Classroom Quiz System

Final Report for CS39440 Major Project

Author: Alexander Michael Levi Taylor (amt22@aber.ac.uk)
Supervisor: Mr. Chris Loftus (cwl@aber.ac.uk)

20th April 2017
Version: 1.4 (Draft)

This report was submitted as partial fulfilment of a BSc degree in Computer Science (Inc Integrated Industrial And Professional Training) (G401)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name .....

Date .....

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name .....

Date .....

Acknowledgements

I am grateful to...

I’d like to thank...

Abstract

The focus of this project was to create an application that can be used by lecturers to allow students to answer questions and display these results live in lectures. The project was based on a pre-existing application, called Qwizdom [1] but this project aims to improve upon Qwizdom in several ways.

The system is a web based application built in Laravel, a PHP based framework. The system provides two main functions, the running of questions on their own, and streaming lecture slides with questions embedded within the slides to students. This system improves upon Qwizdom in several ways. Firstly, it runs via a website rather than via a PowerPoint extension, meaning more lecturers can use it. Furthermore, it provides more functionality such as more possible answers and students cannot submit their own answers as they can with Qwizdom.

Development followed an Extreme Programming approach, utilising a number of practices to help development. Stories were created and used as the functional requirements for the system, and stories were worked on individually within weekly iterations. Analysis, design, implementation and testing were done for each story rather than having an upfront design stage or end testing stage.

TODO: mention user testing when thats done

TODO: summary of eval when written?

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Background & Objectives

1.1 Background

1.1.1 Qwizdom

Currently lecturers at Aberystwyth University have the option to use a service called Qwizdom (TODO cite) which allows lecturers to embed quiz questions into their slideshow presentations. During a lecture, students can join a session through an online portal and have the slides and questions streamed through to their devices. Quiz questions can then be answered by the students and their results can be displayed live by the lecturer, who can also save these results for later analysis.

1.1.2 Replacement

A replacement system was wanted to fix some of the problems that Qwizdom has. Primarily that the University only has one session key, which means only one session can be used by all lecturers at any one time, leading to some clashes. With a new system, there would be no limit to the number of sessions that can be run. Other problems with Qwizdom include:

- Maximum of 6 answers (TODO: was it six? ask Chris)
- Students can submit their own answers by changing the HTML on the page
- Creating quizzes is tied into Microsoft PowerPoint which not all lecturers use
- Aging interface on the student end

1.2 Analysis

1.2.1 Two parts

It was decided that the project would be split into two main parts, the first was to create an application that lecturers can run a simple quiz from without any slides. The second part involves

developing an application or extension to the first part that allows the lecturer to create the quiz as part of a set of slides. The slide the lecturer is viewing would then be displayed in the quiz session on the web where students can access it. Once a relevant quiz slide appears, the students will be given the quiz options to select an answer in the same way as the first part. The lecturer can then display the results in the same way, though it would not be via the web application as before. This part of the project behaves in much the same was as Qwizdom. The reason for having two parts is that the first would be mostly built as part of the development for the second part but if built as a standalone part it allows lecturers to create quizzes rather than a set of slides with quizzes within it. Features like the session joining, front-end view for students and the way answers are submitted would be used in the second part, which means only creating quizzes would need to be added for the first part for it to be stand alone. Additionally, the second part had the potential to be much larger in scope than originally anticipated and as such having the first part to extend in a different direction should the second part become unviable would be provide a safety net.

1.2.2 Part 1

A web based approach was chosen for the first part. This application would allow lecturers to log in to an admin panel and from there create and run quizzes for the students. This would also lay the groundwork for the second part, setting up the front end for students, how sessions are run and how answering questions worked.

1.2.2.1 Framework

The PHP language was selected due to two primary reasons. The first is that the developer was familiar with PHP after using it in an industrial job. The second is that a large number of University projects are PHP based which will make integrating this with the University easier in the future. Whilst Ruby or Javascript might able be applicable, the familiarity and ability to extend PHP made it the best choice.

1.2.2.2 Laravel

Laravel is a web framework written in PHP(TODO: cite laravel), and has been chosen for the web server part of this project. There are a number of reasons for choosing Laravel in this project over other available frameworks. The first and most important is familiarity with Laravel as the developer has used it in the past, and the main framework used in their year in industry was based on Laravel.

Laravel is also the most popular PHP framework available(TODO cite this), which means there is an enormous amount of support available in the form of user forums, video guides and its own tag on Stack Overflow.

Laravel 5 also natively supports web sockets, a technology that was being considered for the first part of the project. Other features of Laravel is its conformance to PSR-2, a PHP coding standard, meaning it would be easier to follow good coding standards more easily.

1.2.3 Part 2

The second part required more extensive research to be done when it was reached in development. But the main suggestion was to create an Microsoft PowerPoint extension much like Qwizdom to create slides embedded with questions. Due to the size of such an undertaking, another potential solution was to create the web application in such a way that an extension could be worked on in the future, i.e. set up the system for future extensions. TODO: cite Microsoft extension stuff

1.3 Process

The methodology chosen for this project was an Extreme Programming based approach. One of the core advantages of using an XP based approach is the ability to adapt to change. The second part of this project, implementing a method to stream slides to students, was much more open to change than the first part where the scope was far more understood. If another process such as Feature Driven Development was used, it would work for the first part of the project as all the requirements and features are known but most likely would be a struggle to use in the second part. XP gives the flexibility needed to complete the second part of the project whilst also allowing the first part to be done with ease.

1.3.1 Practices

A number of practices were selected for the project that work in a single developer project:

- Test Driven Development - Writing tests before coding any of the application logic helps to enhance both the design and also mean tests should actually be written rather than left until the end and ”hacked” in.
- Coding Standards - Keeping to strict coding standards helps ensure the code is good quality and easy to extend in the future as this project may well be.
- Small Releases - Small releases help enforce releases bit of working code regularly and results in a better overall project if the sub parts are all working.
- On Site Customer - This practice is somewhat applicable, the developer can act as a customer, and the project supervisor can also somewhat act as a customer, due to them being the originator of the idea and also a lecturer, one of the main users of this application.
- Merciless Refactoring - Refactoring is already encouraged by using TDD, but it can also be used at other points in the project to ensure the code is structured sensibly.
- Planning Game - This can be adapted to be done by one person at the start of the project, the list of stories was written and then iterations and releases planned out.
- Continuous Integration - Some online tools can be used to provide an CI workflow, to run tests continually with the constant small releases.

