Blogathor

A simple, lightweight, open blog

Content

Introduction

The assignment was to create a database, and a simple PHP script to create and view blog posts. However, what happened when I read the assignment was that I saw "*Oppgaven går ut på å lage en blog der enhver nettbruker kan legge inn et innlegg, og der alle nettbrukere kan se alle innlegg som er lagt inn.*".

This lead me to think "*Oh, each user can add a post, but everyone can read it. Well, at the very least we need a User model then! Otherwise it would mean that everybody could also create posts, right? That's not what we want, is it?*", and it more or less snowballed from there.

Having never written any MVC application from scratch – although I've worked with them in existing applications -- I thought it would be nice to try writing one from scratch. I figured it would improve my learning experience (after all, we do this to learn, right?) and it would be more of a challenge.

I hope it's not a problem that all code (and database tables and columns) are in English instead of Norwegian – this is simply because I'm very used to coding in English and I don't believe coding in other languages is very popular in the industry, and of course, because I don't speak Norwegian.

I also apologize for not using the exact names suggested for database columns, such as "ForfatterNavn" (which I have instead replaced with "first_name" and "last_name", and which are found in the User table); instead, each Post contains an *Author ID*, which is a foreign key originating from the user who created the post, and can be found in the *User* table.

Since I've been working with PHP for quite a while, but recently fallen in love with the MVC-oriented framework of Ruby on Rails, I tried to maintain a Rails-like file structure (without the app/ directory).

I also thought it would be nice to learn how MVC was actually implemented from scratch, as opposed to only work with ready implementations.

Structure & Description

Below I will try to detail some of the naming conventions and design patterns I decided to go with, and how files are organized in the application-root-directory. That which does not fit under Models, Controllers or Views (such as javascript, styleheet, or anything else I think may be good to know about the application) will be found in the *Miscellaneous* column below.

But first, I'd like to specify the versions of PHP, the web server, and DBMS I've chosen to use:

- **PHP**: PHP version 5.5.3
- **Web server**: Apache 2.2.25
- **DBMS**: 5.5.32-MariaDB-log Source distribution

Now that we've got that out of the way, I'll continue to specify the applications structure and other “good-to-know” things about it below:

1. Models

- All models are found in the models/ directory
- Models are (like in rails) "objects of a table". Each model corresponds to a table of the same name (but in lowercase)

2. Controllers

- All controllers are found in the controller/ directory
- Controllers are named [Modelname]Controller (like PostController, UserController, and so on)
- Controllers are require()'d after models.
- The base controller is simply named Controller
- The base controller provides redirect() functionality to derived child-controllers
- The base controller processes the "route" $_GET variable and calls the action/view

3. Views

- Views are found in the views/ directory
- Views can be named anything, as long as they end with .view.php

- Views only contain HTML, and the processing of $parameters (global array returned from the page's controller action with information required for the specific view)

4. Miscellaneous

- Private variables in classes will be prefixed with an underscore (such as $_private_variable).
- Class variables will not be directly accessible by other classes, and will be set and retrieved using getter and setter methods.
- If a stylesheet named [view].style.css is found in the assets/stylesheets/ directory, it will be included after all other stylesheets as to allow CSS overriding on a per-view basis
- For neat and quick CSS/JS, Twitter Bootstrap is used
- For layout in a lovely grid, Twitter Bootstrap's grid is used
- For simple but sweet post formatting, TinyMCE for jQuery is loaded
- All configuration variables (like password salt, site name, site slogan and so on) can be found in config.php
- Database is a singleton class (I know this is a bad practice, but I dont want multiple instances of it... better suggestions are welcome!) and can only have one instance
- Configuration only contains getters, and these are all static (as is all its variables)
- Users may create and edit posts, but they may ONLY edit posts if they are the owners of the post in question. However, administrators should be able to edit any post (currently, they cant).

What I should have done better

This is more or less a disclaimer of things I know, or think I know, that I should have done better. One might say that *hindsight is the greatest teacher*.

1.I should have made the database less of a god object and allowed each model to have more direct communication with the database. By this I mean: ..* Each model should have prepared the statements and run the queries on them. ..* Perhaps using a base model would have been wiser ..* The database class should have only contained a reference to the database connection and the necessary scripts ..* The database class should have contained fail-safe methods for creating tables if they did not exist, and adding default roles and users.

2. I should have let the base controller sanitize and process $_POST variables better and made sure these were passed as parameters to the child controllers actions instead

3. I should have implemented pagination (the backend code is there, I just forgot to add it to the "all" view for posts)

4. I should have used templates and let the views have been classes processing the data (but this felt like abstraction for the sake of abstraction ...)

5. The .htaccess file - I completely failed to write proper rewrite rules for mod_rewrite, so all routes are handled by the $_GET parameter "routes". This might be an easy fix though.

6. I should have implemented better error messages and try/catch'es

7. I should have implemented better checks for when a view/controller/action exists/does not exist

8. I should have added checks for Roles, and made sure administrators had actual administrator rights (right now, roles are pointless :/)

9. Roles should have had its own model (after all, there's a table for it)

10. I should have added the ability to remove posts

11. I should have added the ability to promote/demote a user's role.

12. I should not have followed the YAGNI (*You Aren't Gonna Need It*) principle

Conclusion

Considering my main goal with this, of course, was to learn as much as possible – I would definitely say I have learned quite some new things. Mostly about abstraction, but more importantly how difficult it can be to properly identify a route in the MVC pattern – and how important it is to set a strict rule for in what order parameters come.

It has taught me a bit more about how WordPress' permalinks work, for example; and also a bit more about the importance of using the POST method on forms rather than GET. I also learned a great deal about the flaws of MD5 and SHA1, and why PHP has implemented a rather new function called "crypt" for hashing password and automatically selecting the best available algorithm for doing so.

I have never actually worked with PDO before, although I have been using prepared statements for quite a while (most notably in Ruby), and I had quite the fight with the ? versus :variable approach of doing this. I realized the importance of specifying the PDO::PARAM_TYPE (for example, specifically forcing a parameter to be an integer), and how MySQL/MariaDB doesn't mind integers to be surrounded by apostrophes when seen in a *WHERE* statements, but completely fails when doing the same in a *LIMIT* for example.

I also realized that just because SQL queries can be prepared, they don't necessarily ***have*** to be prepared*.* I did, however, prepare more or less all queries – mainly because it let me worry about other things than escaping variables sent to the database in the PHP code.

All in all, I thought this was a good learning experience, and I sincerely hope future assignments will introduce more things I have yet to learn about.