What I Learned

From Building Project Management Solutions in PHP

Section 1.0 – PHP

1.1 – Authentication

Goal: Create a user authentication system to support signing in users

What I Used: Login Form, Data Sanitation, First Login Check, password_verify, SQL, Error Message, Password Reset Redirect

Summary: User enters their username and password into an HTML form. The form data is then sanitized using *htmlspecialchars()* with ENT_QUOTES set to sanitize single quotation marks in addition to double quotes, ampersands and angle brackets. An SQL query is then executed against the *users* table in the database to determine if the username exists. If the query set returned is empty, an error message is set notifying the user that they have supplied an incorrect value, and the page is re-rendered with the error message in red. If the query set contains a result, a check is made to determine if the user is a first time user by checking a *firstlogin* boolean column. If the boolean is set to false, the *password_verify* function is used to check the user supplied password against the password hash in the database table and upon successful matching the user is redirected to the index page. If the boolean is set to true, a regular comparison check is made against the plain text temporary password in the database table and the user supplied password and the user is redirected to the password reset page upon successful matching. If, in either circumstance, the password does not match then the page is re-rendered with an appropriate error message.

Notes: The password reset page presents the user with a form that asks for their current password and then asks for a new password to be entered twice. A check is made to see whether or not the user has *firstlogin* set to true and chooses whether or not to perform a regular comparison on the temp password or a *password_verify* match on a password hash. If the current password does not match the database, the page is re-rendered with an appropriate error message. If the new passwords do not match each other, the page is re-rendered with an appropriate error message. Upon successfully filling in the form, the new password is hashed using *password_hash* with the algorithm set to PASSWORD_DEFAULT then saved to the *password* column in the database. If the user has *firstlogin* set to true, at this point it will be set to false.

1.2 – Authorization

Goal: Create a user authorization system that prevents site visitors from accessing pages without being signed in and in some circumstances without having the right permissions.

What I Used: Session Variables, SQL, Unauthorized Warning Page, Conditionals in Views

Summary: Whenever a site visitor successfully signs in to the web application, three session variables are set that contain the user ID, the username and a boolean which declares that the browser session is in fact logged in. Every time that a site visitor attempts to access a page with sensitive data, the *loggedin* variable is first checked with *isset()* and then a comparison is made to check if the variable is

set to true. The *username* variable is used to display the currently signed in user in the top right corner beside the *Logout* link, and *userid* is used to correctly populate the *My Projects* view as well as for further authorization checks. For example, if a user attempts to edit a project or a task, or create a new task on a project, a check is performed to verify that the currently signed in user is the author on the project they are trying to edit. If the user fails this check they are redirected to an *unauthorized view* which explains the error. Conditional checks have also been included in the HTML templates to avoid rendering edit links on projects where the currently signed in user should not have permission.

1.3 – Routing

Goal: Create a routing system that directs web application traffic to the correct procedures according to URI and request type.

What I Used: Routes, Router, Classes, Static Methods, Regular Expressions, Controllers

Summary: When a site visitor attempts to access a certain view, their URI and request type are captured and the appropriate methods are called to return the correct HTTP response. When an HTTP request is first received, the Router class calls a static method which creates an instance of a Router object and populates two separate named arrays inside of a named array with two keys. The parent array contains a key called *GET* which holds all URIs associated with *get* requests and their associated action, and another key called *POST* which holds URIs and their associated actions for *post* requests. Each key in the *GET/POST* arrays is a regular expression pattern that allows for wildcard matching in URIs. The sets of URIs and their associates actions are stored in a separate *routes* file which is called after a Router instance is created and performs the necessary actions to populate the routes arrays. Once the arrays have been populated, the Router object calls two individual static methods found on a Request class to return the user submitted URI and request type to a method on the Router class which performs a loop over all of the keys in the appropriate sub-array and calls *preg_match()* on each one of them until it finds a match against the user submitted URI. If a match is found, the appropriate action is called and the correct response is sent back to the user's browser. The actions themselves are stored in Controller classes with individual methods that correspond to a specific URI pattern.

Notes: The bulk of the code that went into making the router system work is courtesy Jeffrey Way and his *PHP For Beginners* tutorial on *Laracasts*. The pattern matching with regular expressions was an extension that was later added through self-study. This was necessary in order to allow for wildcard matching, which in turn allowed for URIs to be created dynamically using the IDs of database records. Project detail pages could be created on the fly without having to manually add a new row to the *routes* file every time that a new project appears in the database.

1.4 – PDO

Goal: Create a database connection by using the PDO class.

