## Introduction

So far, I’ve introduced the basic building blocks of a modern database-driven website. In this post, I’m bringing together some of the things we need to make it all fit together. I’ve written a static content controller which, together with an associated model and view, will let us integrate static content into our website while automatically setting any required links. I’ve also created an automated test framework that will run a series of tests from the file **runtests.php**. This is the main content of this post. I’ve written unit tests for some of my classes to show how this is done. Finally, I’ve added a session object to the context. Finally, I’ve started on a user object that we’ll use for login and authentication, but I’ll leave this unfinished for now. This is just to show that it’s OK for some of the parts of a system to be unfinished.

There’s a lot to get your heads around in this update, so take your time.

## Front End Controller

I’ve changed the front end controller (***website.php***) to allow a home page, static content and people list. I’ve changed the name from personList to people, just because I can. I’ve written a dummy home page with just one link (to the people list) and placeholders for privacy and contact pages. These are ***home.html***, ***privacy.html*** and ***contact.html*** in the html folder.

I’ve updated the .htaccess file to allow access to two scripts: website.php and runtests.php. I’ve not yet implemented a menu bar, but the site is now basically usable. Start by pointing your browser at the folder and watch the URL line as we click on various links. Note that the user can type directly into the address bar and things should just work.

Try typing ***runtests.php*** into the address bar and it should run the automated tests. From now on I’ll create automated tests as I create classes.

## Static Controller

The static controller just crates a model (models/static.php) for the path specified, creates the view and gives it the model and then asks the view to prepare the content. The model (models/static.php) just reads the content from the html directory. The view (views/static.php) just sets the content from the model into the template. This is the basic pattern we’ll use for all controllers.

## Test Driven Development

Every time we write or change code, we need to test it. We also need to re-test the code we’ve previously written to make sure we haven’t broken anything. Typing into a browser is totally unproductive as a testing method, so we need to automate things.

In PHP, I normally use the PHPUnit testing framework. However, there are issues using this with CPIT’s setup, so I’ve written a basic framework we can use. I’m using two directories: **tests** and **unitTests**.The test directory contains script files that define the tests we’ll run. The unitTests directory defines php scripts that we’ll use for unit testing. The script files are in json format and define the following parameters:

|  |  |
| --- | --- |
| name | The name of the test – what’s being tested |
| author | Whoever write the test |
| setupSQL | An array of SQL commands to run before running the test |
| unitTest | The unit test php file to run |

All unit tests inherit from the UnitTest class in the lib directory. This provides basic support for the assertions we need. I’ll add to this class as we identify any further functionalist needed. All parameters are optional, but in practice, we should always have a name and author and should have at least one of setupSQL or unitTest. The best way of understanding the framework is to take a look at the scripts I’ve written.

## Some Unit Tests

The first script (test001.txt) simple sets up the people table on the database. The second test (test002.txt) runs a unit test (***SessionTest***) on the Session class I’ve added with this post (described later). The third test (test003.txt) runs a unit test (***UriTest***) on the URI class I’ve written.

We normally write a unit test for each class we write and by convention name it by appending Test to the class name. The ***SessionTest*** class overrides the ***doTests*** method it inherits from ***UnitTest*** to call methods on a Session object and check that it is doing the right thing by making a number of assert calls. Similarly, the ***UriTest*** class overrides the ***doTests*** method it inherits from ***UnitTest*** to call methods on a URI object and check that it is doing the right thing by making a number of assert calls. This is the basic pattern used by all unit tests.

I mentioned in an earlier post that it might take a bit longer to get started when we use a framework, but productivity is greatly enhanced as we progress. This testing framework is part of that thinking. All we need to do to retest everything we’ve done before is to type “runtests.php” into the browser’s address bar. Retesting by hand is a total waste of our time!

## Session

PHP already has a global object ($\_SESSION), so why am I writing my own one? The basic problem is that PHP normally stores data in a file in a temporary directory on the server. This works fine for small sites (running on one server only) unless the site is hosted on a shared hosting provider. With shared hosting, other sites on the computer might be able to access the file causing privacy issues. With larger sites, we might want to run the site on multiple computers and use load balancing to allocate requests to computers. In both cases, we’ll need to store the session data on a database rather than on the local computer.

Using the build-in $\_SESSION is probably OK for our projects. However, by encapsulating session access in a class, I can avoid scattering access to $\_SESSION across our scripts. That way I can plug-in an alternative implementation using a database if I need it without breaking any existing code. ***Dependency injection*** again!

I’ve followed the usual pattern of defining an Interface ISession with the methods I need and then creating a class Session to implement the interface. The session object is created by the Context class so I can add an entry (later) to the configuration file to enable automatic selection of the session technology I want to use. For now, it’ll just create the standard Session object.

## User Object

I’ve written a basic user class and the front end, ***website.php***, creates one. That’s as far as I’ve gone. The next step is to create a unit test for the user class. Once I’m happy with that, I’ll write the login and authentication logic and build that into the framework.

## Summary

Hopefully, this post will give you a good idea as to where the framework is heading. So far, we can implement a basic MVC design with routing to controllers handled by a front-end controller. We have a template approach that enables us to ***separate the concerns*** of UI design (html and css) from that of the logic of the system, which will be implemented in php. The static controller allows us seamlessly to integrate static content with dynamic content. The unit testing framework will make testing of the system much more productive. Using abstract super-classes for our models, views and controllers minimises the code we have to write for each.