1.3.2 Stories - functional requirements

Before any development could start, a list of functional requirements was needed. For an XP based project, these would be in the form of ”stories”. The stories have a difficulty ranking associated with each item in relation to the other stories. 1 would the easiest and 10 the hardest. Stories for the first part:

- (6) Admins can create quizzes via the website
- (4) Quizzes contain a variety of questions
- (7) Admins can login to the website and run a session with a quiz
- (8) Multiple different sessions can be run simultaneously
- (3) The Admin specifies which question is being run by clicking next/prev question etc
- (5) Sessions can be joined by users via the website
- (3) Up to 300 users should be able to join and answer questions
- (1) Users answer the question being displayed by the quiz
- (2) The Admin can see what percentage of users connected to the session have answered
- (4) The Admin can show the results of the question in a sensible format e.g. graph
- (2) Admin can then save results as CSV or XML
- (2) The Admin can load from saved file to display again
- (4) Users should not be able to submit their own answers by altering the HTML, as they did with Qwizom

These ones fit the second part, though there are some stories from the first part that are applicable within this part.

- (1) The Admin creates slides in a slideshow editor (Most likely Microsoft Office Power Point)
- (2) The Admin can place question slides within the slides during creation
- (10) The Admin can stream these slides to a session
- (5) Users can join the session and follow the slides as they are used
- (4) The specified question slides will act as questions for the users connected to the session
- (2) Results are handled in the same way as the first web only part
- (3) The Admin can embed HTML content in quiz slides during creation

1.4 Planning Game

A plan was created early in development:

- 10 iterations beginning on Thursdays remaining from when the original plan was made (22/02/2017)
- Leave 2 iterations for Report Writing and emergency bug fixing at the end, from iterations beginning 20/04/2017
- 8 iterations between planning and iteration 8, these to be devoted to majority of coding
- 3 iterations before mid-project demonstration from initial planning

It was decided that it would be beneficial to try and get the majority of coding done within the 8 iterations specified, this would give the final two iterations breathing room to put together the final document more formally and give some bug fixing time and cleaning up time. In terms of releases, the aim was to have sets of features that would be releasable at the end of every iteration, though no planning of which features being released when was made.

Chapter 2

Design

This chapter will give an overview of the design of the final system, however will not go into much detail for individual parts of the system. Those designs are within the next chapter as design took place within each iteration rather than at the start. TODO: could this chapter actually be after the Iteration chapter, might make more sense?

2.1 Overall Architecture

The Laravel framework uses a Model-View-Controller design pattern to build applications. It works in the same way as a normal MVC application. Data is stored and manipulated within the models, controllers handle most of the interactions the user has with the system, and the views are the web pages presented to the users. Within Laravel, there is also a front controller, which is used to route incoming HTTP requests to the appropriate views and controllers [2].

There are two basic sections of the system, the student end, and the lecturer end. The lecturer end is an admin panel that they can login to, to create and manage their quizzes. The student end consists of the portal for connecting to a session, and the session pages themselves that display the slides and questions for users to answer.

Students only have two bits of functionality available to them, being able to join a quiz and then being able to answer questions. Lecturers however can do much more on the site. On the admin panel they can create new quizzes, create and add questions to the quizzes and also add slides to the quizzes. They can also change their user details including their session key. The main operation they can perform is that they can run a quiz, this then takes them to the same view as the students, albeit with a small control panel for changing questions or slides, and displaying the results of the quiz.

2.2 Database Design

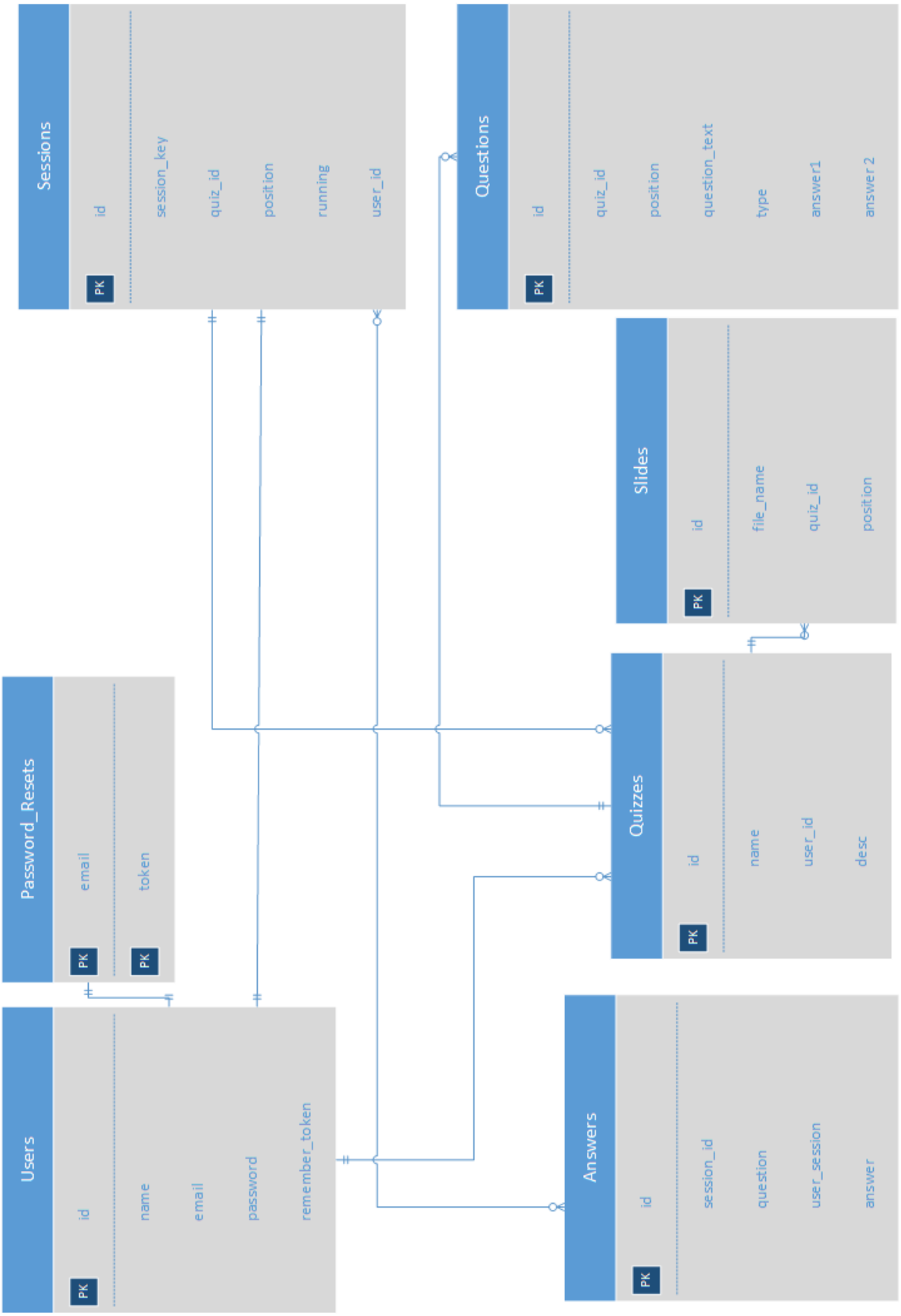


Figure 2.1: Entity relationship digram for final database

There are seven tables within the application. Two of these were generated by running a Laravel command to make the authentication part of the site, the users and password_resets tables. These provide basic user login.