What I Used: PDO, Classes, MySQL, SQL

Summary: Whenever the user performs an action that queries the database, a PDO object representing a connection to the database is used to prepare and execute appropriate SQL statements and handle the data being returned in the query set. A bootstrap file is responsible for the initial work of setting up the

PDO by calling a static method on a Connection class which takes values from a *config* file and attempts to create a connection to a database, which in this case is created using MySQL. If the PDO is successfully created, it is stored within a QueryBuilder class which in turn has a number of methods which use the PDO object to prepare and execute queries, manipulating the returned data by dumping it into predefined or generic classes. All of the tables in the database were created manually using MySQL-client and SQL statements and the queries contained in the QueryBuilder methods were also written from scratch and refactored as much as possible. Where needed, the data was returned into custom classes which define methods such as those necessary to format the timestamps into more readable structures, as well as class constants which contain valid entries for columns with predefined defaults.

Notes: The bulk of the code that went into making the Connection and QueryBuilder classes work is courtesy Jeffrey Way and his *PHP For Beginners* tutorial on *Laracasts.* The QueryBuilder class was extended to include all of the custom queries necessary for making this particular application function. Data sanitation was also extended slightly and refactored into its own function to protect the database from SQL injections.

1.5 – Other

Goal: Define other related PHP tools that were utilized in this project

What I Used: Views, Classes, Functions, Namespacing, Composer

Summary:

Views were created by combining HTML and PHP together in php files that defined the overall structure of the document as well as the logic that allows for dynamic rendering of content. Partial views were created to contain reusable code such as the *head, nav* and *footer* sections of the pages. Loops were used to populate lists and drop down selections, and conditionals were used to control what was rendered according to the rules of authentication and authorization.

Classes were utilized in this project to contain related data and the logic needed to manipulate it in a single place. Static methods and class constants were created to allow for logic to be called without having to instantiate class objects where appropriate. All database actions were contained to a single place, as was the logic for routing web application traffic and calling the correct views. Class definitions were created for projects and tasks so that additional logic could be performed on the data contained in their individual database records. Controller classes were created to separate the logic for different kinds of view creation and querying. A *dependency injector* called *App* was created that simply registers objects that the application depends on, allowing them to be called from a centralized location.

Two functions were created for use throughout the application. A *dd* function was added for debugging that takes a single argument, a PHP variable, and calls *var_dump()* on that variable inside of a call to *die()* so that the value of the variable may be inspected at a given point in the application's execution. A *cleanData()* function was also created which takes a named array (typically $_POST) and returns a new named array which contains all of the keys of the original array with the values having been converted by *htmlspecialchars()* with ENT_QUOTES set.

Namespacing was utilized to make it easier to keep track of the classes being used by the web application, as well as giving them more unique identifiers in the event that conflicting class names are introduced later on. Classes can, in most cases, be referred to by their name with the *use* declaration set at the top of the file that is calling them.

Finally, the *Composer* dependency manager was used to keep track of the locations of all of the classes defined in the project to allow for them to be used without having to call *require* on the php files which contain the class definitions.

Section 2.0 – CSS

2.1 – Forms

Goal: Style the HTML forms

What I Used: Unordered List, Labels, Fieldsets, JavaScript

Summary: All of the forms were styled using *unordered* lists with each list item containing a *label* with its display set to inline-block and its related input or *textarea* field, or a *submit button*. The *labels* were given uniform width with text-alignment set to right, and in the case of *textareas* the vertical-align is set to top. *Fieldsets* were used in conjunction with the *legend* tag to promote accessibility, and a single function was added in JavaScript that sets the focus to the first input field on certain forms. All form data is sanitized on the server before anything is ever done with it, and some form fields such as *select* boxes are populated by class constants to ensure that their values are consistent with constraints set on the database level. Hidden input fields are used to submit data to columns with foreign key constraints such as the *userID* column on project records and the *projectID* column on task records.

2.2 – Navigation

Goal: Create a navigation bar for signed in users

What I Used: Unordered List, Display Attribute, Floats

Summary: A navigation bar was created to render along the top of the window when a user is signed in using a conditional check in the HTML template responsible for that portion of the view. The navigation bar is an *unordered list* with no padding, margins or list styles where the list items have been converted to inline-blocks so that they stack horizontally instead of vertically. Using *float: right*, the currently signed in user has their username and a logout link situated at the far right side of the navigation bar.

2.3 – Cards

Goal: Create cards to display info about projects and tasks

What I Used: Tables, Flexbox, Pseudo Classes

Summary: The *My Projects, Project Detail, Search* and *Archive* views have their content displayed as a column that is centered in the page using a flex container where the individual flex items have been rendered as HTML tables and *h2* headers. The tables form cards which contain either a snippet of the project details as is the case with *My Projects, Search* and *Archive,* or the full range of data as is the case with the *Project Detail* pages. Using the *nth-child()* pseudo class, the table rows alternate in background color to aid in readability.

