ECWM604

Advanced Web Development

Coursework 2

(https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Admin)

Username: admin
Password: adminpassword

Name: Munaib Hussain

Student ID: w1423768

15th January 2016

Table of Contents

Chapter 1: Evaluation of Technologies

The JavaScript framework Backbone.js played a central role in the implementation of the administration interface, as opposed to framework CodeIgniter which was present within the other sections of the quiz application. This framework in a similar manner to CodeIgniter provided an overall structure and set of class to enforce an MVC structure, thus aiding in the process of avoiding the use of unstructured code. Furthermore, a number of technologies were used in conjunction with Backbone.js to provide a number of functionalities, some of these technologies include: AJAX, JQuery, JSON and a REST API. Learning and utilising the various mentioned technologies had both advantages and disadvantages which made the process of implementing the administration both easier and harder in certain aspects when compared to the other sections of the quiz application in which these technologies were not used.

Backbone.js provided a number of classes that contributed greatly to the implementation of the administration interface, one of these classes was Routers (Backbone.Router). Backbone's routing capability functioned in a similar manner to CodeIgniter's views, however routing was much easier to grasp when compared to views, thus making the implementation of a seemingly multiple paged interface easier and faster when compared to the other sections of the application.

Routing provided the ability to map the segments of the URL preceding a hashtag (#) to a specific function within Backbone.js, as such when a certain URL is visited within my application a specific function was called, which in turn loaded a view which interacted with Models and/or Collections. A major advantage of this feature is that although all the code for the various states of this interface were present on a single page, which differs from the other sections of this application in which the individual pages are delivered by separate PHP files, as a result, I was still able to create a robust interface in which the various states could be navigated via the browser history (forward/back buttons) despite all the information being updated via AJAX. An example of this in use would be "#/Quiz/:id" being used to call the view in charge of questions which displayed all the questions for a particular quiz. Furthermore, the presence of ":id" within the Router URL brings me to another strength held by Backbones Router which greatly attributed to the development of this interface, this being "Dynamic Routing", this feature allowed the ids to be passed from the URL to a given function and then to the view. The use of Routing and Dynamic Routing rather than the views method provided by CodeIgniter made the process of developing a multi-layered interface much more painless. Although Dynamic Routing works in a similar manner to passing data via a GET request, it made the process of passing data and retrieving and utilising the data (e.g. to get questions associated with a quiz) a simpler task as the data could simply passed to the correct function with a simple line of code (due to all the code existing on a single)page, unlike like CodeIgniter where it would be stored in the GET superglobal variable and would need to be retrieved from there from a separate file. Due to absence of the need to pass super-global variables between views within Backbone.js, the learning curve for learning Backbone's routing was far less steep as such it made the process of create the administration interface far more efficient as less difficult when creating the various views within CodeIgniter as Backbone.js required far less steps to pass data , I believe the routing abilities Backbone.js leverages to make the most it's single page layout within the administration interface is a great advantage held over the previously used framework.

A further class provided by backbone was the Model class, this class provided the ability to store the data retrieved through the use of the REST API. This class provided a number of attributes and functionality which aided in the process of managing and manipulating the data retrieved via AJAX calls easier. On the other hand, a counterpart used within CodeIgniter which served a similar purpose was associative arrays, these arrays provided a similar set of capabilities that were much easier to utilise making the process of manipulating retrieved data easier than when Models were used in Backbone.js. The major advantage held over Models by the arrays was the ability to store multiple piece of data, as opposed to Models which are only able to store one set of data at a time, this issue was rectified with by the existence of the Collections class provided by Backbone. However, the need to use two separate classes (Models and Collections) as opposed to the single array within CodeIgniter made CodeIgniter far more simplistic during the development stages as it required less upkeep thus making the process of retrieving and data for later use much easier. However, that said Modals provided a greater set of set of functionality which overshadows its many disadvantages.

Two attributes present in all Models within the administration interface are: “urlRoot” and “idAttribute”, these attributes provide each Model with the capability to automatically derive and generate a URL for the sync with a REST service. Furthermore, the presence of these attributes aided in deriving the correct request method to be used. The presence of these features reduced the need to manually alter URL as the payload sent to model could automatically be used to allow Backbone to make a number of decisions, unlike in the previous sections of this application in which certain functions would need to be manually called and data would need to be inserted into these functions (such as passing data manually from a view to a controller to a model), thus making Models much easier to manage leading to a faster speed of development with less effort. The “urlRoot” attribute allowed the Model to use its personal id to generate a URL through the use of the default “url” function. Since the id of a Model is used to build the URL which specifies how the record can be identified on the backend (via the REST API) an “id” attribute is needed when a payload is stored within the Model as a result of a set or sync. As a default Backbone assumes the id (primary key) is named “id” which is not the case within my system with some examples including “quizId” and “questionId”, as such the “idAttribute” was used to map the models id to the payloads id in order to rectify this issue. A prime example of the advantages of this functionality is present when saving model data using the provided “.save()” function, if an id is present in the data sent to the model a “PUT” request will automatically be sent (resulting in an update), however if the data lacks an id a “POST” request will automatically be used (resulting in an insert), thus saving the need for extra logic to be defined when saving data to the model and consequently sending a “POST” request making the development of the administration interface easier, however this only occurs when the REST API being used as the backend is successfully set up to function correctly. Once Backbone.js has successfully set up to the Backend (Rest API). It requires far less maintenance than required by CodeIgniter as Backbone out of the box provided a number of default operations (GET, POST, PUT, DELETE) accessed via methods such as “.save()”making the process of manipulating the data much less pain staking than writing database queries and functions by hand, making Backbone.js much easier to maintain in the long run during the development process.

As mentioned above Backbones “Collection” Class was used in conjunction with the Model class (since a collection is a set of Models). This class provided the useful ability to store a collection JSON results as part of an AJAX GET Request against the Rest API, thus allowing a number of Models with different information to be stored as opposed to a regular Model which is limited to a single set of data at a time. This Collection class was especially useful when used with views, as when any new changes were made to a Model within the Collection they could automatically be updated through the “.fetch()” method with its new data and could then automatically be passed to function designed to render it within the DOM (achieved via the “.listenTo()”), this was especially useful as it provided the ability for batches of data to be kept up to date. Consequently, the collections class was much more difficult to manipulate than its counterpart within CodeIgniter this was mainly due to the content needing to be accessed being buried several layers deep due to the various attributes within the JSON object created by Backbones Model class. On the other hand, the arrays used within the other sections of this application provided a greater sense of control, as such this is one feature that made the process of developing the administration harder compared to the other sections of the application.

AJAX (Asynchronous JavaScript and XML) played a key role in the implementation of the administration interface as it allowed calls to be made to the RESTful service to retrieve and display data on the page without the need to reload the page unlike the other sections of the application which solely rely on PHP code. The other sections of this application required new views to be loaded in order to display new and related data (such as questions relating to a particular quiz), this was not the ideal when considering user experience as it not only decreased the speed as a whole page would need to be loaded, but in cases of validating data such as forms all data entered would be lost. Consequent, a combination of AJAX and JQuery were a great use in rectifying these issues as relevant data could be retrieved from the backend ".php" files using AJAX whilst the DOM is updated via jQuery. However, whilst AJAX was much more convenient from a user's perspective it holds a number of drawbacks that made development of the administration interface harder during the development process. Whilst AJAX had a number of advantageous attribute as it was used more it became apparent that working with the simplistic PHP code present in the other sections of the application was much easier due to the ability to actively debug issues. The fact that a new view/page was not loaded as a result of an AJAX request make the process of debugging using various means more difficult, thus making the development process harder than when compared to the other sections of the application. Within the first half of this application I utilised various debugging methods to analyse the issues within the PHP code I had written, some of these included: reading error messages provided by CodeIgniter and utilising functions such as "var_dump()" to analyse data structures, these techniques allowed me to get to grips with the various issues within my code and aided in the processes of rectifying these issues. However, due to the AJAX's Asynchronous nature my various debugging methods became unusable and as such when dealing with my highly problematic backend code (due to an imperfect REST API) the process of identify and rectifying bugs and issue became much more difficult, thus making using ajax to develop this application harder to get accustomed to when compared synchronous nature of the other sections.

In combination with AJAX jQuery was also used to the handled the data (JSON) returned via the REST API was a result of an AJAX request, jQuery played an integral role in updating the DOM (Document Object Modal) of the page by adding new: tags, elements and data, thus eliminating the need to reload the page. During the implementation process as mentioned above jQuery handled the manipulation of the DOM by appending new elements to the DOM along with updating already existing data, this was achieved through the use of: "append()", "val()" and "html()". These methods played a central role in the implementation of the administration interface as they not allowed data to be displayed to the administrator, but also provided he ability to retrieve data from the DOM and be utilised within in the code, some examples of this in use would be the fact I was able to create a form and add the appropriate details to the input fields within the form based on the button clicked and the fact I was able to retrieved serialized form data to be passed to the REST API. However, the manipulation of the DOM came with a number of drawbacks that were not present in the PHP code written in the other sections of this application, once the most pain staking issues I came across during this implementation was that of binding and unbinding events. This issue was not present when using PHP to manipulate data as all the events were binding a single time upon the pages load, however due to continuous act of creating and destroying elements/views in the DOM (via JQuery) the existence of "ghost views" became apparent causing events to be fired multiple times, this issue made the process of working with both jQuery and AJAX together much harder. This issue which was taxing to fix and made the process of developing the single page application using AJAX and jQuery more difficult and time consuming.

Rest was an underlying technology which supported Backbone.js throughout the implementation of the administration interface, this technology was used throughout the development stages of the administration interface to communicate to the backend to retrieve and send data using request methods such as: GET, PUT, POST and DELETE through http via URLs (e.g. 'resource/question/question/1'). The major advantage of the REST API that became apparent through its continued use was that of is simplicity, a highly impactful constraint present during construction of the other sections of the application was the thought about the logic needed to make a function work e.g. how to construct a where clause in a database query to retrieve the correct data. The use of a REST API eliminated this constraint as due to its simplistic nature through naming resources within the URL and the use of verbs such as GET and POST sped up the development significantly allowing me to focus on how to handle the data retrieve rather than how to get it, thus making the implementation of the administration interface easier once the REST API had successfully been set up in comparison to the other sections in which specific database queries would need to manually be run each time a certain functionality would need to be achieved as the backend was tightly coupled with the code being written. A further advantage that highlights the ease of use and speed of development granted by the use of the REST API was that of flexibility and the ability to mash data from different sources to create a functioning application. This is clearly present within the implementation of the administration interface which features two different URLs being used in conjunction to retrieve both the quiz and question named with minimal effort, within the other sections of the application, this issue required far more effort through the use of elements such as: multiple where clauses, foreach loops and arrays, as such this point further enhanced the fact that the REST API decoupled myself as the developer from the worries of the backend once the development had started was a great assets in the pace and quality of the code being written as it eliminated a number of issues that existed in the previous sections of the application.

Whilst, the capabilities provided by the REST API were highlighy advantageous by aiding in implementation efficiency through its ability to decouple myself as the developer from the backend, this was only achieved once the backend aspect of the REST API was correctly functioning. Unlike CodeIgniter which provides a set of abstracted database classes which supported and simplified the construction of various database queries, such equivalent was not available with no out of the box solution provided, the code which handles the various request methods and the data being requested and sent over the database had to be created, this was a difficult task as I was not well acquainted with handling URI/URLS and other RESTful aspects whilst I was acquainted with the SQL aspects. Consequently, once developed REST was a great asset, however the implementation of the REST API before it could be used proved difficult when compared to the out of the box solutions provided by CodeIgniter (such as the database classes to make querying the database easy and simplistic). Lastly, JSON was used heavily (as previously mentioned) as this was format of the results returned by the REST API, this format when used in conjunction with jQuery and Backbone was very easy to handle due to the many built in functions provided by Backbone.js and jQuery (such as ".models") making the manipulation of the data retrieved much easier. However in some cases the contents of the JSON object became very crowded and hard to manage which made the process of obtaining the data needed much more time-consuming as they were nested several layers deep, thus making it harder to use when a large quantity of data was being managed.

In conclusion, I believe the combination of the various technologies used in conjunction with Backbone.js for the most part were far easier to get to grips with when compared with their counterparts present in the other sections of the application, thus making the process of developing the administration interface much easier and faster.

Chapter 2: Design and Discussion

2.1. Administration Interface Requirements

1. **The administration interface shall be secured by username and password.**

   1.1. The administration interface shall provide input fields for both username and password to allow administrators to enter their credentials. These input fields should be accompanied by a submit button to submit the entered username and password.

   1.2. The administration interface shall verify the credentials upon submission before allowing access to the administration interface and an error message such as: "Username or Password are incorrect" should be displayed if the credentials entered are incorrect.

2. **The administration interface shall display an index of editable quizzes.**

   2.1. The administration interface shall first display a view listing the various quizzes available to edit on a single vertically scrollable page. Each editable quiz shall be accompanied by the quizzes title, description and image in order to give the administrator an overview of the quizzes details. They will also be accompanied by two buttons: a button to view a quizzes questions and a button to edit the selected quizzes details.

   2.2. The administration interface shall contain a "New Quiz" button to create a quiz atop the page, this button shall prompt the administrator they are able to click the button to add a new quiz.

3. **The administration interface shall display a set of editable questions for a chosen quiz.**

   3.1. The administration interface shall display a set of questions for a selected quiz, each available question will be accompanied by their name, answer and image name, thus allowing the administrator to understand the details for each question. Each question shall also be accompanied by an edit button to edit a particular question and an options button to display a view containing the questions available options.

   3.2. The administration interface shall contain a "New Question" button atop the page, prompting the administrator that they are able to click this button to be able to add a new question for the chosen quiz.

4. **The administration interface shall display a set of editable options for a chosen question.**

   4.1. The administration interface shall display a set options for a selected question within, each accompanied by their name along with an "Edit" button prompting the administrator that they are able to edit a particular option by clicking it.

   4.2. The administration interface shall contain a "New Option" button atop the page, prompting the administrator that they are able to click this button to be able to add a new option for the selected question.

5. **The administration interface shall display a modal to manage quizzes.**

   5.1. The administration interface shall display a modal when either a new quiz, question or option is being created or edited. This modal should contain the appropriate inputs depending on what is being targeted e.g. an input for a quizzes: name, image and description if a quiz is being created or edited, an input for a questions: name, answer and image if a question is being created or edited and an input for a options name if an option is being created or edited. These inputs should be accompanied by a submit button to submit the entered information.

5.2. The administration interface should retrieved the appropriate information for a quiz, question or option and place it in the correct input fields when an edit is being carried out. A delete should prompting the administrator that a delete procedure can be run (on the targeted quiz, option or question) should also accompany the inputs and submit button only if an edit is occurring.