The quizzes table contains information about the quizzes themselves, and are associated with a user. The questions and slides tables both store information about their respective parts of a quiz, with each row within these tables associated with a quiz. Questions store information about the actual question including all the answers, not in the design above are the other eight fields for answers up to answer10. The slides table stores the file name of a slide image that has been converted from a pdf slide. Both of these tables store the positions of their items within a quiz.

The sessions table stores the information about the runnable session, each user has one associated session row. Within this row, the session_key is stored, which is the key that students would use to connect to a session. Additionally it stores information about the session when its running, specifying if it is running, and what position it is at. This position references the positions specified int he questions and slides tables, though there is no actual foreign key relationship between them.

The final table, answers, stores all the responses from users to questions. It stores which question, the answer given and the user that submitted the answer. The user_session is a cookie value rather than a user from the database.

Chapter 3

Iterations

3.1 Iteration 0 16/02 - 22/02

3.1.1 Initial Work

This iteration did not consist of any development, that begins in iteration 1. It consisted mainly of research and some design work. But most importantly it included writing the stories for the project. Some other smaller bits of work involved setting up a diary and a Trello board for documenting all the work.

3.1.1.1 Research

Research was focussed on several areas, and some documents were produced from this research. The first area of research was on the frameworks available, after which Laravel was selected. Hosting options were also explored and a document discussing both of these was produced.

Some additional research was also completed on the Microsoft PowerPoint Add-In for the second part of the project. This was mainly to get a sense of the amount of work involved, and what language/ environments would be needed. This research did not go into much depth as further research will be undertaken when that part of the work is reached.

3.1.1.2 Design

The design work completed was a database design. This will be needed throughout the whole project so makes sense to write now, though it may be subject to significant changes.

3.1.1.3 Stories

A list of stories was produced and written up in a separate document. These stories will be the basis of the project and act as the functional requirements for the evaluation at the end of development.

3.2 Iteration 1 23/02 - 01/03

3.2.1 Story: Admins can create quizzes via the website

3.2.1.1 Analysis - Breakdown of Tasks

This story is quite large and should be broken down into several substories:

- (3) Admins can log into a backend
- (2) They are presented a list of their quizzes
- (5) They can create a new quiz in the backend
- (3) They can edit an existing quiz they own

These have been added to the story list document.

3.2.2 Story: Admins can log into the backend

3.2.2.1 Analysis - Breakdown of Tasks

- Create users table
- Add login page
- Add register page

3.2.2.2 Design

Design was limited due to the automated builder. However it added some view files and a default HomeController for a basic homepage.

3.2.2.3 Implementation

Implementing this was far easier than originally anticipated, Laravel comes with the needed tables out of the box and has a command to run that sets up simple auth for users: *php artisan make:auth*

3.2.2.4 Testing

Because the login and auth is handled by Laravel by default, testing it was deemed to not be a priority. However, three simple tests were written to ensure it never breaks due to future changes. Dusk Tests:

1. Test to ensure the application redirects to /login is the user is not already logged in
2. Test for logging in with a user in the database
3. Test for registering a new user

3.2.3 Story: They are presented a list of their quizzes

3.2.3.1 Analysis - Breakdown of Tasks

- Make the homepage the QuizController rather than the HomeController
- Check and get the user who is logged in
- Display a list of quizzes for that user

3.2.3.2 Design

The HomeController to be removed in this period of work and the QuizController used in its place. (TODO: Add UI mock)

3.2.3.3 Implementation

Quite an easy amount of work, changing the controller was a simple change to the routes and then updating the quiz view file to use the same layout as the original home views. To get the user, a helper function is provided: *auth()->user()* which gets the user object. Obtaining the id from this is simple and then using an Eloquent ORM call it is easy to find all the quizzes owned by that user.

3.2.3.4 Testing

Dusk tests for this story:

1. Test to see if the /home page lists the quizzes as it would on the /quizzes page
2. Test to see if a quiz that belongs to a user is present on the page
3. Test to see if a quiz that belongs to another user is not present on your page whilst your own is

These tests highlighted a problem with the testing framework however. Chrome is used as the Remote Web Driver for running these application tests in. For each test a new migration is made within the test database, thereby wiping the data created within each test. An unintended side effect however is that every new user created starts at id=1 in the users table (a user has to be created for all these tests.) The Chrome driver seems to remember that the user of id=1 logged in, in the previous test and therefore skips the auth step. This means the test order is messed up due to the test trying to log in even though it is already logged in. The solution to this is to create each new user in the next id record, whilst this is somewhat convoluted, it seems to work.

Potential future tests: Use sessions have two users log in and see/ not see the relevant quizzes.

3.2.4 Non-Story Work

3.2.4.1 Database Work

Some initial work that is needed for almost all the stories is having a working database set up to store the users, quizzes, questions etc. Seeing as the amount of work to setup all the tables and their relationships would not take long, it was decided that this could be done all at once at the start of the iteration.

While creating these tables it was possible to create the controllers needed within the application at the same time using: *php artisan make:model *name* -mc*

3.2.4.2 Seed Data

Because of the amount of changes to the database that were being made, the tables were repeatedly wiped and seeding some data was needed. To do this some seeders were generated with *php artisan make: seed *name**. These were created under database/seeds/ and simply required creating new objects of the desired Model and adding the various fields as parameters of the objects. These objects are then saved to the database. These seeders can be run when a migration is called such that the data is replaced as soon as its lost.

3.2.4.3 Layout Changes

The initial *make:auth* command created a default home page with a menu bar and some basic styling. This styling was created with Bootstrap and looks quite nice so the basic styling has been kept. This layout was modified somewhat to add some menu options that persist across pages. This layout is then used by all the backend pages created by extending it.

3.3 Iteration 2 02/03 - 08/03

3.3.1 Story: They can create a new quiz in the backend

3.3.1.1 Analysis - Breakdown of Tasks

- Need quiz/new form to add name
- Add validation
- This then redirects to quiz.show for this new quiz
- This page needs a button that can add a question

3.3.1.2 Design

The design is split into two parts, design for the quiz/new page and a design for the question/new page: (TODO: insert images for these, maybe some form of activity diagram?)

3.3.1.3 Implementation

A create page was added for quizzes, this was linked to from the home page. This was mapped to /quizzes/create address. A simple form was added to this page with the name and description of the quiz. Questions would be added on an individual quiz page. The form sends the form as an HTTP POST request to the /quizzes page which then maps onto the 'Store' action in the Quiz Controller. This action is used for server side validation and saving the data to the database using an Eloquent function. This Eloquent function also returns the id of the new quiz. The user_id is assigned to the quiz by simply getting the current user. Using this id, this Store function redirects to the newly created quiz.

