## Introduction

This post introduces the last major building block in our project architecture – the front-end controller. The front-end controller acts as the gate-keeper for the web site. It checks all incoming requests, makes sure the user is authorised to do what is requested and hands control over to whatever controller should handle that request – a process called routing. At this stage, we’ll just introduce the controller. The next post will describe how we can use the apache **rewrite** module to enforce the pattern and reject any request that does not conform to the pattern.

## URIs and URLs

A URI *identifies* a resource on the internet. When the URI refers to the *location* of the resource, we call it a URL. Using a URI gives us a lot more flexibility than using a URL, because we are not tied down to a specific location. We’ll be using URIs in the project. The format of a URI is defined in RFC 3986 (<http://tools.ietf.org/pdf/rfc3986.pdf>). It is defined (p.16) like this:

|  |  |
| --- | --- |
| URI = | scheme ":" hier-part [ "?" query ] [ "#" fragment ] |
| hier-part = | "//" authority path-abempty |
|  | / path-absolute |
|  | / path-rootless |
|  | / path-empty |

For our purposes, the scheme is either *http* or *https*. The *hier-part* gives us the main resource information. The *query* part is an optional set of name and value pairs. The optional *fragment* identifies a place within the resource. Note, however, that the fragment is client-side only (p.25). It can be used by the browser, but is not sent to the server. Returning to the main hier-part, the *authority* is: [ userinfo "@" ] host [ ":" port ]

The *userinfo* part can even have a password (name:password) but never do this – it is deprecated. We won’t be using userinfo. The *port* part is used if the server is listening on a non-standard port. We’ll be sticking (mostly) to standard ports so we won’t use this either. The *host* part identifies the server. This is either a name or an IP address. We’ll mostly use the name localhost, meaning the server running on the local computer.

The part after the authority (*path-abempty* in the RFC notation) is the most interesting part from our perspective. This identifies which resource on our server we should give the user. In PHP terms, there are two parts to this. The first part identifies the running script. This will always be the front-end controller. The second part, called PATH\_INFO, is the part of the URI after the script. This is the part we’ll use for routing. When we implement the rewrite module, we’ll even hide the front-end controller, resulting in what is called a “clean URI”. Our URI pattern will be:

[front-end]/identifier…

## Some examples

To make this more real, consider some possible identifiers:

|  |  |
| --- | --- |
| /admin/customers | Shows a list of customers |
| /admin/customer/45 | Shows customer #45 |
| /admin/customer/edit/45 | Edit customer #45 |
| /products | Shows a list of products |

Because we have separated our URI pattern from script locations, we can implement whatever scheme makes sense to us. Note that using the convention of starting all admin requests with /admin makes it really easy to check authorisation; our routing logic will simple reject any such request if the user does not have administration rights. We’ll cover security in some depth in future posts, but this should give you an idea about how we will implement security in depth in our web sites.

## Core logic

There are two core functions of a front-end controller:

* Figure out which controller should handle the request (routing)
* Make sure the user is allowed to do this (authorisation).

In addition, we typically use it to implement some functionality that is common to every page and to enforce our design standards. To get a sense of this, let’s take a look at the personList script from the last post:

|  |
| --- |
| <?php  include 'lib/context.php';  include 'controllers/peopleController.php';    $context=Context::createFromConfigurationFile("website.conf");  $people = new PeopleController($context);  $people->process();  ?> |

Let’s refactor this to make the class name a variable:

|  |
| --- |
| <?php  // common stuff  include 'lib/context.php';  $context=Context::createFromConfigurationFile("website.conf");  $controller="People";  // run selected controller  $controllerPath='controllers/'.strtolower($controller).'Controller.php'  $controllerClass= $controller.'Controller';  include $controllerPath;  $people = new $controllerClass($context);  $people->process();  ?> |

With this change, the script could handle any controller as long as we set the variable $controller appropriately. In essence, that’s what I’ve done in the attached script. I’ve also added a bit more structure, such as try/catch and logging. I’ve left the issue of authorisation for a future post. Hopefully, you’ll be getting a sense now of where we are heading with this framework.

## Running the scripts

1. If you haven’t done this before, run ***testDatabase.php***. This will create and populate the database table you will need for the other scripts.
2. Run the script ***personList.php***. This runs the stand-alone script for our People controller.
3. Type the URI <http://localhost/pr294Framework/website.php/personList> in your browser. This will run the front-end controller script ***website.php*** and ask it to use whatever controller is right for a person list. Note that if I decided I’d rather call this “people” in the URI instead of “personList”, all I’d have to do is to change “personList” to “people” in the router; I wouldn’t need to change any other scripts. This flexibility allows us to define clean URIs that are both meaningful to users and search engine friendly.