2.2. Database (Additions and Discussion)

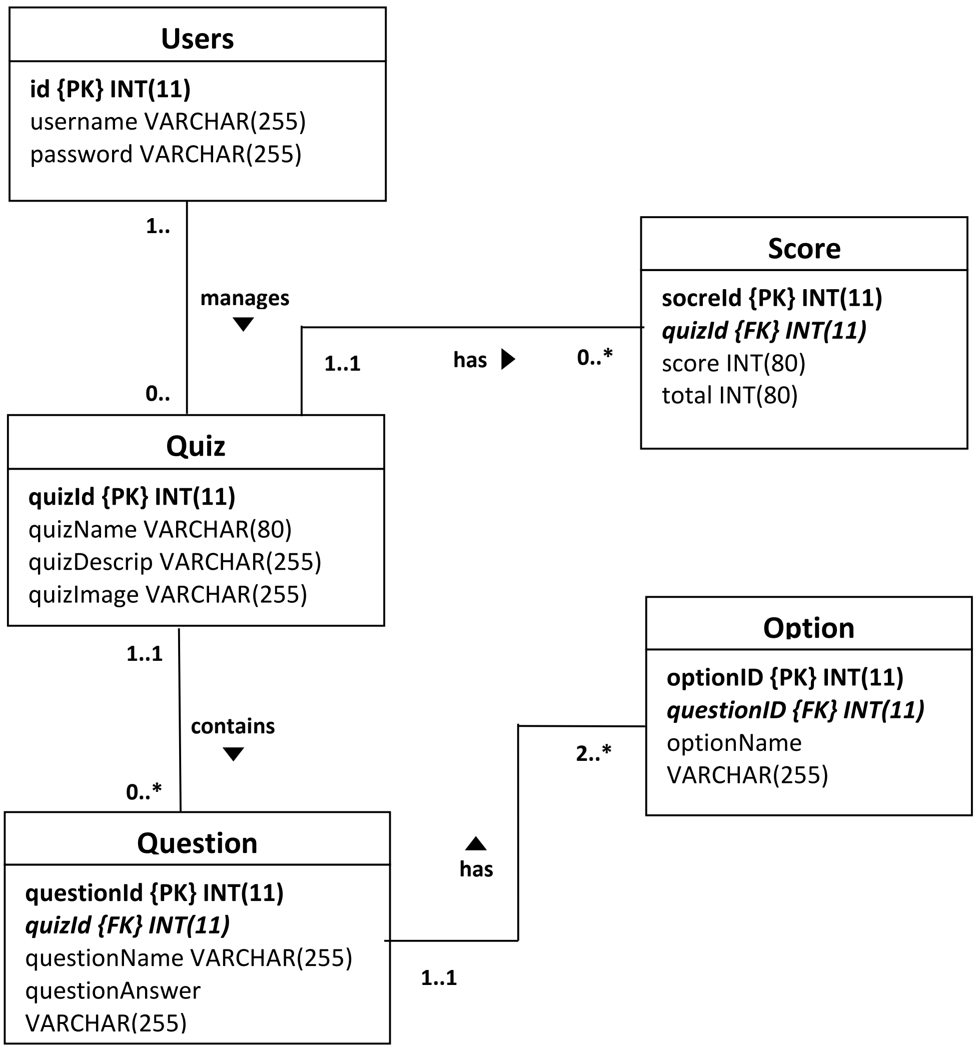


Figure. 1

Whilst no changes were made to the existing tables within the SQL database, a number of new tables were implemented to support the implementation of various functionality outlined within the requirements for the administration interface. A table named "Users" was created to support the first requirement stating "the administration interface shall be secured by username and password", whilst a table named "Score" was created to support the functionality of displaying an average score of all previous quiz attempts to the user.

As previously stated the administration interface should be secured by both a username and password to only allow authorised administrators to access the inner workings of the quiz application. This requirement has been handled by the addition of the "Users" table within SQL database (present in Figure. 1), this table contains a number of columns which enable the functionality to authenticate administrators to either allow or deny access the administration interface. The "Users" table possesses an auto-incrementing primary key present within the column named "id", this primary key makes use of the datatype integer to support its functionality to uniquely identify each user (administrator) present within the administration interface, consequently resulting in the elimination of multiple identical users existing within the table and application. The existence of the primary key "id" aids in ensuring the integrity of the data as it ensures when the details within the "Users" table are being edited or added only a single and the intended row is being targeted, thus avoiding cases where a single user possess multiple sets of credentials or their details being altered by mistake. A further requirement states that a user should be able to make use of a username during the login process, this requirement has been achieved through the addition of the column named "username" also present within the "Users" table, the fields within this column have been assigned a data type of varchar to allow the administrator to have freedom between using a mix between both characters and numbers within their passwords. Whilst the length of the varchar within the "username" fields have been limited to 100 (as shown through "varchar(100)"), the decision to implement this data type is enhanced by the fact that a combination of both numbers makes the username less predictable and harder to guess thus increasing its effectiveness from a security standpoint.

An administrator along with a username is required to have a password in order to access the administration interface (as specified within requirements), therefore a column named "password" has been created to store this specific piece of data. Whilst a simplistic system would store the passwords as plain text within these fields, this approach has been avoided within this implementation to sidestep a number of security flaws, as a result, the secure has algorithm ("sha1") has been utilised within the data being stored within the "password" fields by assigning them a sequence of bytes. The major advantage behind the hashing of the data within the password fields is that although a combination of a username a password is used to authenticate access to the administration interface, without the use of secure hashing the passwords which are the cornerstone to the authentication process would become comprised much more easily when targeting by a potential hacker. An example of an advantages behind the implementation of hashing would become evident in the case of a hacker gained access to the database for the quizzing application and therefore having the ability to access the "Users" table (which stores the passwords and usernames). This attacker would be able to learn a great deal of compromising usernames and passwords from a single glance without any effort being applied on their behalf, therefore, the secure hash algorithm has been used as a precautionary measure in the case of this scenario. Furthermore, the secure hash algorithm the password fields make use of as a preventative measure is named "sha1", the major advantage of the behind the use of the "SHA" family encryption within our system/database is the fact that it is free and it is suited to a relatively small system such as this quizzing application which contains very little sensitive data. Furthermore, to support the storage of encrypted passwords the "password" fields have been assigned a data type of "varchar(255)". The data type of varchar has been used to support the mix of characters (both numbers and letters) that are created as a result of hashing the password and the maximum length of 255 has been purposefully chosen to provide the "password" fields with the capability to store the resulting hashed string.

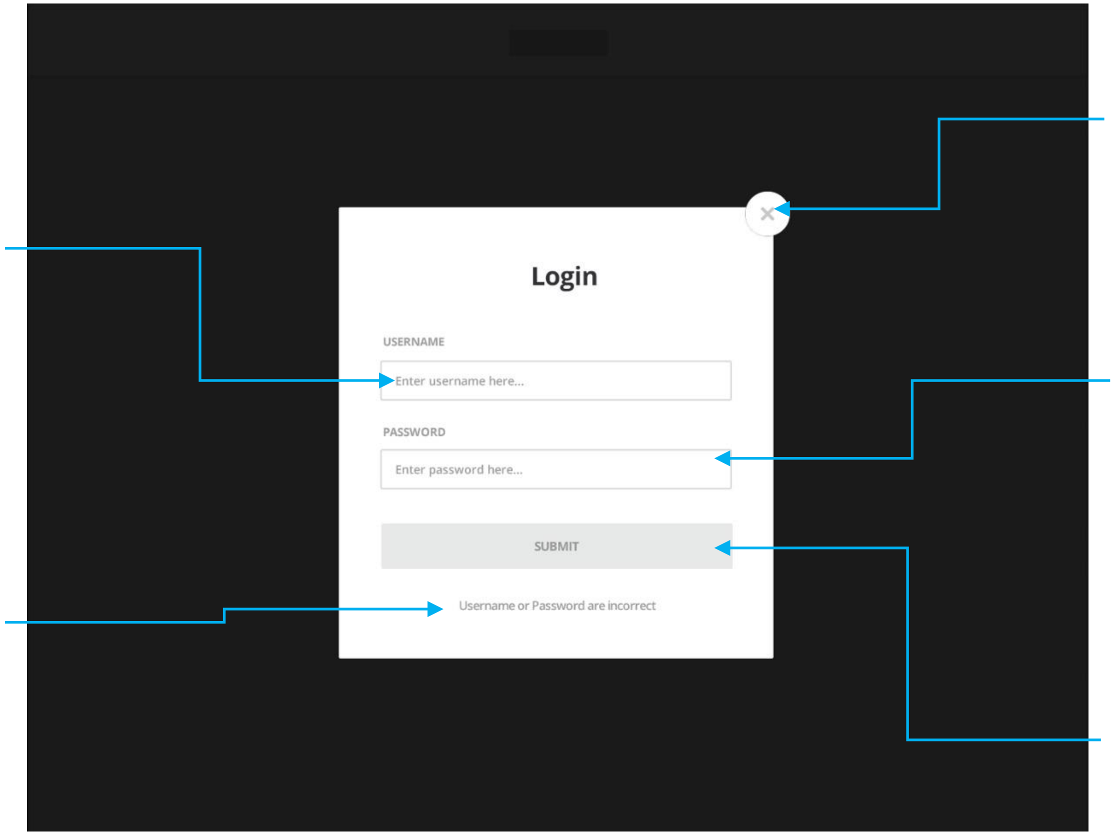
It is stated that the "administration interface will allow the administrator to manage quizzes" this is reflected through the multiplicity present within between tables ("Users" and "Quiz"). The "Users" table makes use of a one to “0..*” since each administrator within the “Users” table can at minimum manage no quizzes but can also manage multiple quizzes and their inherited questions, as such the relationship between these tables signify this functionality. Lastly, in regards to the “Users” table security, the lack of additional columns within this table was a purposeful design choice as the scope of the application does not require additional details that could prove compromising to the administrators, therefore only the data needed is stored. Whilst usernames can be a “real” name, the administrators are not forced to adhere to rule, as such we are able to avoid the need for an administrator within the database to risk their real-name or any other details being released during a potential break. Consequently, these details have been omitted from the database table as they are not required for the systems intended functionality.

As stated above a table named “Score” has been added to the SQL database to support the implementation of the functionality within the requirements stating a user “should also see how they score against the average of all previous quiz attempts”. This table helps achieve this requirements functionality through utilising its parent “Quiz” table’s primary key (“quizId”) as a foreign key, this key has also been named “quizId” within the “Score” table (as depicted in Figure. 1). As a result, when storing and retrieving the scores that have been achieved by users they will be associated with a specific quiz due to the foreign key constraint that has been built (through the use of “quizId”), thus allowing scores to be filtered by their specific quiz to be used when conducting calculations. Furthermore, the “Quiz” and “Score” tables make use of a one to “0…*” relationship between one-another, since a single quiz is not required to have a score associated with it (an example of this situation would be a new quiz which has not been taken yet), however it can also have multiple scores from multiple attempts also. In addition the “quizId” within the “Score” table like its counterpart present within the “Quiz” table utilises the integer datatype to match the parent’s primary key. However, unlike the “Quiz” tables primary key the fact that the foreign is not required to be unique supports the one to “0..*” relationship within the implementation meaning any single quiz can contain a number of scores within the score table. Furthermore, in order to successfully calculate the average for a set of scores attached to a single quiz it is required to be able to store the scores achieved by the users within the database. This has been achieved through the creation of the “score” and “total” columns, whilst the former is used to store the total amount of correct answers, the latter is used to store the total amount of answers available, thus giving a clear view of a user’s score. Both of these columns have been given a data type of integer to ensure the integrity of the data entered as is it certain that the data within these fields will be a number due to their nature, thus allowing calculation to be run upon retrieval. Furthermore, a column named “scoreId” has been added to the “Score” table, this column contains auto-incremented integers acting as a primary keys to uniquely identify each score entered within the “Score” table if needed.

2.3. Mock-ups/Wireframes

Username Input Field
This first input field will store the username for the login attempt. Above the input field will be a label named “username” prompting the admin to enter their username into the input field. Furthermore, this input field will also contain a placeholder reading “Enter username here....”.

Error Message (AJAX)
An error message will be made visible stating “Username or Password are incorrect” upon a failed login attempt. This error message will be appended via jQuery depending on the data retrieved via an AJAX request.



Close Button
An exit button will be present in the top right of the login modal and will be used to cancel the login process and to return to the previous screen. Thus button is depicted by a cross icon.

Password Input Field
The password field will be the second available input field and will contain a placeholder “Enter password here...” along with a label above named “Password”, these aid in prompting the admin to enter the password into this field.

Submit Button (AJAX)
This button labelled “Submit” will be tasked with submitted the data stored within the input fields to the backend using AJAX, this process verified the details and decides whether a user has the appropriate rights to access the administration interface.

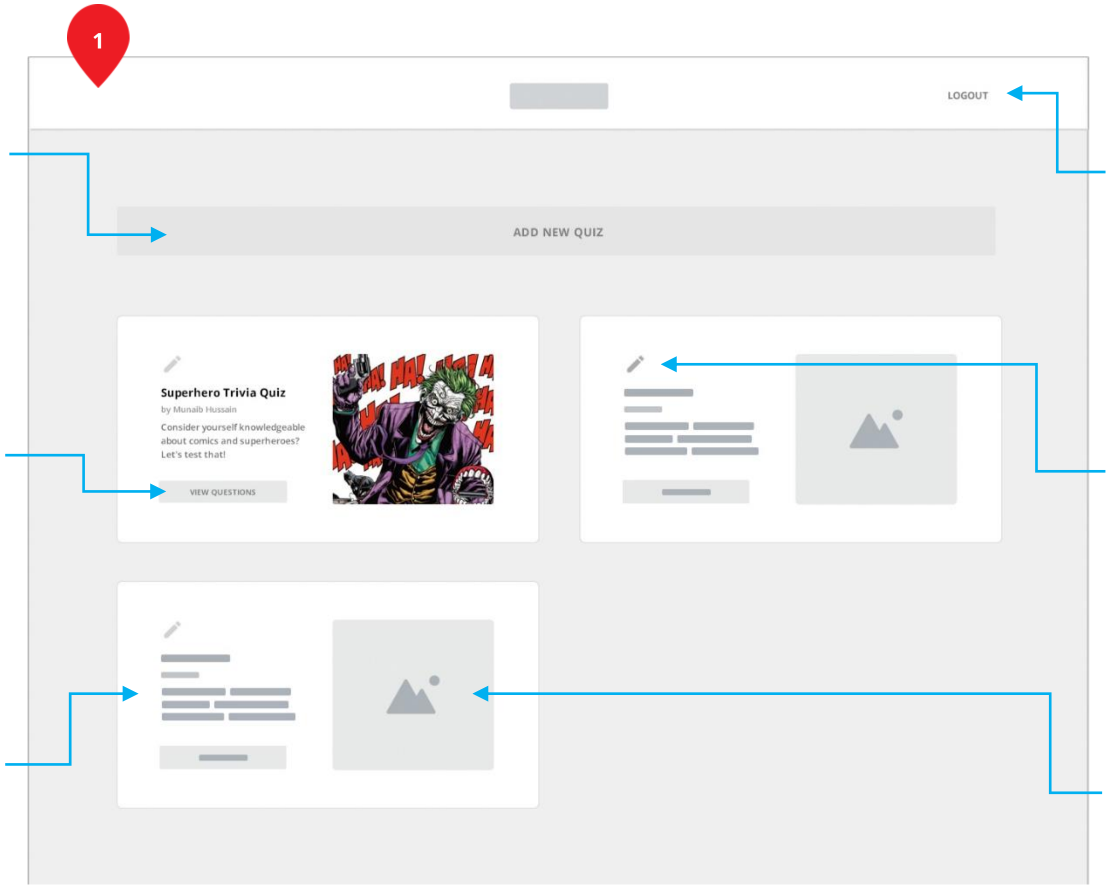
Login View/Modal

This login modal will be used to grant access to the administration interface, as such it contains two input fields with one designed to contain the administrator’s username and one to contain a password. Furthermore, below these input fields is a submit button labelled “submit” which will post these details to be verified using the backend script via AJAX, if the credentials are correct the administrator will be redirected to the administration interface, however, if the credentials are incorrect the user will receive an error message reading “Username or Password are incorrect” which will be added to the bottom of the login modal prompting them as to the reason they failed to access the administration interface.