On a quiz page, it displays the name and description and has a button to add any questions to the quiz. This button links to the questions/create page that functions in the same way as the quiz creation, albeit with the relevant fields in the form. The questions are assigned to the quiz by passing the quiz id to the question creation page in a GET variables via the url. This allows the question to be assigned to the quiz during its creation in the database and to be redirected back to the quiz page.

For validation there are two parts, one on the HTML form and the other handles server side. The HTML validation (client side) uses simple HTML 5 validation like the "required" attribute in the form inputs. Further frontend validation could be added utilising JavaScript but the HTML solution was far easier to implement, being the addition of one word and not several lines of code that have to be built and added to the correct pages. The server side complements this as client side validation can be circumnavigated by determined users.

The server side validation in Laravel is well supported and made as easy as possible to implement. The Store function used to save data from the POST request simply calls *$this->validate()* on the request data. Inside this validation function, various rules can be imposed on individual pieces of the request data such as simply requiring that is is filled or that it has to have a minimum length [3].

Something that Laravel enforces is the use of CSRF tokens to prevent cross site request forgery attacks on the site. A token is generated for the project during its creation, and this token must be placed in a hidden field within every form so that it can verify that the authenticated user is the one actually making the requests to the application [4].

3.3.1.4 Testing

Dusk tests:

1. Test to create a new quiz
2. Test the HTML validation for creating a new quiz
3. Test backend validation of quizzes
4. Test a quiz contains questions that have all been added before hand
5. Test to add a question to a quiz
6. Test to make sure the question page contains the relevant information

A problem encountered in this set of tests is the speed of testing. Running these tests takes about a minute. The reason for this is that the tests use DatabaseMigrations rather than DatabaseTransactions, meaning that the database is migrated for every test. Using transactions would reduce the time taken by only doing a migration at the start and then using a transaction for each test. Alternatively having a pre migrated test database on which you run transactions would work too, but this would mean any time you change your database, you would have to remember to migrate the test database.

After attempting to use the transactions it appears as though they are not usable within Dusk. This is because Dusk is running in another process and migrations is the only choice [5].

3.3.2 Story: They can edit an existing quiz they own

3.3.2.1 Analysis - Breakdown of Tasks

- There is an edit button on quizzes that links to quiz/edit
- You can edit the name of the quiz
- You can remove questions
- You can add questions
- You can edit the content of a question
- You can delete a quiz and its associated questions

3.3.2.2 Design

There was not much design for this stage as it only really adds an edit page which should look the same as the create pages in the above story except that the input fields are pre-filled with data that is being editing. Additionally a delete button should be added for quizzes and questions. (TODO: more design here)

3.3.2.3 Implementation

Pre filling the edit page was done by getting the record in the database by the id and just setting the default values of the input fields. The main difference to the creation stage was that the form had to use the HTTP PATCH method rather than POST. This PATCH method is automatically routed to the update function in the controller [6]. Within this function, server side validation is performed and and Eloquent called to update the row in the database. It then redirects back to the quiz page for editing either a question or quiz.

Delete buttons were added to the quizzes and questions, these were not a simple <a >tag that linked to a delete page like the create and edit pages. This had to be a small form that sent an HTTP DELETE request along with an id to the /quizzes or /questions page. This was mapped to the destroy functions in the respective controllers which simply called an Eloquent function that removed the record. Due to the way the database is designed when a quiz or question is deleted, the linking row inside the quiz_question table must also be deleted.

Another thing that had to be changed was to make the foreign key reference columns in the quizzes and questions table cascade delete on deletion of the record. This meant that they would delete the rows in the quiz_question table that referenced the row being deleted. For a quiz deletion, this had to go further and also find the questions associated with the quiz and delete all of those, which used a simple Eloquent function to find them using the quiz_question table before the rows in that table were deleted and then delete the questions using those ids.

3.3.2.4 Testing

Dusk Tests:

1. Test the deletion of a quiz
2. Test the deletion of a question on a quiz
3. Test the edit page of a quiz
4. Test the edit page of a question

3.3.3 Non-Story Work

3.3.3.1 Seeding

Some more data needed to be seeded for this subsection of work, the question data. This involved creating some new Model Factories and using them in the seeders correctly. One issue was trying to create many questions for individual quizzes, but this was overcome using some very basic looping and calling the model factories in the right places.

3.3.3.2 Quiz Description

It was decided that quizzes should probably have descriptions attached to them, in case the lecturer needs reminding of what it is in 6 months time. This involved creating a new migration and simply adding a column to the table.

3.3.3.3 Travis Setup

Travis is a CI tool that can be used to run tests automatically on git pushes. Setting it up involved doing some spike work with another git project. It is really easy to set up, needing you to simply add a travis.yml file which specifies what Travis will do after a push. The project also has to be added from Github which is simple.

3.4 Iteration 3 09/03 - 15/03

3.4.1 Story: Admins can login to the website and run a session with a quiz

3.4.1.1 Analysis - Breakdown of Tasks

This story is quite large and whilst it will probably be split into a number of tasks, the first thing to do was some spike work on the way this system would work. One route is with WebSockets, a relatively new technology that allows the server to push data to the page quickly and easily. This would be nice solution as it is relatively future proof and seems like a better alternative to other solutions that involve a lot of JavaScript and forcing page changes on the users.

3.4.1.2 Design

WebSockets were introduced into Laravel 5 and have become one of the defacto ways to update the front end in real time. Unlike in some other web frameworks such as Ruby on Rails the web sockets in Laravel requires some extra set up. In Rails the WebSockets can be run on the main web server that is used to run the site [7], in Laravel however another server has to be set up to run these. Laravel offers several different drivers for running the WebSockets, including a Redis server and a third party application called Pusher [8]. Pusher handles most of the work for you and requires little set up other than creating a free account [9].

3.4.1.3 Implementation

Pusher was chosen due to its ease of use and due to its high recommendation rate within the Laravel community. A problem with Pusher is that due to it being a third party service, it is not free forever (it has a number of users limitation). However you could host your own Redis server to mitigate this cost. This means that the system had to be designed in a way that the driver for WebSockets could be changed with ease.

After configuring the spike work application to use Pusher, a simple Laravel Event was created to send a message to the Pusher. To test this event there were a couple methods implemented. The first was to simply register a route that triggers it when the page is visited. (TODO: show an example of an event call?)

Or with a custom made artisan command that can trigger the event: php artisan quiz:send message. A command is very useful for testing the event as it can be used without building a button on the front end to trigger it.

Two online guides were used to help write this section of the system due to WebSockets being a new technology [10] [11]. Though they were used to get the basic concepts working, the final code used within the project is tailored to the system and is therefore different overall.