Section 3.0 – MySQL / SQL

3.1 – Creating Tables

Goal: Create tables with appropriate column definitions

What I Used: MySQL-Client, SQL, Foreign Keys

Summary: In order to make this web application function as intended, a database was needed to achieve data persistence. A free and open source fork of MySQL called MariaDB was used as the relational database management system, and a CLI utility called MySQL-Client was used to interact with the database files. Three database tables were created in the database by submitting SQL statements through the CLI client. One table holds information relating to users, including a unique ID, usernames, passwords and a special column which identifies whether or not a user is new and therefore needs to reset their password from the temp password provided by the system administrator. The username column is of the type *varchar* with an upper limit set to 50 characters and a *unique* constraint to ensure that no two users can have the same username. The password column is a *varchar* with an upper limit of 255 characters to allow for any type of hashing algorithm to be used. If *firstlogin* is set to true, it is assumed that the user is still using a temporary password, otherwise a hash containing the salt and algorithm information is stored.

Another table was created to hold data related to individual projects. A unique ID is assigned to each project to aid in indexing. In addition to aiding with indexing, the ID is used to relate the project to its associated tasks and to refer to individual projects in the web application code. A title, client information and description are saved as *varchar* types, while comments are a *text* field which defaults to an empty string. Project status and the department which it belongs to are chosen from a list of preset values using the *enum* type with *Inactive* set as the default status. The date_started field auto populates using *current_timestamp* each time that a new row is created, and created_by is typically filled in by a hidden input in the web form which takes its value from the currently logged in user. Created_by is further constrained using a *foreign key* which ensures that its value must be the ID of an existing row in the users table. Finally, the quote field must be an *integer* type.

The third table holds data pertaining to individual tasks. Like users and projects, tasks have a unique ID that is *auto-incrementing* and a *primary key*. Tasks have a title, description and type which are set as *varchar* type. They have a date_set field which auto populates using *current_timestamp*. There are two *enum* columns to contain task status and who the task is assigned to. Tasks also have a projectID field which is constrained with *foreign key* to ensure that the field is populated with an existing ID from the projects table.

In total these three tables contain all of the data necessary to support user accounts and the creation of projects and their related tasks and their relations to one another. Tasks are related to projects, and projects to users.

3.2 – Custom Queries

Goal: Create queries to retrieve, create and modify rows in database tables

What I Used: PDO, SQL

Summary: Three main types of queries are used in this web application. *Select* queries were used to return records from the database according to different kinds of requirements. This was by the the most prominent type of query, as much of what the web application does it display information to the user. The *search* view queries the project table using the *like* parameter which enters the search query string as a regular expression which checks against each description field in the database to return all records containing the query string in their description. The *archive* view returns all projects in the database and returns them in specialized cards which show the name of the author in addition to other snippets of information. The *My Projects* view is similar to *archive*, with the exception that it returns all projects with a *userID* that matches the currently signed in user. *Project Detail* pages return all details pertaining to individual records and in turn also displays all task records associated to the specified project.

It should also be noted that in every case that a *select* query was executed, the results were loaded into either a generic or a custom object using *PDO::FETCH_CLASS*. This allowed for the results to be manipulated as objects rather than items in an array, which was simpler to reason and created the opportunity to write methods that would perform useful operations on the record data.

The views for editing projects and tasks are similar to *Project Detail* except that they return the record data in form fields to allow for more convenient editing. Upon submitting edits to existing records, an *update* query is used that automatically builds up a statement containing all fields to be updated based on what was submitted.

Finally an *insert* statement was designed to allow users to add new projects and tasks to the database by collecting all of the parameters and automatically creating a statement based on how many parameters were sent in.

Section 4.0 – Closing

In closing, this project served as a great way to become familiar with the basics of PHP and SQL. The initial MVC framework was created by following the *PHP For Beginners* tutorial by Jeffrey Way at *Laracasts*, and proved to be a really great primer for getting involved with PHP development. By extending the framework there were opportunities to learn more about the language such as how to use regular expressions to enable wildcard matching, and how to use built-in functions such as *password_hash* and *password_verify* to support user authentication and authorization.

Creating *views* proved to be a useful experience in writing suitable HTML structures and styling them appropriately with CSS, while also using loops and conditionals in PHP to allow for dynamic content

rendering. The various *forms* created for this application allowed for the practical application of JavaScript as well as pre-population of form fields using stored data and data sanitation using a custom function that relies on the built-in *htmlspecialchars().*

Perhaps one of the most important aspects of this project was the practical experience gained in interacting directly with MySQL / MariaDB and creating custom SQL queries from scratch. The PDO class provided by PHP proved to be a very powerful and flexible tool once learned.

As far as writing production worthy code goes, there are still plenty of great reasons to choose a framework such as Laravel for writing stable, scalable and secure web applications. However, building one up from scratch using only native PHP functions and a database CLI has provided a great experience that has allowed me to improve my problem solving skills and my ability to think creatively outside of the box while familiarizing me with some of the most useful features that PHP has to offer, as well as how databases really work when you aren't leaning on an object relational mapper.