Add New Quiz Button
A button to add a new quiz labelled "Add New Quiz" will be placed below the header, this button will open a modal to add a new quiz by entering its details into the provided fields.

View Question Button
A button labelled "View Questions" will be placed below each quizzes details and will open a view displaying the chosen quizzes question when clicked by an administrator.

Quiz Name/Description (AJAX)
The name and description for each quiz will be retrieved via AJAX and appended using jQuery, consequently each quiz will have a details. These elements will both be placed below the "Edit" button with the description being below the title.


1
LOGOUT
ADD NEW QUIZ
Superhero Trivia Quiz
by Munaib Hussain
Consider yourself knowledgeable about comics and superheroes? Let's test that!
VIEW QUESTIONS

Header
A header will be present atop the page with a horizontally centered logo within it. A "Logout" button will also be aligned to the right of the header allowing the administrator to sign out from the administrator area.

Edit Button
An "Edit" button (depicted by a pencil icon) will be placed atop the titles for each quiz, this button will also open a modal allowing an admin to edit the chosen quiz. This will be achieved by the provided input fields containing the chosen quizzes information.

Quiz Image (AJAX)
The image associated with each quiz will be placed to the right of the quizzes information, this image will be retrieved via AJAX and appended to the containing element using jQuery, and as such each quiz will have different image that can be changed using these technologies.

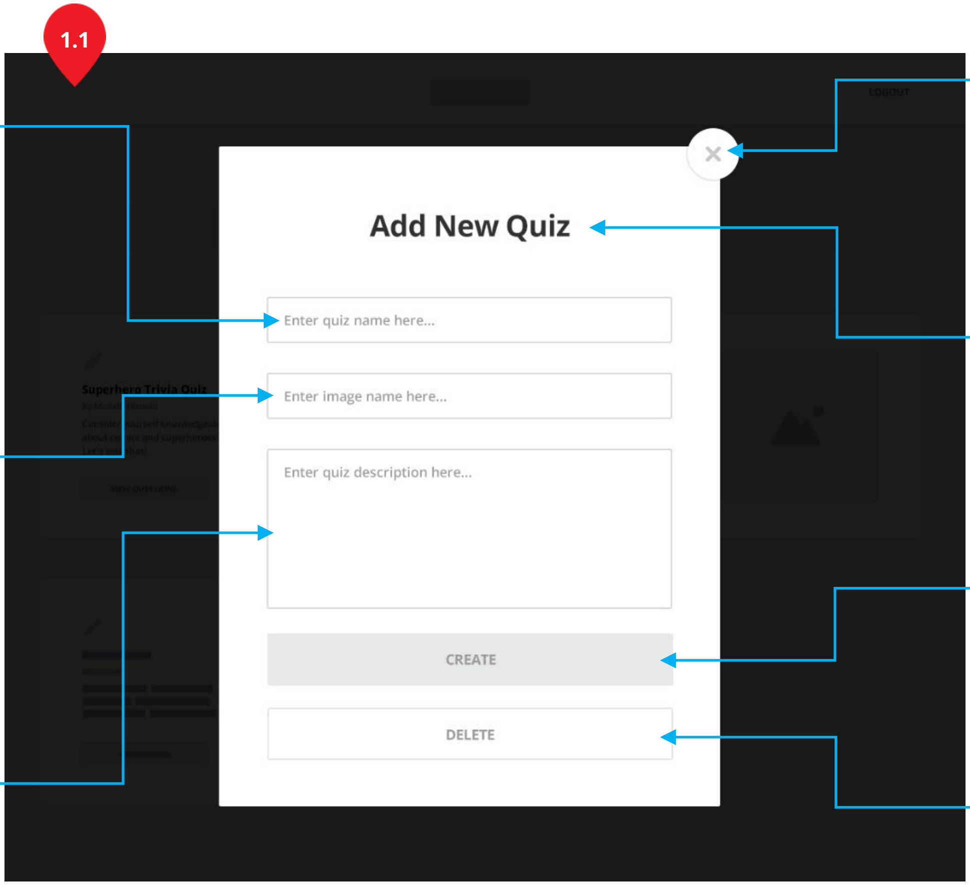
Admin Quiz View

This first view within the admin interface is tasked with displaying all the available quizzes along with their details. Each of the available quizzes (each tile) will be appended using jQuery once their details have been retrieved via AJAX. Therefore, each quiz and their associated names and descriptions will be different to one another. Furthermore, each quiz will be accompanied by an "Edit" button and a "View Questions" button, the latter will display all the questions for a particular quiz, and whilst the former will result in a modal being displayed (achieved through the use of jQuery) to allow for a quiz to be edited. Furthermore, the "Add New Question" will be present to allow the addition of a new quiz. Lastly, upon update of any of the quizzes all the details for the quizzes shall be re-retrieved using AJAX and re-appended using jQuery to display the new quiz information for each quiz, an example would be the title of a quiz being altered from "Super Hero Trivia Quiz" to "Superhero Trivia Quiz" upon the update being completed all the details for the quizzes shall be retrieved and displayed once again.

Quiz Name Input **(AJAX)**
An input to enter/update a quizzes name will be placed here. If the "Add New Quiz" button was clicked to open this modal then the placeholder "Enter quiz name here..." shall be present. However, if an "Edit" button was used the quizzes name (such as "Superhero Trivia Quiz") shall be retrieved via AJAX and placed in this input.

Quiz Image Input **(AJAX)**
An input to enter/update a quizzes image name will be placed here. Similar to "Quiz Name Input" depending on what button selected either the placeholder "Enter quiz description here..." or the image name for the chosen quiz (e.g. super- hero.jpg") shall be present.

Quiz Description Input **(AJAX)**
This input allows the creation or updating of a quizzes description. Therefore, it contains the description of the chosen quiz (which is retrieved through the use of AJAX) when an "Edit" button is selected and the placeholder "Enter quiz name here..." if the "Add New Quiz" button is selected.



Close Button
This button (represent by a cross icon) on click will result in the cancellation of the create/update process for a quiz and will remove the modal, thus returning the admin to the previous screen.

Manage Quiz Modal Title
Depending on whether the "Add New Quiz" or an "Edit" button (beside a quiz) was selected to open this modal, the title will either be "Add New Quiz" or "Update Existing Quiz" (this is achieved via the use of jQuery).

Create/Update Button
This button shall submit the details in the input to the Rest API and will be labelled "Create or "Update" depending on the button selected to open the modal.

Delete Button
This button and will remove the quiz in which the edit button was clicked. Consequently, this button will only be visible when modal was opened using the "edit" button.

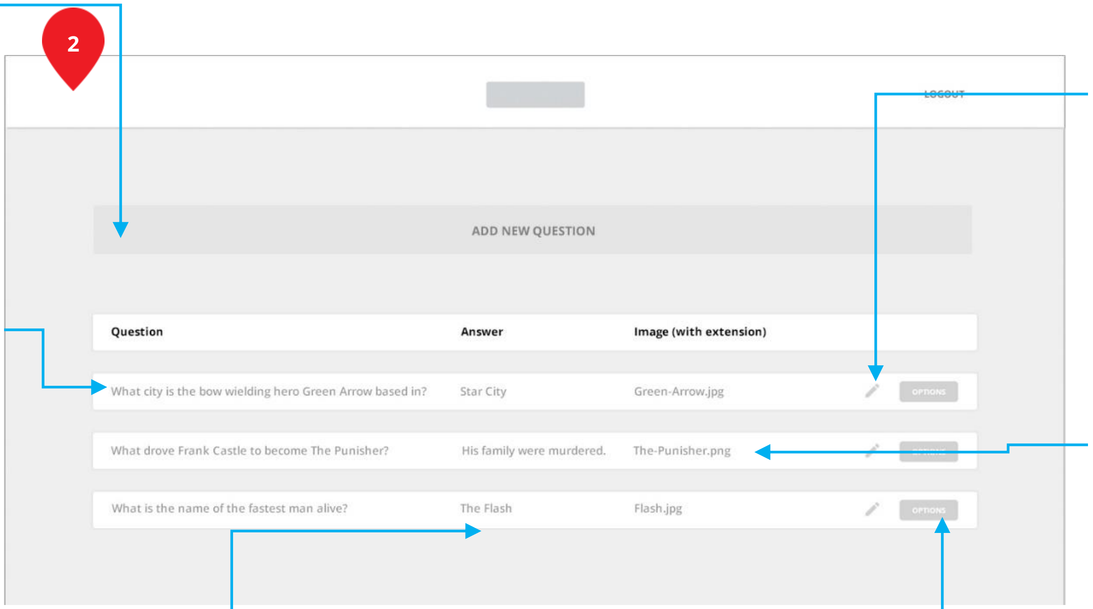
Create/Update Quiz Modal

This modal shall be displayed within the "Admin Quiz View" to allow the administrator to either update or add a new quiz, this is achieved by altering the contents of the modal using both AJAX and jQuery when a specific button is selected to open this modal. If the "Add New Quiz" button is used the inputs would be empty and contain there specific placeholders, however if the "Edit" button is selected the appropriate values for the chosen quiz will be retrieved via AJAX and placed within the input using jQuery to be edited. Furthermore, through the use of jQuery a number of other details shall be altered such as the title being either "Add New Quiz" or "Update Quiz" and the submit button being labelled either "Create" or "Update". Lastly, if the "Edit" button is used to open the modal a "Delete" button shall also be added to allow the deleting of the chosen quiz.

Add New Question Button
A button labelled "Add New Question" will be placed at the top of the page, this will also open a modal allowing for the admin to enter the details for a new question and add the question to the selected quiz.

Question Title (AJAX)
The first item in each row shall be the questions title (e.g. "Which city is the bow wielding hero Green Arrow based in?". This information will be retrieved using AJAX and appended to the row using jQuery (upon update this information shall be altered)

Question Answer (AJAX)
The second item in each row shall be the questions answer (e.g. "The Flash". This information once again be retrieved and displayed using both AJAX and jQuery and will so be retrieved once again if a question is either updated or added.



Question	Answer	Image (with extension)
What city is the bow wielding hero Green Arrow based in?	Star City	Green-Arrow.jpg
What drove Frank Castle to become The Punisher?	His family were murdered.	The-Punisher.png
What is the name of the fastest man alive?	The Flash	Flash.jpg

Edit Button
This button shall be the penultimate button in each row and will open a modal allowing the admin to edit a questions details (Name, Answer and Image) by providing the appropriate input fields filled with the corresponding information (also achieved by using AJAX to retrieve the information and jQuery to manipulate the retrieved information to place it in the fields.

Question Image Name (AJAX)
The third item in each row shall be the image name (along with its extension) which will correspond to each questions image.

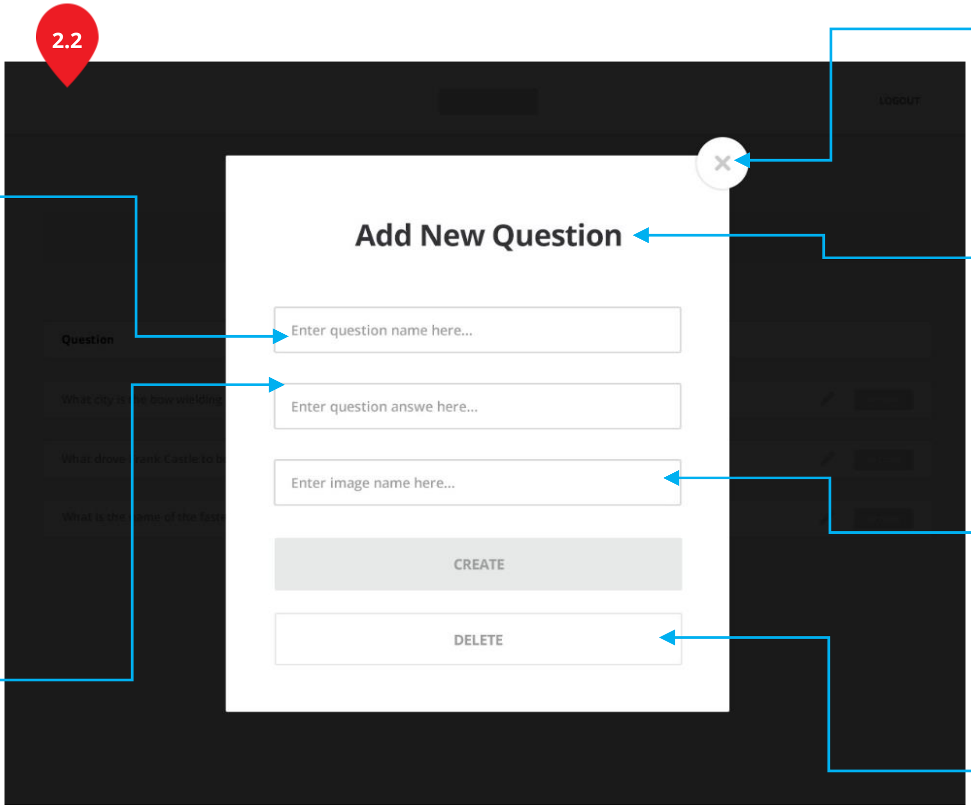
Options Button
A button to view a questions options will be placed at the end of each questions row, this button shall be labelled "options" and will open a new view containing the options and their

Admin Question View

The second view shall display all the questions and their details for the selected quiz for an admin to manage. A row will be added for each question through the use of jQuery and will contain the details for each question (title, answer and image) which will retrieved from the REST API through the use of AJAX. These rows and their details will be re-appended when a question is updated or added (by once again using AJAX and jQuery) to ensure up-to-date information is displayed. Within each row an "Edit" button will also be present allowing the admin to edit the chosen questions details through the use of a modal which will contain a set of prefilled input fields containing the details for the chosen question (achieved through AJAX retrieving the data and jQuery appending it to the input fields). Furthermore, an "Options" button will also present that will allow a view containing the options for the chosen question to be viewed.

Question Name Input (AJAX)
An input to questions name or update an existing questions name shall be the first available input field. The input field shall solely contain the placeholder text "Enter question name here…" if the manage questions modal was opened through the admin clicking the "Add New Questions" button, however if the "Edit" button present in the rows for each available question was clicked instead then the questions name shall be retrieved via AJAX and placed into the input field using jQuery e.g. "Who is the fastest man alive?".