3.4.1.4 Testing

3.4.2 Non-Story Work

3.4.2.1 Refactor Controllers

The first major piece of work was to refactor the controllers and models into a far more sensible structure. The problem was that the Eloquent functions for modifying and reading from the database was within the controllers. In a full MVC system, this functionality should be inside the models. To fix this, the Eloquent functions were refactored into the respective quiz and question models. This would make it easier if anything ever needed to change within the logic for any database interactions.

3.4.2.2 Changing the DB Structure

The original design was changed, and the quiz_questions table for linking quizzes and questions together was removed. The original reason for this table was most likely such that questions could be reused. However, after thinking about the potential for that to happen, and the issues that the structure was causing in the model logic it was decided that the quiz_question table was more of a hindrance than a help.

The questions table now simply has a quiz_id column that references the quiz it belongs to. Doing this means that the relationships between the two tables are much easier to define in the models, simply having a belongsTo and hasMany function in both that automatically return the necessary data. Thanks to the previous refactoring of model logic, changing this functionality was quite quick. Deleting rows in the database was also simpler now that there was no quiz_questions table, all the related questions when a quiz is deleted can be deleted at the same time using the onDelete cascade property in the database.

3.4.2.3 Front-End Setup

This was the first time that any custom CSS was written and Laravel uses sass to generate its css. To build this SASS into CSS, and also to build any future JavaScript, Laravel Mix was needed to run builds for this code. For this, npm and node had to be installed so that they could run their Webpack build scripts. There were some issues trying to get the build scripts to run, even though it worked on fresh installs of Laravel, but eventually a Github issue was discovered that had some solutions. https://github.com/JeffreyWay/laravel-mix/issues/478

3.4.2.4 Dusk and Travis

The aim was to try and set up Laravel Dusk on Travis. Dusk can use a few different browser drivers for running the tests in, by default this is Chrome but Travis comes preconfigured with PhantomJS which is also supported by Dusk. It should be a simple swap in drivers and then to set Travis to run a PhantomJS server. Unfortunately this does not seem to work, and no reason can be found. A supposedly working copy was even cloned and that does not work for me therefore I have to conclude it is currently not working.

3.5 Iteration 4 16/03 - 22/03

3.5.1 Story: Admins can log into the website and run a session with a quiz

3.5.1.1 Analysis - Breakdown of Tasks

After doing some spike work last week, this task can now be approached and broken into several sub tasks:

- Set up config for Pusher
- Add event for broadcasting
- Write a command to trigger this event
- Add a button to quizzes to trigger this event
- Display this on the front end using the JavaScript which listens for Pusher events
- Add a session key box to front page
- Add session id to users
- Allows admins to change their id
- When admin clicks run, this id can be entered into the key box to join a channel with the name of the id
- The user will be presented with the initial quiz page which will be default filled with the name and description of the quiz
- The admin will see the same page but with an ”admin panel”
- This admin panel has a next and previous button for questions
- When these buttons are pressed, the question is sent to pusher
- These question are rendered as a form on the user end and admin end
- The user can submit the form

3.5.1.2 Design

Mockups of the student end of the system:

Figure 3.1: Design for a quiz on a desktop

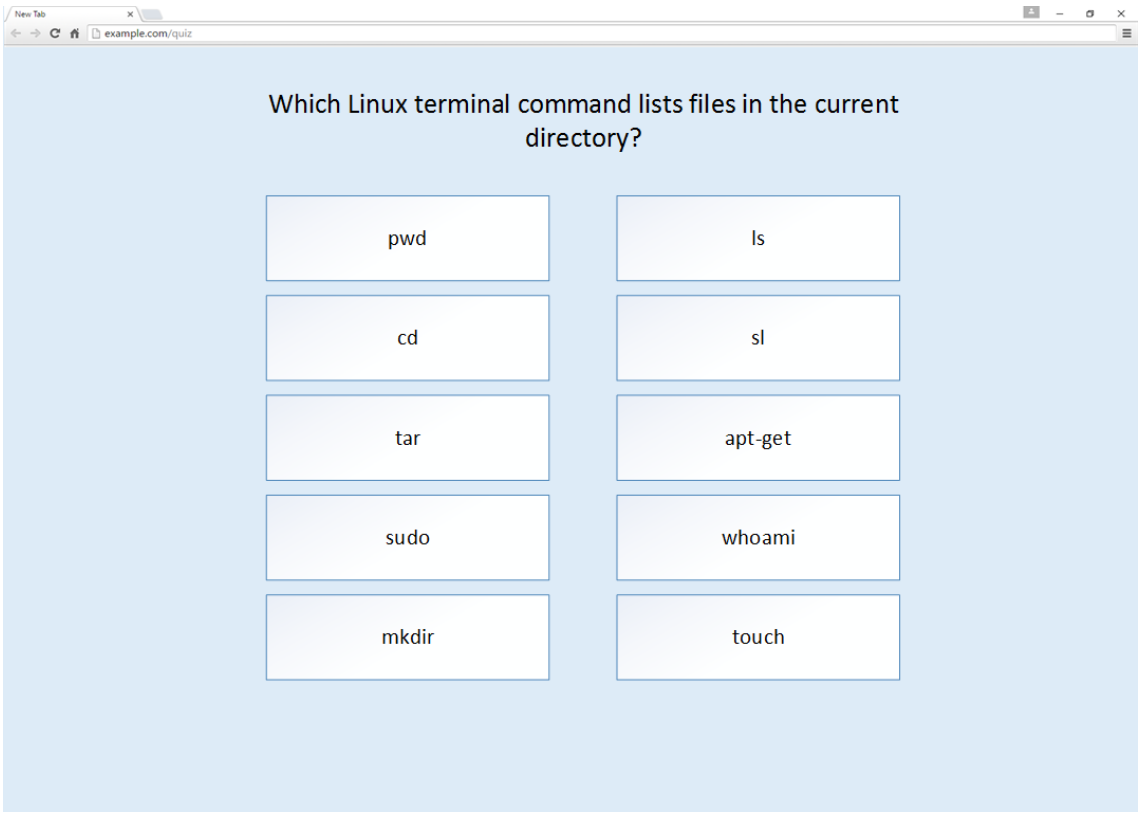
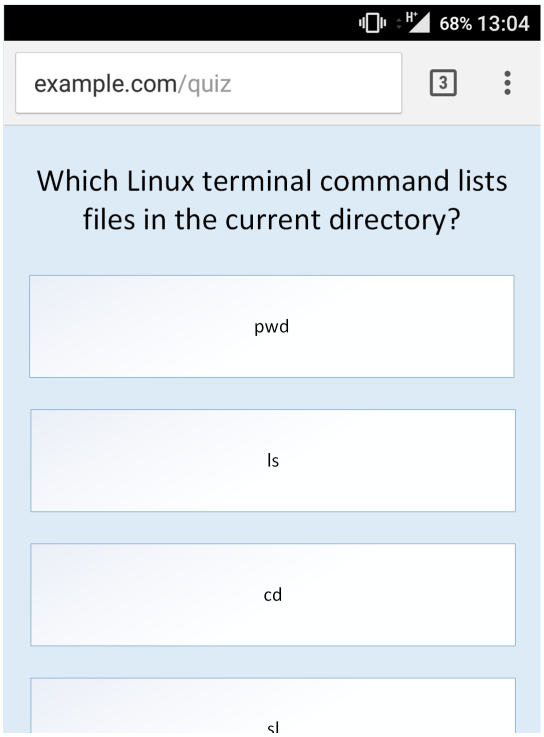


Figure 3.2: Design for a quiz on a mobile device



3.5.1.3 Implementation

It began by configuring pusher, creating an event and writing some basic JavaScript to append to the front page. Some work was done on the admin panel as well, making it only visible to logged in users (the lecturer running the quiz), and adding the functionality for previous and next question buttons. These are buttons that send AJAX requests to quiz controller actions which then trigger the event for WebSockets with the appropriate question data.

