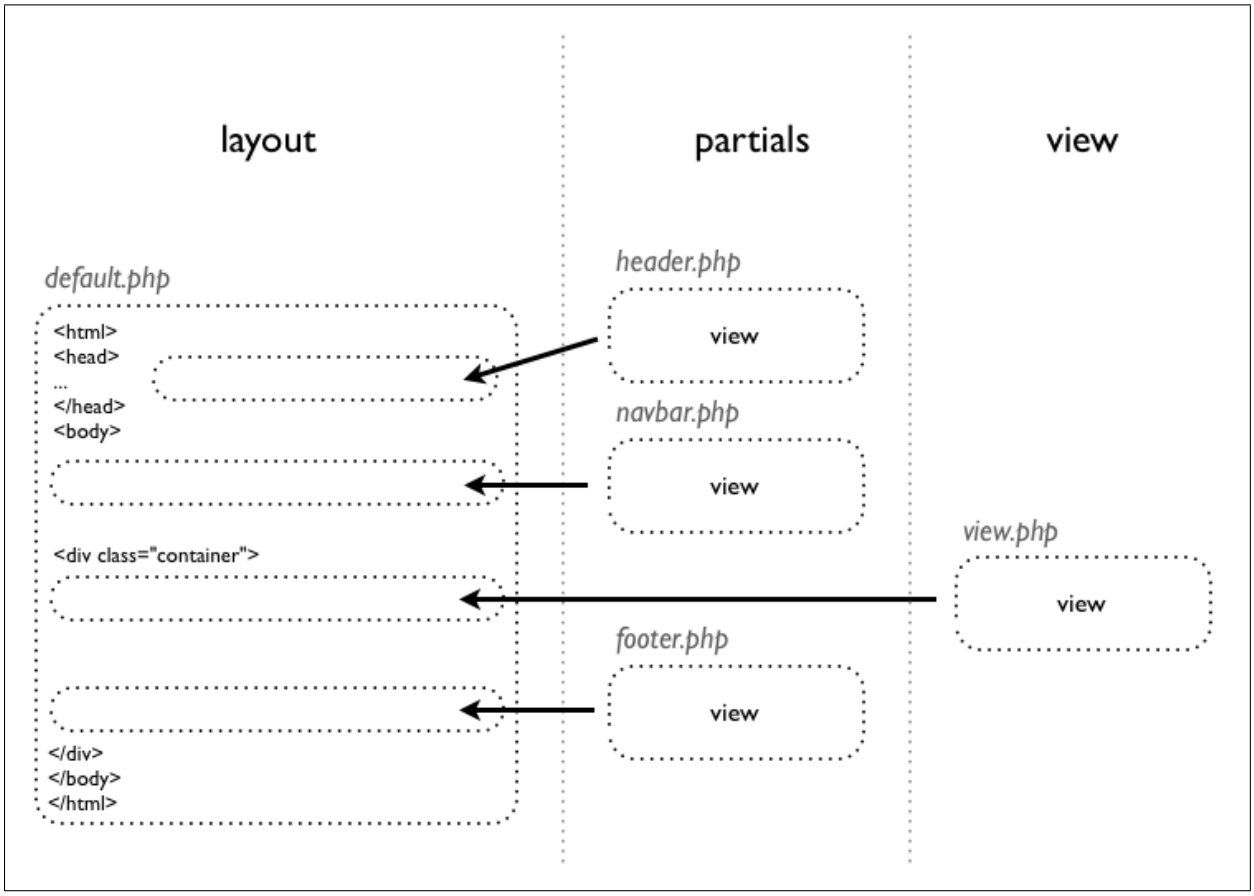

Fig. 10.2 Two step design pattern

The controller sets the data and view file to render and calls the layout.