Question Answer Input (AJAX)
An input to allow the administrator to enter or update a questions answer shall be the second available input. Once again if the "Edit" button was not used to open this modal then it will contain the placeholder "Enter question answer here…" However if it was used, then the chosen questions answer (e.g. "The Flash" shall be placed in the input field.



Close Button
This button (represented by a cross icon) will cancel the addition or update process for a quizzes question and return the admin to the "Manage Questions View".

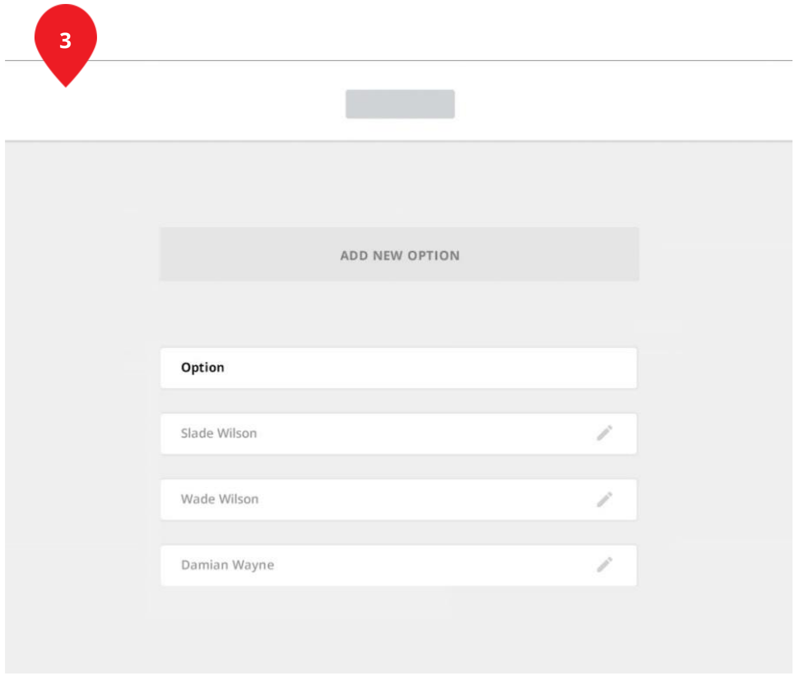
Manage Question Modal Title
Depending on button selected ("Add New Question" or the "Edit"), the name of the title can be either "Add New Question" or "Update Existing Question". The changing of the titles shall be achieved using jQuery.

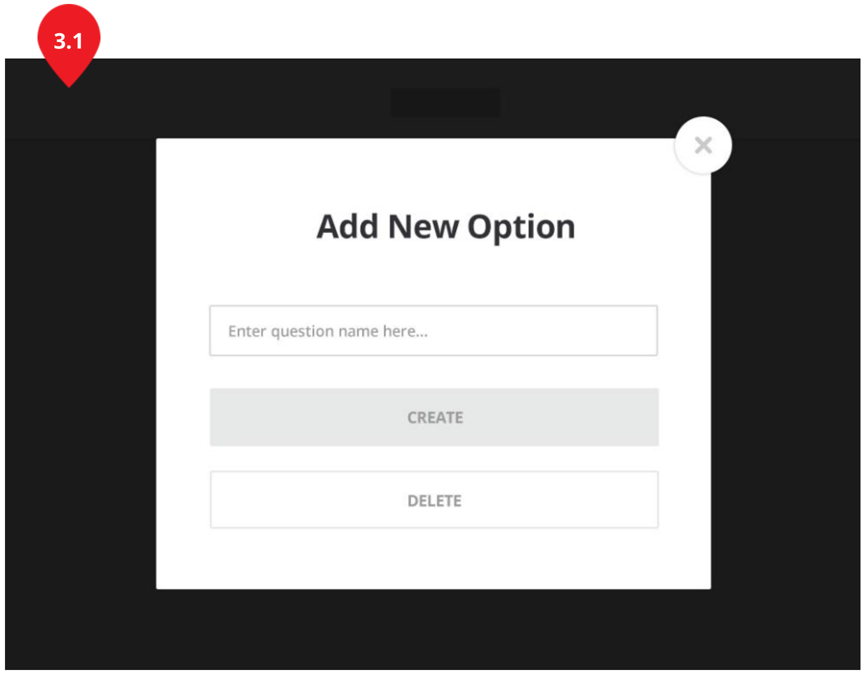
Question Image Name Input (AJAX)
The last input field shall allow the user to enter/edit the questions image name. Depending on what button was clicked to open the modal this input will either contain the placeholder "Enter image name here…" or a questions information (such as: "flash.jpg").

Delete Button
This button and will remove the question attached to the "Edit" button clicked. Therefore, this button when an "Edit" button was used to open this modal.

Create/Update Question Modal

This modal manages the questions for a chosen quiz and is present within the second view ("Manage Questions View"). This is displayed when either the "Add New Question" or "Edit" buttons are selected (achieved through jQuery's ability to append classes). The inputs present this modal link to a chosen questions: name, answer and image name, consequently, when the "Add New Question" button is selected these inputs shall be empty with their appropriate placeholders, however, when an "Edit" button for a particular question is selected, the questions details are retrieved via AJAX and added to the corresponding input fields to be edited. Furthermore, similar to the modal present on the previous screen through the use of jQuery a number of alterations are made to other elements, these include: the title either being: "Add New Question" or "Update Question" and the submit button for this modals form being "Create" or "Update" depending on the button selected to open the modal. Lastly, a delete button is also appended through the use of jQuery if the "Edit" button is used to open the modal.

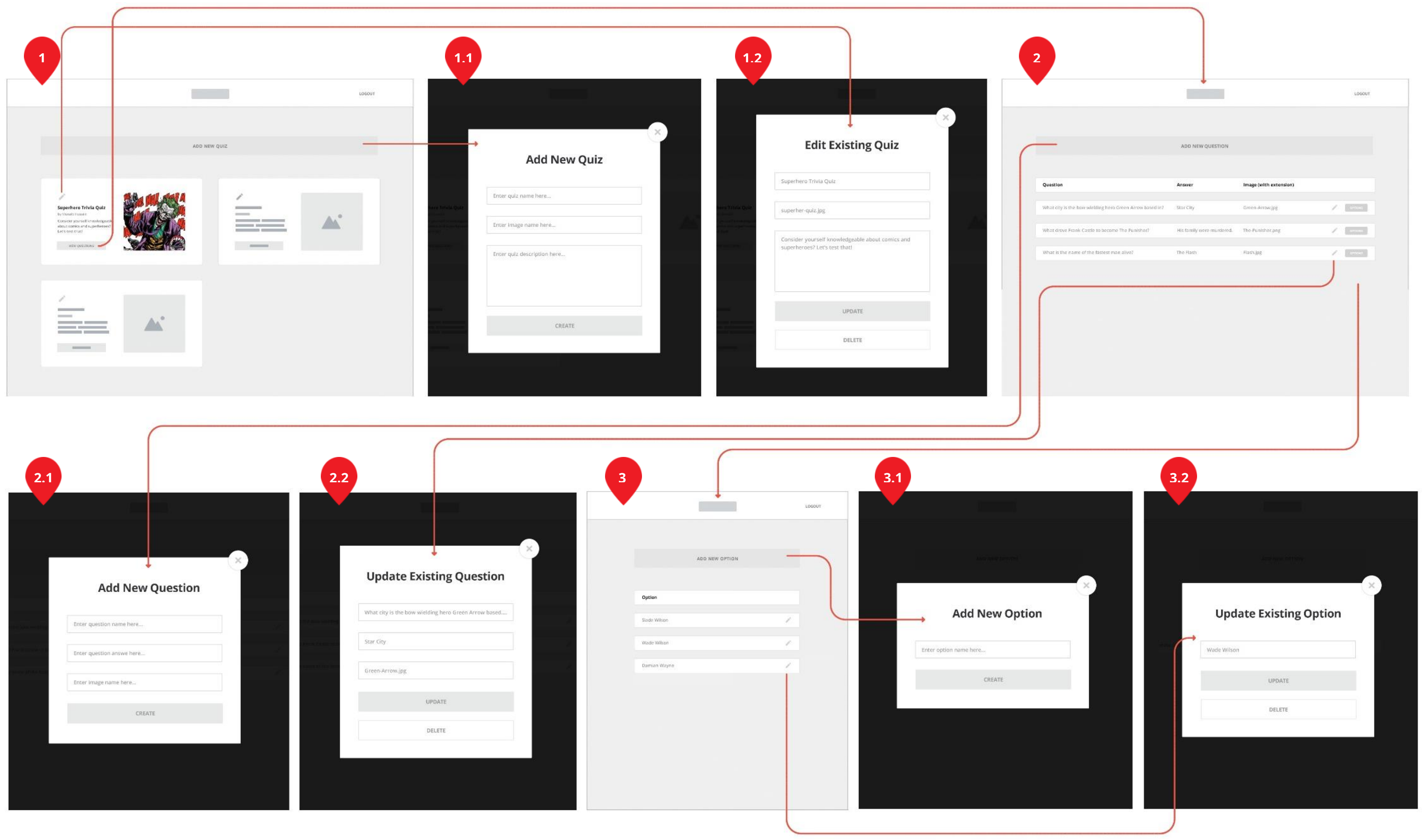




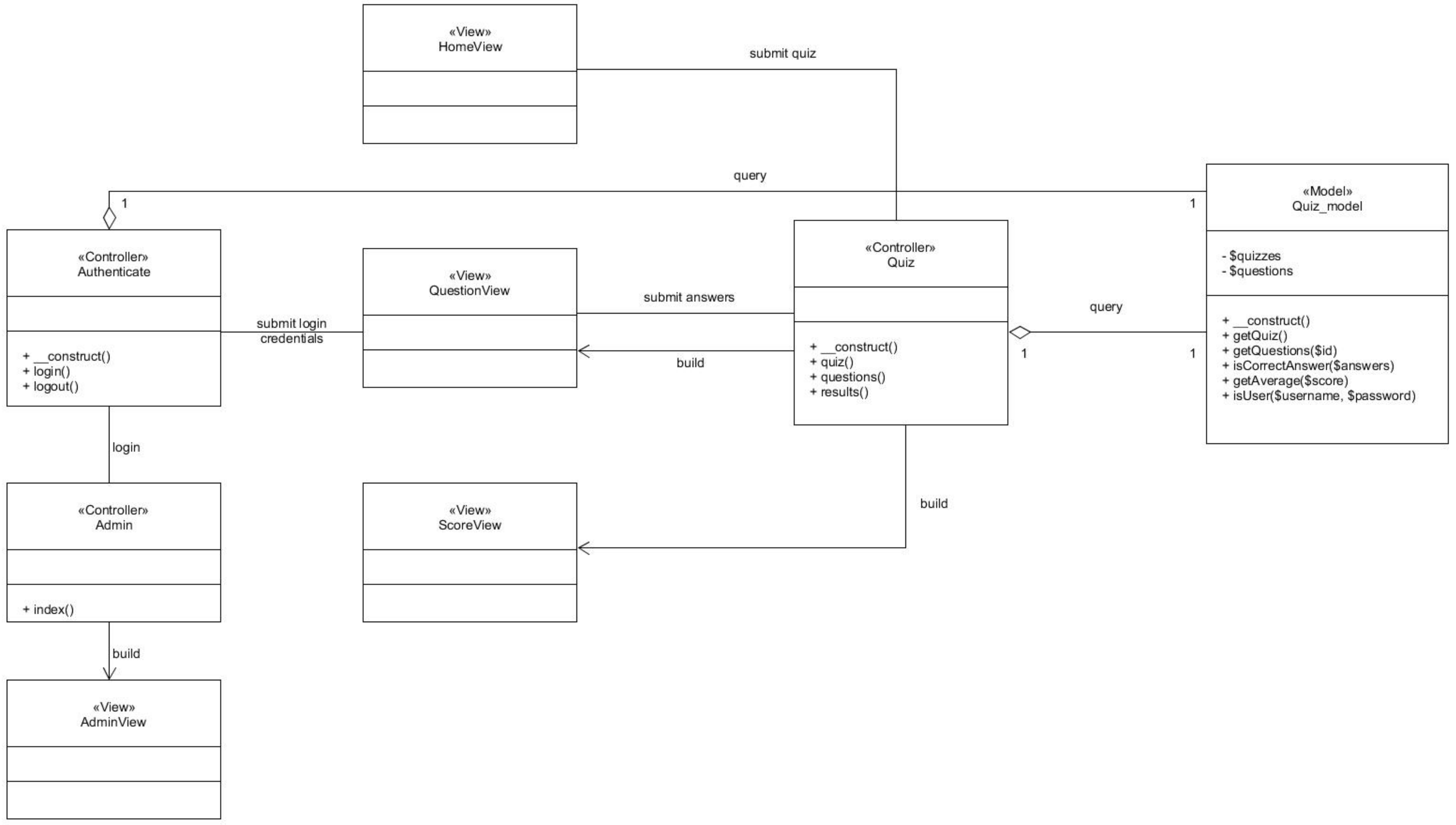
Admin Option View and Create/Update Option Modal

The final view present within the flow of the administration interface is the "Admin Options View" (as shown on the left), this view similar to "Admin Question View" as it will append a row for each option available for a question, these rows will be added using jQuery and the data within them will be retrieved using AJAX. Furthermore, also present within the "Admin Options View" are the buttons "Add New Option" and an "Edit" button which is present within each row and once one of the mentioned buttons are clicked a modal (as shown on the right) will be opened. The modal will contain a single input field with the placeholder "Enter option name here…" if it opened using the "Add New Option" button signifying a new option is to be added, however if the "Edit" button is selected the option name will be retrieved through the use of AJAX and placed within the input field to allow the admin to edit it. In addition, depending on the method used to open the modal (whether an option is being edited or added) a number of differences will be present (achieved through the use of jQuery), the title of this modal will either be "Add New Option" or "Update Excising Option", the submit button will be labelled either "Create" or "Update". Lastly, a delete button labelled "Delete" will be appended using jQuery when the "Edit" button is selected, this button grants the admin to remove the chosen option. The contents of the "Admin Options View" will also be altered upon the addition or editing of an option, consequently resulting in the new set of information/data being retrieved through an AJAX call and the retrieved information being manipulated and re0appended to this view using jQuery.

2.4. Administration Interface Basic Flow (After Login)



2.5. Class Diagram



Chapter 3: REST API Documentation

3.1. Quiz Rest Calls

GET	/resource/quiz/quiz/	
Purpose	Get all available quizzes.	
Response	On success, the HTTP status code in the response header is 200 OK.	
Returns	A list of all available quizzes.	
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/quiz/quiz	

GET	/resource/quiz/quiz/{id}	
Purpose	Get the quiz with the specific id.	
Response	On success, the HTTP status code in the response header is 200 OK.	
Returns	A single matching quiz.	
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/quiz/quiz/2	
Request	**Parameter:**	**Value:**
	{id}	A quiz id (a selection of ids can be found from the above quiz call e.g. "2").