Early into the iteration a major flaw with using the WebSockets came up, whilst new content was easy to add to the page, if a new user joined the session late, they would see a blank page/ or the original content of the page that has not yet been removed with JavaScript. To remedy this, the current position in the quiz should be kept track of and the PHP on the page should load the question specified at that position. At the same time, the WebSockets will be running and updating the page from that point onwards in the session. It was decided the best place for keeping track was within the session table, which now has appropriate columns for the position, quiz and if it is running.

Rendering the actual questions calls an action in the question controller, which takes the type of question and the quiz id and position. Using this data it renders one of several available views, one for each type of question like multiple choice or boolean. The views render a simple form showing the question, the possible answers as buttons and a submit button.

For a question to be rendered by default on page load as described above, the page includes the question by rendering the above view. However, it does not know the type of question to render, so it loops over the types and checks if the type of the question is equal to the ones defined earlier in the config file and renders the view if they are the same type. One problem with this is accessing the custom config file. The function to do this was inside the question controller and not the quiz controller.

Whilst the function could be copied, it was better to create one single helper function that is global to the application. This involved creating a helpers file and adding the function there and then registering the helper function in the composer.json file [12].

3.5.1.4 Testing

No testing as the story is not yet complete.

3.6 Iteration 5 23/03 - 29/04

3.6.1 Story: Admins can login to the backend and run a session with a quiz

3.6.1.1 Analysis - Breakdown of Tasks

Tasks left over from the previous week:

- Form on front page redirects to quiz
- Questions rendered as a form
- User can submit the form with their answers
- Questions need limiting on going above/ below max and min

3.6.1.2 Design

3.6.1.3 Implementation

Users still had to connect to the quizzes from the welcome page. A problem with this is that the sessions are being run under the url /quiz/sessionb_name. This means that submitting a GET or POST form will not work as the url cannot be specified, as the user enters the session key. A solution was to have an input field that took the session key and then used JavaScript to redirect to that session. An alternative solution would be to use a form that redirected to a controller action that then itself redirected to the correct session page. Whilst this method would work, having multiple redirections in one request is not good practice or nice for the users browser.

The JavaScript also allowed some client side checking to be done, but there also needed to be server side checks. A custom Middleware class was written to handle the checking of valid session keys. Middleware provide the functionality to filter incoming HTTP requests within controllers [13]. When the form was submitted, the Middleware function would check the database, and if the requested key was not in the database, would redirect back to the welcome page with an error. Else, it would allow the redirection to the quiz session.

When the page receives a message through the Web Sockets, it first removes the content of the page with JQuery. It then checks the position of the quiz, if it is zero, then it appends the quiz start data on to the page. This start data includes information about the quiz, like the name and description. If the position is not zero, then it will render the question at that position in the quiz.

To render a question, an Ajax request is made to a question page that will be populated with the appropriate data using the information from the Web Sockets as the variables for finding the correct question. The Ajax'ed page content is then appended on to the current page using more JQuery.

For the user to submit an answer, the form again had to use JavaScript. This was primarily to make the form user friendly and for different questions types. JS was used to add a class to the options and highlight the one that were chosen, be it one option in a multiple choice question or multiple answers on a multiple selection question. When the student hits submit, it gets the answers by selecting the options with the classes that are added when an option is selected. It

then sends a POST request with Ajax to the quiz session page. The POST'ed data consisted of the answers given, in the form "answer*" where the * was the answer number as well as the question number. This data is not yet handled by the server.

An interesting bug with multiple selection questions was that when the answers were selected, to get just the answer value, the array of elements was iterated over and a new array with the answers is produced. However, due to using a JQuery array function to iterate over these, it created a JQuery collection, rather than a simple array. This collection was not accepted by the Ajax request and this collection has to be transformed into an array with a vanilla JS toArray() function.

The last part of development for this iteration involved adding some simple checks to the next and previous position functions to ensure they did not go below 0 or above the number of questions in the quiz.

3.6.1.4 Testing

TODO: write about the tests

3.6.2 Stories Completed

This iteration and the last have both operated underneath the umbrella of the one story. This is not necessarily accurate as the work that has been completed with this one story covers a multitude of stories. The reason they weren't split up is slightly unclear but part of the reason is the severe overlap between them and some of them being more like functional requirements than areas of work to do. The stories that are now complete:

- Multiple different sessions can be run simultaneously
- The Admin specifies which question is being run by clicking next/prev question
- Sessions can be joined by users via the website
- Up to 300 users should be able to join and answer questions (this story is somewhat subjective, but in theory it should scale upwards with no issues)
- Users answer the question being displayed by the quiz

3.6.3 Non-Story Work

3.6.3.1 User Page

A large amount of work was done on the user pages, this was to ensure that admin could view and edit their details including changing their session keys. This work was rather simple and just involved creating the user blade pages, adding the various functions to the controller and model and adding a little bit of functionality to the authentication actions for registering a new user.

3.6.3.2 JavaScript Clean Up

A lot of JavaScript that was being rendered on the page is not used and useless to the project, in fact the libraries are quite large and take up a significant amount of space. Removing these unused libraries, VueJs being the largest, frees up the JavaScript output file considerably, which will increase load times significantly.

3.7 Iteration 6 30/03 - 05/04

3.7.1 Story: The Admin can show the results of the question in a sensible format e.g. graph

3.7.1.1 Analysis - Breakdown of Tasks

- Save the data
- Ensure no users submit twice
- Read from that data in the admin panel
- Prettify this results tab

3.7.1.2 Design

TODO: add design images here of the admin panel

3.7.1.3 Implementation

There was quite a lot of work to do in this story, the first item in particular was the majority of work. At first it was decided that a simple database could be used to store all the responses however it was soon realised that users could probably submit more than once if we don't associate users with a response. The best way to deal with users that don't login is to use the Laravel session cookie to identify them anonymously.