Listing 10.7 controller for two step design pattern: ex10.4.php

```
1  <?php
2
3  // booting up, setting variables and session
4  session_start();
5  date_default_timezone_set ('Europe/Brussels');
6
7  $title = 'two step design pattern';
8  // setting the view file
9  $view = 'ex10.4.home.view.php';
10
11 // rendering the layout
12 require_once('ex10.4.default.php');
13
14 ?>
```

10.6 front-controller + two step design pattern, wrapping up

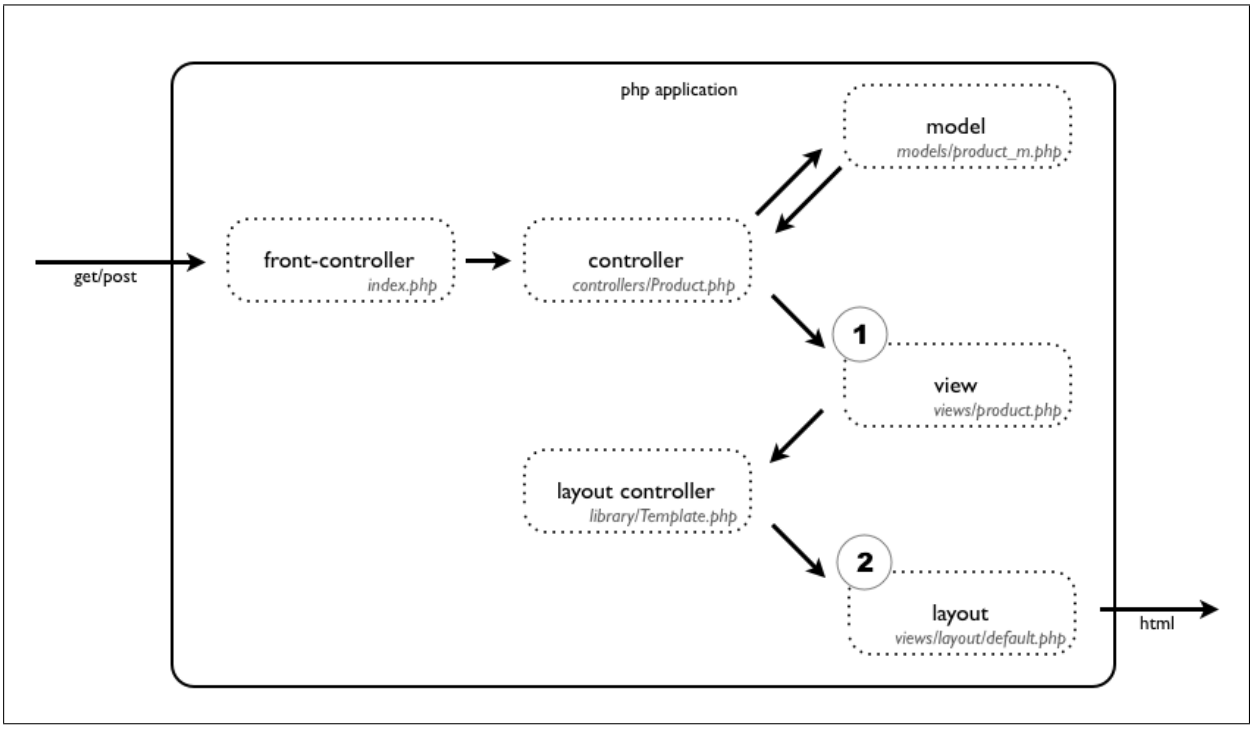

Fig. 10.3 Front controller and two step design pattern

Refactoring this two step design pattern into a template library object and adding some basic URL-functionalities gives a framework for web development. It is available at `https://projectwerk.webontwerp.khleuven.be/projects/dynweb2012examples` subfolder `05framework_template`.

It certainly isn't finished but it hopefully gives a head start in developing a web application.

Exercise 10.1 - refactoring and building your framework

- Combine all the frameworks in the svn repo dynweb2012examples to build your own framework which is capable of:
  - rerouting based on a frontcontroller with flexible creation of new controllers
  - using a database (with a singleton factory) and model classes
  - using a templating engine to reuse layout parts
  - messaging across pages (using something like flashmessage)
  - having some general date-, url-, validation- and other functions
  - ...
- Perhaps you need some abstract database and controller class and make alle database and controller classes inherit from these abstract classes.
- Choose a handy folder structure and naming pattern for your files.