GET	/resource/quiz/title/{query}	
Purpose	Get the quiz with the specific title.	
Response	On success, the HTTP status code in the response header is 200 OK.	
Returns	A single matching quiz.	
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/quiz/title/Comicbook Trivia Quiz	
Request	**Parameter:**	**Value:**
	{query}	A quiz title (the titles for each quiz can be found in from the results of the first quiz call e.g. "Comicbook Trivia Quiz").

POST	/resource/quiz/quiz/	
Purpose	Add a new quiz.	
Response	On success, the HTTP status code in the response header is 200 OK.	
Returns	The single result containing the details of the added quiz.	
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/quiz/quiz/	
Request	**Parameter:**	**Value:**
	quizName	The name/title of the new quiz.
	quizDescrip	The description of the new quiz being added.
	quizImage	The image name (followed by its extension) for the new quiz.

PUT	/resource/quiz/quiz/{id}
Purpose	Update an existing quiz.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	The single result containing new details for the updated quiz.

Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/quiz/quiz/1/
Request	**Parameter:** **Value:**
	{id} quizId — The id of the existing quiz (this can be found in the results of the first quiz call shown).
	quizName — The new name/title of the quiz.
	quizDescrip — The new description of the existing quiz.
	quizImage — The new image name (followed by its extension) for the existing quiz.

DELETE	/resource/quiz/quiz/{id}
Purpose	Removes a quiz with a specific id.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	The result of the delete process (success or fail).
Request	**Parameter:** **Value:**
	{id} — The id of the quiz being deleted (this id is located in the results of the first quiz call shown).

3.2. Question Rest Calls

GET	/resources/question/question/
Purpose	Get all questions from all quizzes.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	A list of all available questions.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/question/question/

GET	/resources/question/question/{id}
Purpose	Get a question with a specific id.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	A single matching question
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/question/question/1
Request	**Parameter:** **Value:**
	{id} — A question id (a selection of question ids can be found from the above question call).

GET	/resources/question/quiz/{id}
Purpose	Get all the questions with a specific quiz id.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	A list of questions for a single matching quiz.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/question/quiz/1
Request	**Parameter:** **Value:**

	{id}	A quiz id (a selection of quiz ids can are present in the first quiz call as well as the first question call).

POST	/resource/question/question/
Purpose	Add a new question to a specific quiz.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	The single result containing the details of the added question.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/question/question/
Request	**Parameter:** **Value:** quizId — The id of the quiz the new question is being added to (a selection of quiz ids are present in the first quiz call as well as the first question call). questionName — The contents of the question being posed. questionAnswer — The answer for the new question. questionImage — The image name (followed by its extension) for the new question.

PUT	/resource/question/question/{id}
Purpose	Update an existing question.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	The single result containing the new details for updated question.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/question/question/1
Request	**Parameter:** **Value:** {id} questionId — The id of question being updated (this id can be found in the first question call shown). questionName — The new title the question being posed. questionAnswer — The new answer for the existing question. questionImage — The new image name (followed by its extension) for the existing question.

DELETE	/resource/question/question/{id}
Purpose	Removes a question with a specific id.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	The result of the delete process (success or fail).
Request	**Parameter:** **Value:** {id} — The id of the question being deleted (this id is located in the results of the first question call).

3.3. Option Rest Calls

GET	/resources/option/option/
Purpose	Get all options available.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	A list of all available options.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/option/option/

GET	/resources/option/option/{id}
Purpose	Get an option with a specific id.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	A single matching option.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/option/option/1
Request	**Parameter:** **Value:** {id} An option id (a selection of ids can be found in the above option call).

GET	/resources/option/name/{query}
Purpose	Get the option with the specific name.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	A single matching option.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/option/option/13
Request	**Parameter:** **Value:** {query} An option name (this can be found in results of the first option call above).

GET	/resources/option/question/{id}
Purpose	Get the options with the specific question id.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	A list of matching options for a question.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/option/question/2
Request	**Parameter:** **Value:** {id} An question id for the chosen options (a selection of question ids can be found in the above questions calls, in addition to the option calls).

POST	/resource/option/option/
Purpose	Add a new option.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	The single result containing the details of the added option.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/option/option/
Request	**Parameter:** / **Value:** questionID — The id of the question the new option is being added to (a selection of question ids are present in the first option call as well as question calls). questionName — The option name.

PUT	/resource/option/option/{id}
Purpose	Update an existing option.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	The single result containing the new details for updated option.
Example URL	https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest/resource/option/option/1
Request	**Parameter:** / **Value:** {id} optionID — The id of option being updated (this id can be found in the first option call) questionName — The new contents of the option being updated

DELETE	/resource/option/option/{id}
Purpose	Removes an option with a specific id.
Response	On success, the HTTP status code in the response header is 200 OK.
Returns	The result of the delete process (success or fail).
Request	**Parameter:** / **Value:** {id} — The id of the option being removed (a selection of ids can be found in the first option call along with a specific id for a question being found in the second option call).

Appendices

Appendix A: Admin Controller (Admin.php)

```
<?php

class Admin extends CI_Controller {

	function __construct() {
		parent::__construct();
		$this->load->model('quiz_model');
		$this->load->library('session');
		$this->load->helper('url');

		// Prevent caching of page, to stop page from being accessed to via the back
		// button after logout (Making sure a web page is not cached, 2016).
		header("Cache-Control: no-cache, no-store, must-revalidate");
		header("Pragma: no-cache");
		header("Expires: 0");
	}

	/* ============================================================================
	   This index function will load automatically upon the controller being called, it will check if the
	    session is set (indicating a user is logged in). If the session is not empty the Admin View will be
	    loaded, but if it is empty the user will be redirected to the Quiz Controller which will  load the
	    home page.
	   ============================================================================*/

	public function index() {
		// Store session data in $session["username"].
		$session["username"] = $this->session->userdata('username');

		// If it is empty the user is not logged in, as such redirect to homeview,
		// else load adminview.
		(!empty($session["username"]))
		? $this->load->view("adminview", $session)
		: redirect('/Quiz/quiz');
	}

}
```

Appendix B: Authenticate Controller (Authenticate.php)

```
<?php

class Authenticate extends CI_Controller {

	function __construct() {
		parent::__construct();
		$this->load->model('quiz_model');
		$this->load->library('session');
		$this->load->helper('url');
	}

	/* ==========================================================================
	   The login function will retrieve the posted superglobal variables (via AJAX) and store them in
	   $username and $password. These variables will then be sent to the "isUser" method within
	   the model which will then return a value based on if the credentials exist within the database.
	   ==========================================================================*/

	public function login() {

		// Retrieve posted data and pass is to the isUser function which returns the array of results
		// if the username and password combination exist within the database and is stored in $res.
		$username = $this->input->post("username");
		$password = $this->input->post("password");
		$res = $this->quiz_model->isUser($username, $password);

		// If the contents of $res are not empty it means the user exists as such the session data
		// containing the users username is created signifying the user is logged in.
		if($res) $this->session->set_userdata(array("username" => $username));

		// Echo the JSON of the return of $res (true or false)
		// echo json_encode($res);
		echo json_encode($res);

	}

	/* ==========================================================================
	   The logout function will destroy the session variables (used to indicate) whether a user is
	   logged in) and redirect the user to the quizzes home page.
	   ==========================================================================*/

	public function logout() {
		// Destroy the session and its variables
		$this->session->sess_destroy();
		// Redirect user to homeview/
		redirect('/Quiz/quiz');
	}

}
```

Appendix C: Quiz Controller (Quiz.php)

```php
<?php

class Quiz extends CI_Controller {

        function __construct() {
                parent::__construct();
                $this->load->model('quiz_model');
        }

        /* ======================================================================
           The quiz function will load the first view when the controller is called and in our case it
           loads the home view and displays all available quizzes.
          ======================================================================*/

        function quiz() {
                // Get available quizzes from the model and store the returned array in the
                // array $quizindex which have 'quizzes' as their key.
                $quizindex['quizzes'] = $this->quiz_model->getQuiz();
                // Send the quiz array $quizindex to the view and display the home view
                $this->load->view('homeview', $quizindex);
        }

        /* ======================================================================
           The questions function will display all the questions and their options that are linked to
           the quiz selected by the user (achieved by passing the quizId appended to the url).
          ======================================================================*/

        function questions() {
                // Get id appended to url with key 'id' using GET, then store within the
                // variable $quizID.
                $quizID =  $this->input->get('id', TRUE);
                // Send quizID as a parameter and get an array of questions which are then
                // stored within $newquestion;
                $newquestion['questions'] = $this->quiz_model->getQuestion($quizID);
                // Send questions to view and display Question View as an associative array
                // with a key of "questions" and then display questionview.
                $this->load->view('questionview',$newquestion);
        }

        /* =========================================================================
           The results array retrieves the posted array and if it exists it sends the array containing
           the selected options and the question id to the model which then calculates the score,
           total and message and returns the array containing these values. This is then stored in
           the $res which is then passed to the view to be displayed.

           Update: results has been modified to not only calculate and return score, total and message
           to the 'scoreview', but also return the average score based on all the previous attempts.
           $res and $average are both passed to the 'scoreview' by being stored in the array $data.
           =========================================================================*/
```

```
    function results() {
                    // Initialise $res (results) array & get posted array and store in $post.
                    $res  = array();
                    $post = $this->input->post();
                    // If post array is set get results by passing the posted array as a
                    // parameter to the models function 'isCorrectAnswer' which returns an
                    // array cont score, message and total which are then stored in $res.
                    if(!empty($post)) $res = $this->quiz_model->isCorrectAnswer($post);

                    // Pass the returned $res to the 'getAverage' method which calculates
                    // and returns the average of all the scores for the quiz being taken.
                    $average = $this->quiz_model->getAverage($res);

                    // Store both the $res and $average in a $data array with the appropriate
                    // keys so they can both be sent to the score view.
                    $data = array();
                    $data["score"] = $res;
                    $data["average"] = $average;

                    // Load the score view to display results and send $data.
                    $this->load->view('scoreview', $data);
    }

}
```

Appendix D: Rest Controller (Rest.php)

```
<?php

class Rest extends CI_Controller {

        function __construct() {
                parent::__construct();
                $this->load->model('rest_model');
        }

 /* ===============================================================================
    _remap is called upon the controller load and is used to pass the URL after the controller name as an
    argument so it can be manipulated. This function handles the request types of: GET, POST, PUT and
   DELETE and loads the appropriate functions.
  ===============================================================================*/

        public function _remap() {

                    // The method being requested is identified and the correct function
                    // is called as a result, this is achieved via IF statements.
                    $method = $this->input->server('REQUEST_METHOD');
                    if($method == "GET")  $this->get();
                    if($method == "POST")  $this->post();
                    if($method == "PUT")  $this->put();
                    if($method == "DELETE")  $this->delete();

        }

 /* ===============================================================================
    The 'get' function will be called when the request method is 'GET', this function converts the url into
    an associative array which is then used to retrieve results from the database, these results are then
    returned in JSON format.
  ===============================================================================*/

        public function get() {

                // I convert the URL to an associative array with every two values being a key and value.
                $args = $this->uri->uri_to_assoc(2);

                // The resource value is retrieved and stored within the $type variable so it is known what
                // the user is trying to access (questions, Quiz or option). This is then remove from the
                // array so the same array can be can be used at a later point.
                $type = $args["resource"];
                unset($args["resource"]);

                 // $args and $type into the 'get' function which returns an assosiative
                // array which is then stored in $res and is converted to json and echoed.
                $res = $this->rest_model->get($type, $args);
                echo json_encode($res);
        }

/* ============================================================================
  The 'post' function will be called when the request method is 'POST', this function will determine
  where the data is being posted to through the resource specified in the url and will then pass the data
  to the insert method within the model which will insert the query and return the data that was
  inserted which will then be returned in JSON format.
  ============================================================================*/

        public function post() {

                // The variables used within this function are initialised here, $notJSON stores the form
                // post, whilst $args stores an associative array derived from the URL and $type includes
                // the resource being posted to (question, option or quiz).
                $notJSON = $this->input->post();
                $args = $this->uri->uri_to_assoc(2);
                $type = $args["resource"];
                $post = "";

                // isJSON is used to determine whether the post request is from a form post. If this amounts
                // to false it means the data is not from a form and instead from backbone in JSON format as
                // such an alternate method of retrieving the data can be used and the data can be decoded
                // from its JSON format.

                if($notJSON) {
                  $args = $notJSON;
                } else {
                  // 'php://input' is used to retrieve all the raw data passed via post.
                  // $post = file_get_contents("php://input");
                  $post = file_get_contents("php://input");
                  $args = json_decode($post);
                }

                // $args and $type into the 'insert' function which inserts a row into the
                // containing the data in $args into table specified in $type
                   $res = $this->rest_model->insert($type, $args);

                // The inserted data is returned back in JSON format in case the data needs to be
                // manipulated
                 echo json_encode($res);

        }

/* ============================================================================
   'PUT' once again the url will be used to determine where the data is being PUT to and the data to be
   Inserted is retrieved from the raw post data and is passed to the update function which will update the
   records and return the data that has been updating which will be returned back in JSON format.
  ============================================================================*/

        public function put() {
                // Variables are once again initialised the url converted into an associative array
                // array and the resource being access is stored within $type.
                $args = $this->uri->uri_to_assoc(2);
                $type = $args["resource"];

                // put stores the data being passed via post and this is decoded and stored in
                // args which is passed to $res which update the record.
                $put  = file_get_contents("php://input");
                $args = json_decode($put, true);
                $res = $this->rest_model->update($type, $args);

                // Success is echoed in JSON format if no errors are returned.
                echo json_encode($res);

        }

 /* ===================================================================================
   The delete function is called when the request method is 'DELETE', this will use the id retrieved from the
   post data and the type retrieved from the url which are passed to the 'delete' function to remove the
   specific row from the database. The delete function will return the delete row, which will be returned via
   JSON.
   ===================================================================================*/

        public function delete() {

                // The resource in which a row is being deleted is defined by converting the
                // URL to an associative array, the resource is then stored within $type and
                // this key and value are removed from the $args array.
                $args = $this->uri->uri_to_assoc(2);
                $type = $args["resource"];
                unset($args["resource"]);

                // $type and $args are passed to the delete function with $type containing the
                // table to delete from and $args contains id of the record being deleted.
                $res = $this->rest_model->delete($type, $args);

                // Echo the returned value (either "success or failed") based on the execution
                // of the delete statement, this can be used to act accordingly by the application.
                echo json_encode($res);

        }

}

?>
```