To store all these responses in a database would be quite a lot of data, rather than having say six rows for a question with six answers that just tallies up the answers the actual session name had to be stored with the answer given so it could be changed. This meant that now there would be a row per answer, equalling at worst case 300 rows per question. This was thought to be a bit drastic so an alternative was suggested: to use a CSV file with each row having the user and an answer, with each question being a separate csv file. These would be stored locally under a session folder in the public directory. These files would be created and destroyed during quiz cycle. This method would also be a good for the downloading of answers from another story.

Ultimately however, trying to write all this data to a csv was too troublesome to be worth it. If a user changed their answer then the entire csv had to be copied with one line changed, editing a single line is not possible with PHP. Therefore it was decided that going back to a database based method would be better, with the idea that at the end of a quiz, all the data would be deleted.

An answers table was created that specified the session, question, user and the answer given. The first three of those were used in a unique composite key to prevent multiple answers from the same users on the same questions in the same session. The model was then written to save this data when data was POST'ed to the /results url. Other functions include those to delete the data when a quiz was ended and various association functions.

To read this data on the front end, an AJAX request was set up from an admin panel button that GETs the same results url which calls an action to render the results data in a JSON format.

The results are in a very basic format in a key-value pairs of answer, total number of selections. TODO: maybe talk about this more

This JSON data could then be used in the ChartJS library which was selected to render the results bar chart on the page. TODO: cite http://www.chartjs.org/ and also http://www.chartjs.org/docs also https://www.sitepoint.com/generating-random-color-values/ also https://stackoverflow.com/questions/240660 php-how-do-you-change-the-key-of-an-array-element and https://stackoverflow.com/questions/37699485/skip-decimal-points-on-y-axis-in-chartjs

3.7.1.4 Testing

3.7.2 Story: Users should not be able to submit their own answers by altering the HTML, as they did with Qwizom

3.7.2.1 Analysis - Breakdown of Tasks

Very few tasks as will be explained in the design subsection:

- Add check for empty arrays on frontend
- Add check for empty arrays on backend

3.7.2.2 Implementation

Thanks to the design of the previous subsection, changing the data meant an empty key was sent if the user tried to change the data submitted to the server. This could be checked in either the JavaScript or PHP end, but it was decided both would be the best for maximum security.

3.7.2.3 Testing

This is hard to test, as Dusk does not allow the changing of HTML on the page or the possibility to submit false data. User testing might be a good replacement for Dusk tests here.

3.7.3 Story: The Admin can see what percentage of users connected to the session have answered

3.7.3.1 Analysis - Breakdown of Tasks

- Get total number of responses
- Post this number on the page as part of the results form

3.7.3.2 Design

This should appear above the results table, possibly as part of the title.

3.7.3.3 Implementation

After attempting to get a number of users connected to the channel it became clear that this number is not readily accessible for public channels. It can be gotten via the Pusher API but implementing that would mean locking some functionality down with Pusher and the long term plan is probably going to involve changing to a Redis based WebSocket system. Therefore this number has just been changed to the total number of responses which should give a good idea to the lecturer of what percentage of users have responded. This was simply added to the title attribute of the chart.

3.7.3.4 Testing

3.8 Iteration 7 06/04 - 12/04

3.8.1 Story: Site should be mobile responsive

3.8.1.1 Analysis - Breakdown of Tasks

- Create standard media query sizes
- Create media query for quiz questions
- Media query for welcome page

3.8.1.2 Design

A quiz page should look like the design specified in iteration 4.

3.8.1.3 Implementation

Finding the media query sizes was simple, Bootstrap recommends some default sizes for phones, tablets etc. (TODO: cite this https://getbootstrap.com/css/#grid-media-queries) It was then a simple task of using the Chrome developer tools to view the page in mobile view and change the various sizes of buttons and titles in the CSS editor to be more mobile friendly.

3.8.1.4 Testing

This is not really possible to test with automated testing, but should be easy to test with users later in the project. Additionally once live the system could be tested with the Google Responsive Test site that gives a score for the usability of a site in mobile view.

3.8.2 Part Two Re-Design

Originally planned to be an extension to Microsoft PowerPoint or a similar technology, such as Libre Office, this was determined to be a large amount of work (TODO: cite this). This meant an alternative solution had to be devised or some extra requirements had to be added to the system. An alternative solution was found rather than abandon the idea of having slides within the quizzes. Slideshow programs usually have the ability to render their slides as PDF, a format which is more heavily supported compared to a propriety format such as .pptx provided by Microsoft. If a PDF is uploaded to the application, PHP can be used to turn these PDF slides into images, which can then be rendered on the quiz pages.

This new approach means some changes to the original stories for the second part, here are the revised stories for this part:

- (1) The admin creates slides in their preferred editor and exports them as a PDF
- (4) The admin can upload these slides to a quiz they have created in the past
- (3) The admin can reorder the questions within the quiz to move them around the slides
- (2) When this quiz is run, it should render the slides as well as questions in the order specified

There are some disadvantages to this however. The main issues is that slide animations are not rendered as separate slides on the PDF. There are extensions for Microsoft PowerPoint that let the slides be rendered with animations occupying separate slides so as to provide a ”fake” animation. Another problem is that the slides would have to be uploaded before a lecture as it can take a few minutes to render PDF slides as images. This could be argued as an advantage however, if lecturers upload their slides before a lecture they only need to log in to the application when then want to run them, no need to bother carrying the slides on a memory stick or saving them to their University storage.

3.8.3 Story: The admin can upload these slides to a quiz they have created in the past

3.8.3.1 Analysis - Breakdown of tasks

- Upload pdf slides
- Convert these slides to images
- Save references to these slides in the database
- Show the slides on the Quiz.show page

3.8.3.2 Design

TODO

3.8.3.3 Implementation

To upload the slides, it was done with a standard HTML file input but the backend used techniques native to Laravel: (TODO cite: https://laracasts.com/series/whats-new-in-laravel-5-3/episodes/12). These images are then saved in the /storage/app/public/slides folder. Within this, they are organised into folders named after the session id and then quiz id within those.

These slides are then converted using a library that uses Imagick - (TODO: cite this https://github.com/spatie/pd to-image). Whilst saving them, references are also saved to the database. This is so that the images can be referenced quickly later on when displaying an slide or question as database reading is quick. For this functionality, a new table for slides was added to the system, storing the file name, quiz it was associated with and a position.

The last task involved showing the slides and questions on the Quiz.show page. Seeing as the new story involved reordering them, simply rendering the slides after the questions would not do, it should render them in order of the position. The main problem is that there is no simple SQL query that can select two sets of results and then order them by a common column. The easier solution was to create a PHP function to create an array of all the items in the order wanted. (TODO: Cite this function in appendix, should I explain here or in the appendix?)

3.8.3.4 Testing

3.8.4 Story: The admin can reorder the questions within the quiz to move them around the slides

3.8.4.1 Analysis - Breakdown of tasks

- Add buttons to reorder questions
- This button increases or decreases position of question
- It swaps the positions of two questions or a slide and question if required
- Add some limits to positioning like not going below 1

3.8.4.2 Design

As a minor change to the system, no major design choices were made here. Simple Bootstrap icons can be used for the position changing arrows, which is the only front end change.

