## Introduction

This post builds on the programming values set out in the last two posts. There’s just a bit more groundwork to lay out before we move on to the main architecture we’ll use.

## The story so far

I’ve looked at the core programming values of *don’t repeat yourself* and *separation of concerns*. I’ve introduced SOLID principles and illustrated the open-closed principle (letter ‘O’) and dependency injection (letter ‘D’) by removing dependency on a particular database technology from all scripts and encapsulating it in a reusable class defined by an interface.

In most development contexts, we start by defining interfaces; what is commonly called an API. This is because Interfaces let us reason about functionality at a high level of abstraction without getting distracted by implementation details. Remember, a class defines what a thing ***is***, an interface defines what it ***can do***. In programming, as I life, many very different things can often share a common set of capabilities. However, interfaces are less useful in PHP than they are in many other languages. There are two reasons for this. First, PHP is a loosely-typed language so we lose the benefit of being able to define data types at the interface level. We can, of course, define and document them separately from the language, but we have no means of enforcing this at the code level. Second, PHP is an interpreted language. It doesn’t have a compiler, so there’s no way it can do the things we normally get from a compiler, such as static analysis.

For these reasons, I’ll move away from starting with interfaces in future posts and use classes instead. These will normally be abstract classes that combine an interface definition with a partial implementation. For the rest of this post, I’ll stick with the interface-first approach.

## The Problem with Databases

System designers often talk about *the* database. That’s because, at the abstract system level it is a single conceptual entity. However, developers work in practice with many databases, even in a single system. For example, there may be a production copy of the web site, running on production servers and using a production database. We certainly wouldn’t want to use this database while we are developing and testing. Moving backwards down the development pipeline, the quality assurance team are likely to have a separate database they use for testing before a release is moved to production. The development team may well have a third copy. Moreover, there may be different versions or database schemas and some developers may be working on one schema while others work on another. In short, there are many databases. However, our code needs to know things like server name, user name and password to connect to a database. These are what programmers often call “magic numbers”. Using them in code is a BAD THING because we have to keep changing (and re-testing!) our code; it’s a waste of our time. The standard solution to this is to use a configuration file to specify parameters like these. By convention we use an extension of *cfg*, *conf* or *ini*. It’s also good practice to put our configuration files in a separate sub-folder. For example, this enables the QA team to simply ignore this subfolder when pulling a version to test so they can continue using their own private configuration and don’t have to publish things like username and password. Configuration files are a GOOD THING!

I’ve used a configuration file to specify the database parameters and also made it extensible so that we can put other things in the configuration file too. Rather than inventing a new file format, I used JSON. PHP has good JSON support, so it’s easy to implement and the format is widely known so I don’t have to write a lot of documentation - a VERY GOOD THING from a programmer’s perspective!

## Context

I’ve packages the database into a higher level container object I’m calling a context. The central idea here is that I can give an object a context and it’ll have everything it needs to run. I’ll include in the context things like the URL, configuration and input data. I’ll start by filling these from the standard PHP environment, but later on, I’ll create a *testing context* that implements the same interface and allows us to run automated tests on the same classes without anyone having to sit in front of a browser and key stuff in. This is essential for any serious system!

## URI

A URI is a logical resource identifier, whereas a URL is the location of a resource. We’ll use URIs and I’ll discuss them more in a future post. A URI looks something like this:

http://authority/path/to/program/morepath?optionA=1&OptionB=2#fragment

The first part (http) is the scheme, normally http or https. Others like ftp are possible, but we’ll just be concerned with http. Authority is usually something like “localhost” or a domain name, but it can be an ip address, can have a user prefix (username@host or user:password@host … don’t do this!) and can specify a port number (e.g. :8080).

The rest of the URI, up to but not including the question mark is the path. This has two parts. The first part identifies a resource, the rest, called “path\_info” is extra information.

The part after the question mark is optional. If present, it contains a list of name=value pairs joined by ampersands (&). This is called the QUERY string and PHP lets us access it through the $\_GET super-global. The last part is the fragment. This is only used on the client side and is not sent to the server.

We’re mainly interested in two parts of this. The first is the path info which will tell our site what to do. The second part will be everything before this on the URI, which we’ll call the site. This probably won’t make a lot of sense right now; hopefully it will make sense after we make a couple of changes I’ll introduce in future posts. These changes will be a) putting all requests through a single front end and b) hiding the name of the front end to get a clean uri. Feel free to google “clean uri” if you want a sense of where we’re going.

I’ve packaged up the database, URI and configuration in a Context object and written a script testConfig.php to exercise it.

## Summary

My goal in these theory posts is to create a framework for your project in a step-by-step way, explaining some of the design decisions as I go. I’m taking an agile approach by looking at one feature at a time and creating just enough surrounding context to complete a feature. This approach will require some refactoring from time-to-time, but that’s always the case with an agile approach; if we stick to SOLID principles, the refactoring won’t be too painful. And DRY means not too many places will be affected by any change.

So far, the only feature I’ve completed is a way of accessing the database that is fully-featured, technology independent and run-time configurable. In this series of posts, the first achieved technology independence but was incomplete. The second achieved a more or less complete implementation. This post has added configurability. This post is the last of this architecture-only series. The next post will introduce the MVC architecture and we’ll start to see how some of this comes together.