Appendix E: Quiz Model (quiz_model.php)

```
<?php

class Quiz_model extends CI_Model {

        private $quizzes   = [];
        private $questions = [];

        function __construct() {
                parent::__construct();
                $this->load->database();
        }

        /* ==========================================================================
           This function gets all the available quizzes and stores their details, such as: name, id and
           description in an array and returns them.
          ============================================================================*/

        function getQuiz() {

                // Get all quizzes from the database and store result in $res.
                $this->db->from('quiz');
                $res = $this->db->get();

                if ($res->num_rows() > 0) {
                        foreach ($res->result_array() as $row) {
                                // For every quiz store an array of details within the already
                                // existing quizzes ($quizzes) array (multi-dimensional array).
                                $this->quizzes[] = array(
                                        "id"   => $row['quizId'],
                                        "name" => $row['quizName'],
                                        "desc" => $row['quizDescrip'],
                                        "image" => $row['quizImage']
                                );
                        }
                }
                // Return the quizzes multi-dimensional array.
                return $this->quizzes;
        }

        /* ==========================================================================
           This function get all the questions (identified using the passed parameter $id) and stores each of
           the questions in the questions array, each question contain an options containing all possible
           multiple-choice options array for the question.
          ============================================================================*/

        function getQuestion($id) {

                        // Get all questions matching the passed parameter $id (quizId)
                        // which is a foreign key to get all matching questions to quiz.
                        $this->db->from('question');
                        $this->db->where('quizId', $id);
                        $query = $this->db->get();

                        if ($query->num_rows() > 0) {
                                foreach ($query->result_array() as $row) {

                                        // Create an array to store questions options.
                                        $optionArray = array();

                                        // For every question get associated options using the questionId
                                        // as foreign key $row['questionId'].
                                        $this->db->from('option');
                                        $this->db->where('questionID', $row['questionId']);
                                        $optionquery = $this->db->get();

                                        // Every matching option and the name into the previously created
                                        // options array ($optionsArray);
                                        foreach ($optionquery->result_array() as $row2) {
                                                $optionArray[] = $row2['questionName'];
                                        }
                                        // Shuffle options array to randomize options when displayed.
                                        shuffle($optionArray);

                                        // Store all questions details in an associative array as well as
                                        // another array containing all the questions options.
                                        $questions[] = array(
                                                "id" => $row['questionId'],
                                                "image" => $row["questionImage"],
                                                "name" => $row['questionName'],
                                                "option" => $optionArray
                                        );
                                }
                        }

                        // Shuffle questions array to randomize questions before returning.
                        shuffle($questions);
                        return $questions;
        }

        /* ===========================================================================
           This function Calculates the score out of the total using an array of answers passed as a
           parameter ($answers) by the controller and then  returns an array with a total, score and message
           based on users score.
          ===========================================================================*/

        function isCorrectAnswer($answers) {

                        $score = $total = 0;
                        $quizId = 0;
                        // Loop through array of user answers separating the key from value
```

```
                    // ('questionId' and 'questionName' respectively)
                    foreach ($answers as $key => $value) {

                            // Use foreign key questionId ($key) to get the correct from
                            // question table and store result in $answer.
                            $this->db->from('question');
                            $this->db->where('questionId', $key);
                            $answer = $this->db->get()->row();

                            $quizId = $answer->quizId;
                            // If users answer ($value) = database answer ($answer) return true,
                            // else return false and store boolean in $correct.
                            $correct = ($answer->questionAnswer == $value) ? true : false;

                            // If true increment $score by 1 and increment $total regardless.
                            if($correct == true) $score++;
                            $total++;
                    }

                    // If score is more than half of total store "Well Done!" in message, else
                    // store "Better Luck Next Time"!
                    $message = ($score > ($total/2)) ? "Well Done!" : "Better Luck Next Time!";

                    // Return an associative array which stores the calculated score, total
                    // and generated message.
                    return array("quizId" => $quizId, "score" => $score, "total" => $total, "message" =>
$message);
        }

        /* ==========================================================================
           This function gets all score and not only adds it into the score table with the id for the appropriate
           quiz, but also retrieves, calculates and returns an average of all the scores for the particular quiz
           submitted.
          ==========================================================================*/

        function getAverage($score) {

                $sumScore = 0;
                // Remove message key and value from array so the same array can be used
                // to query and insert into the score table
                unset($score["message"]);

                // Insert the values quizId, score and total stored in array into the
                // quiz table.
                $res = $this->db->insert("score", $score);

                //Retrieve all rows (user scores) from the score table.
                $this->db->where("quizId", $score["quizId"]);
                $result = $this->db->get("score");

                // Loop through all retrieved results calculate the sum of all scores
```

```
		// for that particular quiz.
		foreach ($result->result_array() as $row) {
			$sumScore += $row["score"];
		}

		// Return the average by dividing the sum by number of rows and rounding
		// and rounding the value up.
		return round($sumScore/$result->num_rows(),0);

	}


	/* ===============================================================================
	   This function stores the passed username and password in an array which not only stores the
	   password but encrypts the password and queries the "users" table using the array and return true
	   or false depending on if a user with the credentials exists. (Code used as a base was found in:
	   WebLecture12.pdf (Courtenage, 2015))
	  ===============================================================================*/

	function isUser($username, $password) {

		// Stored username and password in $credentials.
		$credentials = array("username" => $username, "password" => sha1($password));
		// Use stored array in where clause to query "users" table
		$this->db->where($credentials);
		$result = $this->db->get('users');
		// Return true if a user exists with the credentials else, return false.
		return ($result->num_rows === 1) ?  true : false;

	}

}
```

Appendix F: Rest Model (rest_model.php)

```
<?php

class Rest_model extends CI_Model {

        function __construct() {
                parent::__construct();
                $this->load->database();
        }

 /* ================================================================================
   This function handles returning the get request data, it takes the parameters $type which specifies the
   table name and $args which are the parameters for the search. The keys of $type are altered to their
   values in the database and stored within $query so the query can be run correctly and the results array
   of this query is returned.
   ================================================================================*/

        function get($type, $args) {

                $query = array();

                // Here I map the proper url names to their database counter parts, this was
                // done to remove the need to rename individual columns within the database,
                // which would result in certain aspects ceasing to function.
                foreach($args as $key => $row) {
                        if($key == "question") $key = "questionId";
                        if($key == "quiz") $key = "quizId";
                        if($key == "option") $key = "optionID";
                        if($key == "image" && $type == "question") $key = "questionImage";
                        if($key == "image" && $type == "quiz")     $key = "quizImage";
                        if($key == "title"  && $type == "question") {
                              $key = "questionName";
                              $row = urldecode($row) + "?";
                        }
                        if($key == "title"  && $type == "quiz") {
                               $key = "quizName";
                               $row = urldecode($row);
                        }

                        if($key == "name"  && $type == "option") {
                               $key = "questionName";
                               $row = urldecode($row);
                        }
                        if($row != false) $query[$key]  = $row;
                }

                // A where query is run using $querys key and values as parameters
                // for the search and $type is used to specify the table for the search.
                $this->db->where($query);
                $result = $this->db->get($type);
```

```
            // The results array is stored within $return and if only a single row is
            // returned only the first index is returned (thus removing the '[]').
            $return = $result->result_array();
            return ($result->num_rows() == 1) ? $return[0] : $return;

    }

/* ================================================================================
   This function handles inserting data from the POST request. $type and $args are passed and $type
   identifies the table, whilst $args contains the data to be inserted, once the insert query is run and is
   successful the data inserted is returned by this function, if it is unsuccessful "fail" is returned.
   ================================================================================*/

    function insert($type, $args) {

            // An insert query is run with $type identifying the data and $args containing the data to be
            //inserted.
            $res = $this->db->insert($type, $args);

            // If the query if a success (affected_rows() is more than 0) then the data
            // that has been inserted will be returned else "faile"d shall be returned.
            return ($this->db->affected_rows() > 0) ? $args : "failed";

    }

/* ================================================================================
   This function is called on delete and $type and $args are passed as parameters, $type once again
   identifies the table, whilst $args contains the primary key of the row to be deleted, these are both passed
   to the database delete function and once successful the deleted rows details are returned.
   ================================================================================*/

    function delete($type, $args) {

            // Rather than altering the database key names I have decided to alter them
            // via if statements mapping the ones in the api to their original values.
            // These keys are altered and stored into an array named $query.
            $query = array();
            foreach($args as $key => $row) {
                    if($key == "question") $key = "questionId";
                    if($key == "quiz")     $key = "quizId";
                    if($key == "option")    $key = "optionID";
                    if($row != false)      $query[$key]  = $row;
            }

            // The delete database function is gone the id stored in the $query array
            // identifies the record to be deleted and $type identifies the table it is
            // stored within.
            $this->db->delete($type, $query);

            // "Success" or "Failed" are returned depending if the delete query executed
            // successfully this is decided if 'affected_rows()' is more than 0;
            return ($this->db->affected_rows() > 0) ? "success" : "failed";

        }


 /* ==================================================================================
  Lastly the update function handles updating an existing row with new data $args contains this data. The
  $id is derived from $args and this along with type are used in the where query to find the row at hand
  and once found $type and $args are used to identify the table and insert data respectively through the
  use the update database function.
  ===================================================================================*/

        function update($type, $args) {

            // $type has been used to derive the primary key field of the table at hand, thus
            // allowing data to be updated correctly e.g. the quiz tables primary key would be
            // quizId, whilst the option tables would be optionID.
            $id = ( $type == "quiz")     ? 'quizId'
                : (($type == "question") ? 'questionId'
                : (($type == "option")   ? 'optionID' : ''));

            // $id is used to identify the primary key name and is also used to access the id
            // number stored within the $args array which is used to query the database.
            $this->db->where($id, $args[$id]);

            // Lastly, the update query is run with $type holding the tables name in which the
            // record lies and $args containing the new information to be added.
            $this->db->update($type, $args);

            // If the update query was correct run (i.e. the affected_rows() is more than 0)
            // the inserted data will be returned else, "failed" will be returned.
            return ($this->db->affected_rows() > 0) ? $args : "failed";

        }

}

?>
```

Appendix G: Home View (homeview.php)

```
<!DOCTYPE html>
<HTML lang="en">
<HEAD>
        <meta charset="UTF-8">
        <title>QuizTime</title>
        <meta name="viewport" content="width=device-width, initial-scale=1, minimal-ui">
        <link href="https://fonts.googleapis.com/css?family=Open+Sans:400,600,700" rel="stylesheet" type="text/css">
        <link rel="stylesheet" type="text/css" href="<?php echo base_url() ?>/assets/css/style.css"/>
</HEAD>

<BODY>

        <!-- Login Modal Window -->
        <div class="md-modal">
                <div class="md-content">
                        <header><h1>Login</h1></header>
                        <form>
    <div class="input-contain">
     <input id="username" type="text">
     <label>Username</label>
    </div>
    <div class="input-contain">
     <input id="password" type="password">
     <label>Password</label>
    </div>
    <button id="loginBtn">Sign In</button>
   </form>
                        <div class="md-error"></div>
                        <button id="md-close">✕</div>
  </div>
 </div>
```

```
<!-- Display header with centered logo. -->
<div class="nav-lockup">
        <div class="center">
                <a href="quiz" class="logo"></a>
                <input id="openLogin" type="submit" value="login">
        </div>
</div>

<!-- Display a header with a welcome statement and brief instructions. -->
<h1>Are you itching to take a quiz? Choose from the selection below and get started!</h1>

<!-- A grid containing all the available quizzes. -->
<div class="quiz-grid">

        <!-- Open a loop to loop through passed array named $quizzes and each row is named $quiz. -->
        <?php foreach ($quizzes as $quiz) { ?>

        <div class="grid">
                <div class="grid-img">
                        <!-- Display image by stating path and adding the file name stored in the database by echoing
                        the associative array key "image" after the path to the folder which holds the image name. -->
                        <img src="<?php echo base_url() ?>/assets/img/quiz/<?php echo $quiz['image'] ?>" alt="Quiz Image">
                </div>

                <!-- Display the name of the quiz in a h3 tag by using the associative array key 'name' which
                stores the name of the particular quiz, the same applies to the description which has key of
                'desc' and is stored within a p tag. -->
                <h3><?php echo $quiz['name'] ?></h3>
                <p><?php echo $quiz['desc'] ?></p>

                <!-- Call the questions function which displays the quizzes question and append the id to the
                url using get with a key of id by echo the quiz id at the end of the url 'questions?id=' -->
                <a href="questions?id=<?php echo $quiz['id'] ?>" class="cta-grid">Take Quiz</a>
        </div>
        <!-- Close loop. -->
        <?php } ?>
    </div>

    <script src="https://code.jquery.com/jquery-1.11.3.min.js"></script>

    <script>

        /* jQuery code for login modal window. */
        $("#openLogin").on("click", function(e) {
            e.preventDefault();
            $(".md-modal").addClass("show");
            $("body").addClass("md-open");
            $(".md-error").html("")

        });

        $("#md-close").on("click", function(e) {
            $(".md-modal").removeClass("show");
            $("body").removeClass("md-open");
        });

        $("form input").on("blur", function() {
            if($(this).val() != "") {
                $(this).addClass("is-fill")
            } else {
                $(this).removeClass("is-fill")

            }
        });

        $("#loginBtn").on("click", function(e) {
```

```
				// The username and password have been retrieved from the form using .val() and store them in the variables "user" and "pass".
				e.preventDefault();
				var user = $("#username").val();
				var pass = $("#password").val();

				// The stored username and password have been stored in a javascript object named "data" so they can easily be passed using ajax.
				var data = {
					username: user,
					password: pass,
				};


				$.ajax({
					// Send form data to the login function in the authenticate controller.
					url: "<?php echo base_url() ?>/index.php/Authenticate/login",
					method: 'POST', // Send data using POST request.
					data: data, // Pass username and password stored in data.
					dataType: 'json', // Expecting JSON to be returned
					success: function(data) {

						// If the results of data contains true (user is allowed logged in), then the user will be redirected to the
						// Admin panel, however if it false an error message will be added to the login popup/modal.
						if(data) {
							window.location.href="<?php echo base_url() ?>index.php/Admin"
						} else {
							$(".md-error").html("Your username or password are incorrect.")
						}
					}
				});
			});

		</script>
</body>
</html>
```

Appendix H: Question View (questionview.php)

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="UTF-8">
		<title>QuizTime</title>
		<meta name="viewport" content="width=device-width, initial-scale=1, minimal-ui">
		<link href="https://fonts.googleapis.com/css?family=Open+Sans:400,600,700" rel="stylesheet" type="text/css">
		<link rel="stylesheet" type="text/css" href="/CodeIgniter/assets/css/style.css"/>
	</head>

	<body>
		<!-- Display header with centered logo. -->
		<div class="nav-lockup">
			<div class="center">
				<a href="quiz" class="logo"></a>
			</div>
		</div>

		<div class="container">
			<!-- Create a form that on submit sends data to the results function in the controller by using the
			POST method. -->
			<form action="results" method="post">

				<!-- Loop through $questions array and each row (question) will be accessed using $question. -->
				<?php foreach ($questions as $question) { ?>
				<div class="question-lockup">
					<div>
						<!-- Display the question name using it's key ('name') in the $question associative array. -->
						<h3><?php echo $question['name'] ?></h3>

						<!-- Loop through the options array within the $question array using it's key 'option' with each
```

```
                                        individual option being $option. -->
                                        <?php foreach ($question['option'] as $option) { ?>

                                        <!-- for each option create a radio button using the question key 'id' as the input name and
                                        so each questions options are linked together. Also, the value will be retrieved using $option which contains
                                        the name of the question and these will be echoed in the value and within the displayed in the label. -->
                                        <input type="radio" id="radio<?php echo $option ?>" name="<?php echo $question['id']?>" value="<?php echo
$option ?>" required>

