## Introduction

This post introduces the start of thinking about security. I discuss the ideas of authentication and authorisation and have written a user class that gives an implementation of the ideas. I’ve also written implementations of login and logout, using the MVC architectural style.

## Thinking about security

Security is a big thing for web sites. It’s essential for any serious web site, but sadly, it’s something that is done **really badly** in most web sites. To get a sense of just how bad, take a look at this video:

<https://www.youtube.com/watch?v=CDbWvEwBBxo>

You can also check out the OWASP project at <https://www.owasp.org/>. The core problem is that many websites are designed by people without a computer science background. They are often self-taught and rely on *tips and tricks* they have picked up in internet forums. The net effect is that they are mostly unaware of the risks their clients face.

We will put security at the front and centre of our design. Security is not something that can be bolted-on later; it has to be designed in from the outset. Fortunately, it turns out that the thinking required for security is much the same as that required for good programming so it won’t actually cost us that much in terms of time and effort. There are a couple overarching security ideas I’ll touch on in this post. Hopefully, you’ll see that we have already come a long way towards implementing these. I’ll discuss other ideas in later posts as they come up.

The first big idea is that of an ***attack surface***. A serious site will often have hundreds of PHP scripts. That’s hundreds of possible attack points. When we start to looks at interactions between scripts (e.g. do this to script one and then that to script two), we quickly get to tens of thousands for combinations of two-way interactions and millions for three-way interactions. This is untestable! Sticking to SOLID principles lets us tame these interactions, but we’re still left with hundreds of attack points. Our design uses a front-end controller and an apache .htaccess script to enforce this. That means all accesses must come through a single point and we’ve limited our attack surface to a single script. Of course, that doesn’t make it secure, but at least we’ve started to tame the problem. We can begin to implement a defence in depth by adding measures to this script. We’ve also given the responsibility of database access to a single class: database.php. When we look at threats like ***SQL-injection***, we’ll see that this enables us to provide a solid defence without too much effort.

The second big idea is that of ***authentication*** and ***authorisation***. Authentication addresses how we know the user is really who he or she claims to be. Authorisation addresses what a user should be able to do. We’ll place users into categories based on what they should be able to do: an approach called ***role-based security***. The goal here is to minimise harm and the guiding principle is to grant the minimum rights needed to accomplish the task. Contrast this with the concept among operating systems people of the administrator as a super-user who can do anything! For security, an administrator should be able to administer and do nothing else.

We’ll use a simple conventional approach to authentication: the idea of a username and password. I’ll use the user’s email address as the username. (As a discussion point, what are the strengths and weaknesses of this approach?). The password will be chosen by the user. To minimise the security risk, we won’t store the password in the database, but will store an encrypted version of it using ***one-way encryption***. Using one-way encryption means that we won’t be able to decrypt the password, just test if it was correct. We’ll use the PHP crypt function with a random-salted encryption. This gives a solid defence against rainbow attacks and we’ll use a simple technique of a short sleep delay to make potential hackers pay a high price for brute-force attacks without inconveniencing our genuine users.

I’ve implemented this in a User class with the following API:

|  |  |
| --- | --- |
| getUserID() | Gives the current userID, or null if no user logged in |
| getName() | Gives the current user name, or blank if no user logged in |
| getEmail() | Gives the current user email, or null if no user logged in |
| getDateCreated() | Gives the date and time the account was created |
| getLastLogin() | Gives the date and time of the last login |
| isMember() | True if the user has membership rights, else false |
| isAdmin() | True if the user has administrative rights, else false |
| isValidLogin($db, $email, $password) | Tests if the email and password identify a valid user. Note that this is a class (static) method. |
| login($email, $password) | Logs the user in, throws an exception if invalid. |
| logout() | Logs the user out. |

## I’ve also created a private function, createPasswordCheck, that I may expose later. I’ll deal with this when I look at self-service registration. The O letter in SOLID means that this won’t be a problem.

I’ve implemented the API with two database tables: users and administrators. The users table will end up being self-service and represents members. The administrators table is kept separate because we won’t allow users self-service administrator accounts. I’m using a three-tier approach here (another discussion point here): anonymous users have access only to the public section of the website. Logged-in users have access to the members area and administrators have access to the admin section. I’ve a implemented simple authorisation rule in the front-end controller: All administrator functions will have a path starting with “\admin\”. The controller will respond “page not found” if anyone other than a logged-in administrator tries to access this section.

## Methodology

Note the approach I’ve taken. I’ve encapsulated the required functionality in a User class. The class accesses all dependencies through a context injected in the constructor. I’ve also written a unit test for the class and added it to the schedule of tests that are run automatically. You will be ***required*** to follow this methodology when we start the implementation sprints. A class will only be considered complete if it has been tested and signed off. This will require:

* A completed class
* A completed unit test
* Code review by another team member
* Sign-off by another team member.

Note that I’ve also implemented login and logout scripts using the MVC architectural style. This style, together with SOLID OO principles, is ***required*** for your feature implementations.

Finally, note that I’m keeping a strict separation of html from the PHP code I write; all the html is in the html subfolder. This is an illustration of the principle of ***separation of concerns***. Mixing html and PHP in the same script leads to ***spaghetti code***, which is one of the main criticisms of PHP code. However, this is unfair and really it’s nothing to do with the PHP language. Good programmers will write clean elegant code in any language. Similarly, poor programmers will find a way to write spaghetti code in any language and spend months implementing buggy insecure systems! The disciplines you learnt in SE101 are not just for JADE; they apply to all programming.

## Remaining tasks

You’ll have noticed that I’ve left the menubar unimplemented in the samples so far. That’s because I want the menu to be different for the public, members and administrators. Now that the basic authorisation structure is in place, implementing this is straightforward.