                                        <label for="radio<?php echo $option ?>"><?php echo $option ?></label>
                                        <!-- Close option loop. -->
                                        <?php } ?>
                                </div>

                                <!-- Set image retrieved from database as  a background image by using the image name stored within the $qestion array
                                with the key 'image' by concatenating it with the folder path from the root directory. -->
                                <div class="img-container" style="background-image: url('<?php echo base_url() ?>//assets/img/questions/<?php echo
$question["image"] ?>')"></div>

                        </div>
                        <!-- Close question loop -->
                        <?php } ?>

                        <!-- On submission the answers shall be passed in an array to the result function using POST method with the questionId
                        becoming the key to the selected answer, these can then be split up and used to query the database. -->
                        <input class="btn" type="submit" value="Submit">

                </form>
        </div>
    </body>
</html>
```

Appendix I: Score View (scoreview.php)

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="UTF-8">
		<title>QuizTime</title>
		<meta name="viewport" content="width=device-width, initial-scale=1, minimal-ui">
		<link href="https://fonts.googleapis.com/css?family=Open+Sans:400,600,700" rel="stylesheet" type="text/css">
		<link rel="stylesheet" type="text/css" href="/CodeIgniter/assets/css/style.css"/>
	</head>
	<body>
		<!-- Display header with centered logo. -->
		<div class="nav-lockup">
			<div class="center"><a href="quiz" class="logo"></a></div>
		</div>

		<!-- Display results in a div named .score-lockup. -->
		<div class="score-lockup">
			<div class="score">
				<!-- echo exploded array keys as variables within the heading tags with the correct count being $score["score"]
				and the total number of questions being $score["total"]. -->
				<h1>You scored: <?php echo $score["score"] ?> / <?php echo $score["total"] ?></h1>
				<!-- Also echo the exploded array keys containing the average ($average) and message
				($score["message"]). -->
				<h1>The average: <?php echo $average ?> / <?php echo $score["total"] ?></h1>
				<h2><?php echo $score["message"] ?> </h2>
				<!-- On click run the quiz function in the controller, returning to the home page with index of quizzes. -->
				<a href="quiz" class="cta-score">Return Home</a>
			</div>
		</div>
	</body>
</html>
```

Appendix J: Admin View (adminview.php)

```
<!DOCTYPE html>
<HTML lang="en">
<HEAD>
	<meta charset="UTF-8">
	<title>QuizTime</title>
	<meta name="viewport" content="width=device-width, initial-scale=1, minimal-ui">
	<link href="https://fonts.googleapis.com/css?family=Open+Sans:400,600,700" rel="stylesheet" type="text/css">
	<link href="<?php echo base_url() ?>/assets/css/new/main.css" rel="stylesheet" type="text/css">
</HEAD>
<BODY>
	<!-- This section conains the header, which contins the logo and a logout button. -->
	<header>
		<div class="hd-inner">
			<div class="hd-nav" ></div>
			<div class="hd-logo">
				<a href="#">quiz<b>time</b></a>
			</div>
			<div class="hd-user">
				<!-- The base url along with the path to the logout function has been added to the href of the logout link, thus on click
				   the code within the logout method will be run destroying the session and logging the admin out. -->
				<a href="<?php echo base_url() ?>/index.php/Authenticate/Logout">Logout</a>
			</div>
		</div>
	</header>

	<main class="page">
		<!-- Container for the main content, the available quizzes, questions and options shall be added to this div using both jQuery and Underscore. -->
		<div class="content"></div>
		<!-- This will be a popup modal this will contain the form used to add, edit or delete a quiz, question or option. -->
		<div class="md-modal">
			<button class="close">
```

```
                                        <svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" enable-background="new 0 0 23.332 
23.333" height="23.333px" id="Capa_1" version="1.1" viewBox="0 0 23.332 23.333" width="23.332px" xml:space="preserve"><path 
d="M16.043,11.667L22.609,5.1c0.963-0.963,0.963-2.539,0-3.502l-0.875-0.875c-0.963-0.964-2.539-0.964-3.502,0L11.666,7.29  L5.099,0.723c-0.962-0.963-2.538-
0.963-3.501,0L0.722,1.598c-0.962,0.963-0.962,2.539,0,3.502l6.566,6.566l-6.566,6.567  c-0.962,0.963-
0.962,2.539,0,3.501l0.876,0.875c0.963,0.963,2.539,0.963,3.501,0l6.567-6.565l6.566,6.565  c0.963,0.963,2.539,0.963,3.502,0l0.875-0.875c0.963-0.963,0.963-
2.539,0-3.501L16.043,11.667z"/></svg>
                              </button>
                              <!-- The form will be appended to this containing div.-->
                              <div class="md-content"></div>
                    </div>
                    <div class="overlay"></div>
          </main>

          <!--
                    Below are the Underscore.js templates, these allow data to be passed from Backbone.js and manipulated. Certain templates are 
                    created depending on what view is called.
          -->

          <!-- QUIZ LIST TEMPLATE START -->
          <script type="text/template" id="quiz-list-template">
                    <button class="new-quiz">Add new quiz</button>
                    <ul class="qu-outer">

                         <!--
                              An underscore foreach loop will be run here, this will use the quizzes that were passed from within the Backbone.js code, each available 
                              quiz will be named "quiz". Within each iteration the quizzes name, description and image are retrieved using the "quiz.get()" which contains 
                              the keys value for thee details ("quizName", "quizDescrip" and "quizImage" respectively), these are then stored in the appropriate tags. 
                              Lastly, the quiz id is retrieved in the using the same method detailed above but is concatenated with "#/Quiz/" to be used with routers in 
                              the Backbone.js code.
                          -->

                              <% _.each(quizzes, function(quiz) { %>
                              <li>
                                        <div class="qu-tile">
                                        <div class="qu-info">
                                                <a href="#" class="edit-quiz"></a>
                                                <input type="hidden" value='<%= quiz.get("quizId") %>'>
                                                <h3><%= quiz.get("quizName") %></h3>
                                                <span>by Munaib Hussain</span>
                                                <p><%= quiz.get("quizDescrip") %></p>
                                                <a href="#/Quiz/<%= quiz.get('quizId') %>" class="viewq">View Questions</a>
                                        </div>
                                        <div class="qu-img" style="background-image: url(<?php echo base_url() ?>/assets/img/quiz/<%= quiz.get('quizImage')
%>)"></div>
                                </div>
                        </li>
                        <% }); %>
                </ul>
        </script>
        <!-- QUIZ LIST TEMPLATE END -->

        <!-- QUIZ FORM TEMPLATE START -->
        <script type="text/template" id="quiz-edit-template">

                <!--
                        A form to be appended to the above modal (.md-content) will be created here, the various input fields required of a form are created anda
number
                        of "Underscore.js" if statements are also utilised, to differentiation between editing an existing quiz and creating a new one. As shown by
the below
                        (in lines such as: "<%= quiz ? quiz.get('quizDescrip') : '' %>") if the passed "quiz" object is not null it means an update is being run as such
                        a quizzes appropriate details are added into the input fields, however if it is null (new quiz is being created) the input fields will be left
                        blank.
                -->

                <!-- If an edit is occuring the title within the modal will be "Update Existing Quiz", else it will be "Add New Quiz" -->
                <h1><%= quiz ? "Update Existing" : "Add New" %> Quiz</h1><hr>
                <form class="quiz-form">
                        <input type="text" placeholder="Enter quiz name here..." name="quizName" value="<%= quiz ? quiz.get('quizName') : '' %>">
                        <input type="text" placeholder="Enter image name here..." name="quizImage" value="<%= quiz ? quiz.get('quizImage') : '' %>">
                        <textarea rows="8" cols="40" placeholder="Enter quiz description..." name="quizDescrip"><%= quiz ? quiz.get('quizDescrip') : ''
%></textarea>
                        <button class="submit" type="submit"><%= quiz ? "Update" : "Create" %></button>
                        <!-- The below code specifies that Undercore will only add the input field (to signify an update) and delete button if an edit is occurring. -->
                        <% if (quiz) { %>
                                <input type="hidden" name="quizId" value="<%= quiz.id %>">
                                <button class="delete" type="button">Delete</button>
                        <% } %>
                </form>
                <span class="error"></span>
        </script>
        <!-- QUIZ FORM TEMPLATE END -->

        <!-- QUESTION LIST TEMPLATE START -->
        <script type="text/template" id="question-list-template">

                <!--
                   Similar to the previous (shown below), the relevant details are retrieved for the questions using their key names along with the ".get()" method,
                   as such a row containing a quizzes name, answer, image name along with buttons is created for each question available through the use of an
                   Underscore loop.
                -->

                <button class="new-question">Add new question</button><!-- This button will open the modal to create a new question. -->
                <table class="question-table">
                        <thead>
                                <tr>
                                        <th>Question</th>
                                        <th>Answer</th>
                                        <th>Image</th>
                                        <th></th><th></th>
                                </tr>
                        </thead>
                        <tbody>
                        <!-- Below is the each loop using the passed collection for questions, each iteration will be named "question". -->
                        <% _.each(questions, function(question) { %>
                            <tr>
                                <!-- The various details are retrieved and stored within table cells as shown below. -->
                                <input type="hidden" value='<%= question.get("questionId") %>'>
                                <td><%= question.get("questionName") %></td>
                                <td><%= question.get("questionAnswer") %></td>
                                <td><%= question.get("questionImage") %></td>
                                <!-- An edit button is creation which will open the above modal to allow this chosen quiz to be edited. -->
                                <td><a href='#' class='edit-question' class="btn btn-primary"></a></td>
                                <!-- The "questionId" has been retrieved and appended to the url "#/Question/" to be used with backbone routers
to retrieve all the options associated
                                        with this question. -->
                                <td><a href='#/Question/<%= question.get("questionId") %>' class="button">Options</a></td>
                            </tr>
                        <% }); %>
                    </tbody>
                </table>
    </script>
    <!-- QUESTION LIST TEMPLATE END -->

    <!-- QUESTION FORM TEMPLATE START -->
    <script type="text/template" id="question-edit-template">

        <!--
            Similar, to the previous template a form with the appropriate input fields to create or edit a question is created. A number of Underscore if
            conditionals are used to decide whether to retrieve and place existing data into the input fields or to leave them blank.
        -->

        <h1><%= question ? "Update Existing" : "Add New" %> Question</h1><hr>
        <form class="question-form">
            <!--
                Below the needed input fields are created and if needed the questions details (name, answer, image and id) are retrieved using
".get()" and placed
```

```
                 into the input fields values.
                -->
                <input type="hidden" name="quizId"  value="<%= id %>">
                <input type="text" name="questionName" value="<%= question ? question.get('questionName') : '' %>" placeholder="Enter question
name....">
                <input type="text" name="questionAnswer" value="<%= question ? question.get('questionAnswer') : '' %>" placeholder="Enter question
answer...">
                <input type="text" name="questionImage" value="<%= question ? question.get('questionImage') : '' %>" placeholder="Enter question
image...">
                <button class="submit" type="submit"><%= question ? "Update" : "Create" %></button><!-- Submit button is named depending on the edit
or create procedure. -->
                <!-- The elements within thisconditional are only shown when question does not amount to null (meaning an edit of an existing question is
occuring). -->
                <% if (question) { %>
                    <input type="hidden" name="questionId" value="<%= question.id %>">
                    <button class="delete" type="button">Delete</button>
                <% } %>
            </form>
            <span class="error"></span>
        </script>
        <!-- QUESTION FORM TEMPLATE END -->


        <!-- OPTION LIST TEMPLATE START -->
        <script type="text/template" id="option-list-template">

            <!--
              Once again the details for each option available for a question are retrieved using ".get()" along with an options key "quizName" within a
              Underscore foreach loop. Within this loop a row is created within a table for each available option, this row contains the options name along
              with an "edit" button.
            -->

            <button class="new-option">Add new option</button>
            <table class="option-table">
```

```
                <thead>
                        <tr>
                                <th>Option</th>
                                <th></th>
                        </tr>
                </thead>
                <tbody>
                <% _.each(options, function(option) { %>
                        <tr>
                                <input type="hidden" value='<%= option.get("optionID") %>'>
                                <td><%= option.get("questionName") %></td>
                                <td><a href="#" class="edit-option"></a></th>
                        </tr>
                <% }); %>
                </tbody>
        </table>
</script>
<!-- OPTION LIST TEMPLATE END -->

<!-- OPTION FORM TEMPLATE START -->
<script type="text/template" id="option-edit-template">

        <!--
                Similar, to the other form templates a form with the appropriate input fields is created. A number of Underscore if conditionals
                are used to decide whether to retrieve and place existing data into the input fields or to leave them blank, this allows admins
                to either edit existing information or add their own new information.
        -->

        <h1><%= option ? "Update Existing" : "Add New" %> Option</h1><hr>
        <form class="option-form">
                <input type="hidden" name="questionID" value="<%= id %>">
                <input type="text" name="questionName" value="<%= option ? option.get('questionName') : '' %>">
                <button class="submit" type="submit"><%= option ? "Update" : "Create" %></button>
                <% if (option) { %>
```

```
                    <input type="hidden" name="optionID" value="<%= option ? option.get('optionID') : '' %>">
                    <button class="delete" type="button">Delete</button>
                <% } %>
            </form>
            <span class="error"></span>
        </script>
        <!-- OPTION FORM TEMPLATE EDIT -->

        <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.8.3/underscore-min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.2.3/backbone-min.js"></script>
        <script src="<?php echo base_url() ?>/assets/js/main.js"></script>

</BODY>
</HTML>
```

Appendix K: Backbone JS Code (main.js)

```
(function() {
  window.App = {
    Models: {},
    Collections: {},
    Views: {},
    Routes: {},
  };

  /*============================================================================
  This function is tasked with saving data and able to be called by all views,
  this function accepts a model, a form (jQuery) and a collection as parameters.
  ============================================================================*/

  function saveData(model, form, collection) {
    // The data within the form is serialized using the function defined below
    // and stored within the variable "formdata". Whilst the model is stored
    // within the variable "newData".
    var formData = (form.serializeObject());
    var newData = model;

    // Backbones save method is used to store the "formData" in a model and
    // save the contents of the model by sending either a POST or PUT request.
    newData.save(formData, {
      success: function() {
        // On success the modal will be hidden by removing the class show
        // and the collection will be updated with new data using ".fetch()".
        $(".md-modal").removeClass("show");
        collection.fetch();
      }
    });
  };

  /*============================================================================
  This function is tasked with the deletion of model data and accepts two
  parameters, these include a model and a collection.
  ============================================================================*/

  function deleteData(model, collection) {
    // The "destroy" method is executed on the model to send a delete
    // request to the backend with the models data as the payload, thus
    // resulting in either a quiz, question or option being remove.
    model.destroy({
      success: function() {
        // The collection passed will be fetched once again to contain
        // upto date information.
        collection.fetch({
          // upon the fetches completed the modal shall be removed.
          success: function() {
            $(".md-modal").removeClass("show");
          }
```

```
            });
        }
    });
};

/*============================================================================
This function uses ".serializeArray" to convert form data into an javascript
object and is used on a form. This code was obtained from: "http://stackover
flow.com/questions/1184624/convert-form-data-to-javascript-object-with-jquery".
(Stackoverflow.com, 2016)
============================================================================*/

$.fn.serializeObject = function() {
    var o = {};
    var a = this.serializeArray();
    $.each(a, function() {
        if (o[this.name] !== undefined) {
            if (!o[this.name].push) {
                o[this.name] = [o[this.name]];
            }
            o[this.name].push(this.value || '');
        } else {
            o[this.name] = this.value || '';
        }
    });
    return o;
};

/*============================================================================
ajax prefiler is used to set the base url for the rest calls, allowing for
shorthand notation to be used within both the models and collections created.
============================================================================*/

$.ajaxPrefilter(function(options, originalOptions, jqXHR) {
    options.url = "https://w1423768.users.ecs.westminster.ac.uk/CW2/CodeIgniter/index.php/Rest" +
options.url;
});

/*============================================================================
                              MODELS/COLLECTIONS
Below both a model and collection have been defined for quizzes, questions and
options. Each model makes use of the validate function which checks the
integrity of the data stored within the default. In addition, the models also
contain an "idAttribute" which is used to map an id contained within the JSON
object to the models id. Lastly, each model makes use of a "url" attribute
which contains the Rest API url associated with the model/collection.
============================================================================*/

// Quiz Model
App.Models.Quiz = Backbone.Model.extend({
    urlRoot: '/resource/quiz/quiz/', // This line defines the Rest API url.
    idAttribute: "quizId", // This line maps the models id to the quizId.
    defaults: { // The default values are defined and set to null.
      quizId: null,
      quizName: null,
      quizDescrip: null,
      quizImage: null
    },

    // The validate method is used to check the integrity of he mmodels data and
    // accepts the models "defaults" as parameters
    validate: function(attrs) {
      // If any of the values are empty an error message will returned/displayed.
      if ((attrs.quizName == "" || attrs.quizDescrip == "" || attrs.quizImage == "")) {
        $(".error").html("You must complete all fields.");
        return 'All fields must be filled out';
      }
    }
  });

  // Quiz Collection
  App.Collections.Quizzes = Backbone.Collection.extend({
    model: App.Models.Quiz, // A model is to be used with the collection is specified.
    url: '/resource/quiz/quiz/' // The collection Rest url is defined in this line.
  });

  // Question Model
  App.Models.Question = Backbone.Model.extend({
    urlRoot: '/resource/question/question/',
    idAttribute: "questionId",
    defaults: {
      questionAnswer: null,
      questionId: null,
      questionImage: null,
      questionName: null,
      quizId: null,
    },

    validate: function(attrs) {
      if ((attrs.questionName == "" || attrs.questionImage == "" || attrs.questionAnswer == "")) {
        $(".error").html("You must complete all fields.");
        return 'All fields must be filled out';
      }
    }
  });

  // Question Collection
  App.Collections.Questions = Backbone.Collection.extend({
    model: App.Models.Question,
    url: '/resource/question/'
  });

  // Option Model
  App.Models.Option = Backbone.Model.extend({
    urlRoot: '/resource/option/option/',
    idAttribute: "optionID",
    defaults: {
        optionID: null,
        questionID: null,
        questionName: null
    },

    validate: function(attrs) {
        if ((attrs.questionName == "")) {
            $(".error").html("You must complete all fields.");
            return 'All fields must be filled out';
        }
    }
});

// Option Collection
App.Collections.Options = Backbone.Collection.extend({
    model: App.Models.Option,
    url: '/resource/option/'
});

/*=========================================================================
                                    VIEWS
Below a number of views are created for quizzes, questions and options all of
which contain very similar code, each of these views are passed a collection
and an id from the url. The Question View is passed the quiz id to retrieve
all the relating questions, whilst the Options View is passed the question id
to get all the options for a question.
=========================================================================*/

// Quizzes (Index) View
// This class is tasked with displaying the various quizzes available along with
// with managing the creation, deletion and alterations of these quizzes.

App.Views.Quizzes = Backbone.View.extend({
    el: ".page",

    events: {
        // The various events for the quiz view are defined below.
        "click .new-quiz": "newQuiz",
        "click .edit-quiz": "editQuiz",
        "click  .delete": "delete",
        "click .close": "close",
        "submit .quiz-form": "create",
    },

    initialize: function() {
        // A listener is added to the passed collection which triggers the render
        // functon upon any changes.
        this.listenTo(this.collection, "sync change", this.render);
        // A fetch is called to sync the collection, this also triggers the render
        // function.
        this.collection.fetch();
    },

    // This functions present in all view renders a view containing all the
    // available quizzes by utilising Underscore templating along with jQuery.
    render: function(data) {
        // An existing underscore template is found using jQuery and stored within
        // the variable "template". Next, the collections resulting models are
        // stored within a javascript object named "vars".
        var template = _.template($('#quiz-list-template').html());
        var vars = {
            quizzes: data.models
        };
        // The below code shows the stored javascript object being passed to the
        // template and being rendered in the ".content" div within the html.
        var html = template(vars);
        this.$el.find(".content").html(html);
    },

    // This function renders and displays the modal used to create a new quiz, this modal contains
    // blank input fields and is also created using Underscore templating.
    newQuiz: function() {
        // The correct template is located using jQuery and is stored within the variable template.
        var template = _.template($('#quiz-edit-template').html());
        // The passed collection is set to null to signify a new quiz is being created and is stored
        // in the javascript object named "vars".
        var vars = {
            quiz: null
        };
        var html = template(vars);
        $(".md-modal .md-content").html(html);
        // The class shown is added to the ".md-modal" to make the modal visible.
        $(".md-modal").addClass("show");
        return false;
    },

    // This function is renders the modal used to edit a quiz, this modal contains inputs containing
    // a quizzes existing data and is created using Underscore templating.
    editQuiz: function(e) {
        // this is stored in a variable that to rescope "this."
        var that = this;
        // The id for the chosen quiz is retrieved using jQuery
        var id = $(e.target).parent().find("input").val();
        // The retrieved id stored in "id" is used to query the quiz model which returns the
        // resulting data upon fetch, this data is then passed to a modal template in which
        // the appropriate input fields will be filled in signifying an edit procedure.
        this.model = new App.Models.Quiz({
            quizId: id
        });
        this.model.fetch({
            success: function(data) {
                var template = _.template($('#quiz-edit-template').html());
                var vars = {
                    quiz: data
                };
                var html = template(vars);
                $(".md-modal .md-content").html(html);
                $(".md-modal").addClass("show");
            }
        });
        return false;
    },

    // This function passed a quiz model, the name of the template and the collection being
    // used to the saveData function which is an anonymous function, this function then saves the
    // data.
    create: function() {
        new saveData(new App.Models.Quiz(), $(".quiz-form"), this.collection); // Add New Quiz.
        return false; // Prevent default form action.
    },

    // This function passed both the views model and colelction to the deleteData function which
    // is tasked with removing the data in the model fro the backend and reloading the collection
    // with the new data.
    delete: function(e) {
        new deleteData(this.model, this.collection) // Delete Quiz.
        return false; // Prevent default for action.
    },

    // This function closes the modal window by removing the class "show".
    close: function(e) {
        $(e.target).parent().removeClass("show");
    }

});

// Question View
// This class is tasked with displaying the various quizzes available along with
// with managing the creation, deletion and alterations of these quizzes.

App.Views.Questions = Backbone.View.extend({
    el: '.page',

    events: {
        // The events for the question view are defined below.
        "click .new-question": "newQuestion",
        "click .edit-question": "editQuestion",
        "click .delete": "delete",
        "click .close": "close",
        "submit .question-form": "create",
    },

    initialize: function(options) {
        // The quiz id passed in the url is passed and stored in the variable optons.
        this.options = options;
        // A listener is added to the question collection which fires the render
        // function upon "sync" or "change".
        this.listenTo(this.collection, "sync change", this.render);
        // The id passed to the url is appended and set as the collections url.
        this.collection.url = '/resource/question/quiz/' + options.id;
        // The collection fetches all the JSON data that that is a result of the
        // url defined above.
        this.collection.fetch();
    },

    // Once again, this function creates an udnerscore template and passed the
    // collections models and the quiz id to the template before appending it to
    // the ".content".
    render: function(data) {
        var template = _.template($('#question-list-template').html());
        var vars = {
            questions: data.models,
            id: this.options.id
        };
        var html = template(vars);
        this.$el.find(".content").html(html);
    },

    // Similar to the function "newQuiz", this function display a modal used to create a new question.
    // This modal contains blank inputs which is specified by no collection being stored in the "vars"
    // variable which is passed to the template. The modal template stored within the variable "template"
    // is passed the javascript object "vars" and added to the modal using jQuery.
    newQuestion: function() {
        var template = _.template($('#question-edit-template').html());
        var vars = {
            question: null,
            id: this.options.id
        };
        var html = template(vars);
        $(".md-modal .md-content").html(html);
        $(".md-modal").addClass("show");
        return false;
    },

    // This function is tasked with editing a question, the id for a particular question is retrieved using
    // jQuery and sotred within id, this id is then used to query the Question Model which fetches the data
    // and sends it to the Underscore template which in turn shown a modal containing inputs filled with
    // the questions information.
    editQuestion: function(e) {
        var that = this;
        var id = $(e.target).parent().parent().find("input").val();
        this.model = new App.Models.Question({
            questionId: id
        });
        this.model.fetch({
            success: function(data) {
              var template = _.template($('#question-edit-template').html());
              var vars = {
                question: data,
                id: that.options.id
              };
              var html = template(vars);
              $(".md-modal .md-content").html(html);
              $(".md-modal").addClass("show");

            }
          });
          return false;
        },

        create: function() {
          new saveData(new App.Models.Question(), $(".question-form"), this.collection); // Add New
Question.
          return false; // Prevent default form action.
        },

        delete: function() {
          new deleteData(this.model, this.collection); // Delete Question
          return false; // Prevent default form action (submitting).
        },

        close: function(e) {
          $(e.target).parent().removeClass("show");
        }

      });

      // Option View
      // This class is tasked with displaying the various options for a particular question,
      // along with managing the creation, deletion and alterations of these options.
      App.Views.Options = Backbone.View.extend({
        el: '.page',

        events: {
          // The various click events for the options view are defined below.
          "click .new-option": "newOption",
          "click .edit-option": "editOption",
          "click .delete": "delete",
          "click .close": "close",
          "submit .option-form": "create",
        },

        initialize: function(options) {
          // The id passed in the url is passed and stored in the variable optons.
          this.options = options;
          // A listener is added to the options collection which fires the render
          // function upon "sync" or "change".
          this.listenTo(this.collection, "sync change", this.render);
          // The id passed to the url is appended and set as the collections url.
          this.collection.url = '/resource/option/question/' + options.id;
          // The collection fetches all the JSON data that that is a result of the
          // url defined above.
          this.collection.fetch();
        },

        // This functions render a iew containing all th available options for a particular
        // question, this is achieved using the question id appended to the collection url
        // and passed to the template (this.options.id).
        render: function(data) {
          // An Underscore template is retrieved using jQuery.
          var template = _.template($('#option-list-template').html());
          // The passed collection and question id are stored within an object named vars.
          var vars = {
            options: data.models,
            id: this.options.id
          };
          // Vars is then passed to the specified underscore template and created using
          // jQuery ".html()".
          var html = template(vars);
          this.$el.find(".content").html(html);
        },

        // This function renders and displays the modal used to create a new option, this modal contains
        // blank input fields and is also created using Underscore templating.
        newOption: function() {
          var template = _.template($('#option-edit-template').html());
          var vars = {
            option: null,
            id: this.options.id
          };
          var html = template(vars);
          $(".md-modal .md-content").html(html);
          $(".md-modal").addClass("show");
          return false;
        },

        // This function retrieves the data for a particular option and create a form using Underscore templating
        // and populates the input fields using data retrieved from the model (achieved using the id retrieved
using
        // jQuery).
        editOption: function(e) {
          var that = this;
          var id = $(e.target).parent().parent().find("input").val();
          this.model = new App.Models.Option({
            optionID: id
          });
          this.model.fetch({
            success: function(data) {
              var template = _.template($('#option-edit-template').html());
                var vars = {
                    option: data,
                    id: that.options.id
                };
                var html = template(vars);
                $(".md-modal .md-content").html(html);
                $(".md-modal").addClass("show");
            }
        });

        return false;
    },

    create: function() {
        new saveData(new App.Models.Option(), $(".option-form"), this.collection); // Create/Update Option
        return false;
    },

    delete: function(e) {
        new deleteData(this.model, this.collection); // Remove Option
        return false;
    },

    close: function(e) {
        $(e.target).parent().removeClass("show"); // Hide Modal
    }

});

/*=============================================================================
                                        ROUTERS
The code below maps the segments of the url after the hash to functions,
these urls can also be used to pass "id" values which are also passed to the
below (as shown through the ":id") present.
=============================================================================*/

// Router
App.Routes.Router = Backbone.Router.extend({
    routes: {
        "": "index", // No additional url being added will call the function named index.
        "Quiz/:id": "viewQuestion", // The specified will call the "viewQuestion" function.
        "Question/:id": "viewOption", // The specified url will call the "viewOptions" function.
    },

    index: function() {
        // Below we instatiate the Quizzes Collection and store it in the variable  named
        // "collection".
        var collection = new App.Collections.Quizzes();
        // A view is also created and stored in the "view" variable the previously created
        // collection is also passed to the Quizzes View.
        var view = new App.Views.Quizzes({
            collection: collection
            });
        },

        viewQuestion: function(id) {
            // All events within page are removed to avoid events firing multiple times when new
            // ones are added.
            $(".page").unbind();
            // A new collection for "Questions" is created and stored in the variable named
            // "collection".
            var collection = new App.Collections.Questions();
            // A new Questions view is instantiated and both the previously made collection and
            // id present in the url are passed as paramaters.
            var view = new App.Views.Questions({
                collection: collection,
                id: id
            });
        },

        viewOption: function(id) {
            // All events within ".page" are destroyed to allow for new events to be created.
            $(".page").unbind();
            // A view and collection are created for the options, once again the created
            // collection and id are passed as a paramaters.
            var collection = new App.Collections.Options();
            var view = new App.Views.Options({
                collection: collection,
                id: id
            });
        },

    });

    // This code below instantiates Backbone Router and starts it.
    var router = new App.Routes.Router();
    Backbone.history.start();

})();
```

References

Courtenage, S. (2015). Logins. 1st ed. [PDF] Available at: https://learning.westminster.ac.uk/bbcswebdav/pid-1532415-dt-content-rid-3989877_1/courses/ECWM604.1.2015/weblecture12.pdf [Accessed 12 Jan. 2016].

Making sure a web page is not cached, a. (2016). Making sure a web page is not cached, across all browsers. [online] Stackoverflow.com. Available at: http://stackoverflow.com/questions/49547/making-sure-a-web-page-is-not-cached-across-all-browsers [Accessed 09 Jan. 2016].

Stackoverflow.com, (2016). Convert form data to JavaScript object with jQuery. [online] Available at: http://stackoverflow.com/questions/1184624/convert-form-data-to-javascript-object-with-jquery [Accessed 12 Jan. 2016].