3.8.4.3 Implementation

Main functionality to add was the changing of positions. This was rather simple, adding a simple Route and buttons that triggered the actions associated with the new route. It was decided that the Route would be a POST request rather than GET as technically it is submitting some data, the direction and id of the question. The action it calls simply updates the position in the database after doing some checks of the position it wants to move to. If it is already occupied, it swaps the object at the new position with its current position. This object could be a question or slide.

3.8.4.4 Testing

3.8.5 Story: When the quiz is run, it should render the slides as well as question in the order specified

3.8.5.1 Analysis - Breakdown of tasks

- Find the position in the database and decide whether its a slide or question
- Render a question like normal
- If a slide, render a simple img tag and populate with the image name from the WebSockets
- Resize image to fit the page, including for mobile responsive
- Render question by default for users joining late

3.8.5.2 Design

This should work in much the same way as the questions, in that it will send an Ajax request to a page with the image name and then copy the content of the ajax page into the live page. The content page to Ajax should be pretty simple, a simple container with an img tag. The other tasks are extensions to previously created functions.

3.8.5.3 Implementation

The first task is to determine what is at the position, this involved changing the current function to check if the question at the requested position was null, if it was then it should find a slide at the specified position. As well as the data about the slide or question, it would return a type so that when it was called the system knew what type of content it was to expect. The data was then sent to the WebSockets and another case was added to the JavaScript receiving end for triggering an AJAX call to a slide page. This page would render a simple img tag with the slide specified which is then added to the current page like the questions.

A problem with rendering the images was the files stored within the /storage folder are not publicly accessible, usually only files within the /public folder are. However, there is a simple artisan command to create a symlink from the public folder that points to the storage/app/public folder. TODO add command: php artisan storage:link

3.8.5.4 Testing

3.9 Iteration 8 13/04 - 19/04

3.9.1 Report Writing

The majority of this iteration was devoted to report writing. Whilst there is still some development to do, the report accounts for 30% of the final grade and needed to be somewhat worked on. There were some small changes to phpdoc and comments added to the code, but no actual functionality was changed. In terms of the report, the first chapter concerning the background had been added. All the iterations documents have also been added into the final report, though they will likely need changing before hand in. There are a lot of TODOs scattered throughout, mostly for citations but also some to check with the project supervisor about certain details. Finally, some content was fleshed out for the testing chapter, giving an overview and descriptions of the technologies.

3.10 Iteration 9 20/04 - 26/04

3.11 Iteration 10 27/04 - 03/05

Chapter 4

Testing

4.1 Overview

At the start of the project, testing was to use two practices from Agile. These were Test Driven Development and Continuous Integration (CI). Whilst these two practices both failed due to a mixture of reasons as described in the Iterations chapter, testing was still performed within each iteration. Rather than go over the tests made each week within the previous chapter, this section will summarise the tests. Overall there were x tests (TODO: fill in once development finished) that test the overall functionality of the system. Unit tests were dropped for a number of reasons in favour of application tests (TODO: why? take from iter docs).

4.2 Application testing

TODO: screenshot final output of tests, possible one running for appendices. Also could list all tests somewhere. To do application tests, a testing framework called Laravel Dusk (TODO cite https://mattstauffer.co/blog/introducing-laravel-dusk-new-in-laravel-5-4) was used, which is the default testing framework provided with Laravel 5.4. It runs the tests within a Chrome instance by running a standalone server which then calls the Chrome application installed on the machine that the tests are being run on. The advantages of running tests like this rather than say executing PHP on the server and reading a virtual DOM is that this allows JavaScript on the page to be evaluated and executed. JavaScript is vital to many of the components of the application so if it was not usable within tests, many parts of the system would have to be ignored within the tests.

These tests test the application as whole, rather than individual bits of code. This allows the tests to be written with the original requirements, the stories, in mind and so makes these tests a good way to evaluate the application at the end of its development. Tests were organised by story, as these provided an area of the application that needed to be checked, which makes them more human readable for any future development.

Tests use a test database rather than the production one to ensure the integrity of data on the production system. This test database is left intentionally empty when, with each test performing a migration for the test. Within each test the database is populated with relevant data using a number of database factories. These factories allow the easy and quick creation of any data needed within

the test, though using a normal Model to input data would also still work. At the end of each test, the database is rolled back for the next test to run, leaving an empty database for a new migration and data.

Database transactions could have been used instead, which use a pre migrated database and then any data input during each test is deleted at the end of the test. Whilst the test db could be migrated before any tests were run to allow the use of transactions, this would mean than any future changes to the migration files would mean that the developer would have to remember to re-migrate the test database as well. Doing a migration for every test might be somewhat resource intensive, but it allows the tests to be written and run completely standalone from each other with no need to tie into any earlier setup functions.

4.3 Security testing

One story specifically concerns security: Users should not be able to submit their own answers by altering the HTML, as they did with Qwizdom. The tests for this however are not that easy to do within the confines of Dusk due to the nature of the framework. The framework is concerned with user facing application tests and has no ability to change the DOM of the page or send custom POST requests to the server. This means that this story is better tested in the User testing section.

Other aspects of security can be checked however, being a web application there are a number of well known attack types including SQL injection, cross site scripting and cross site request forgery attacks. These are tested in (TODO: screenshot of these in action)

4.4 User testing

TODO

4.5 Stress testing

Though there are a number of tools out there to do exactly that, there was no time to set up and any official stress testing. However, the user tests within the lecture helped a give a good idea of the stress the system could take. Like many sites though, the application was mostly restricted by its server setup rather than by its design. (TODO: prove after user testing)

4.6 Automated testing

Whilst CI was abandoned early on into the development of the application, the Dusk tests are easy to run and automatically execute within a browser if the appropriate browser drivers are available. This means that it should not be too hard to integrate these tests into a CI tool if required in any future development.

TODO: Stuff to talk about: comprehensiveness of testing incl potential of using bdd and cucumber in future edge cases? how did we manage those (user and application)

Chapter 5

Evaluation

5.1 Methodology

Got rid a couple of practices Liked adaptability Liked stories Like iterations Part 2 changes Overall happy with that

5.2 Tools and Technologies

Laravel, Dusk, Websockets, Javascripty stuff, Chrome? Editor? –Better tools?

5.3 Extra Tools

Diary and Trello

5.4 Story Comparison

5.5 User Testing Evaluation

5.6 Summary

Appendices

Appendix A

Third-Party Code and Libraries

Appendix B

Ethics Submission

Appendix C

Code Examples

Appendix D

User Survey Results

Annotated Bibliography

[1] Qwizdom, Inc., "Experts in Audience Response Systems & Training — Qwizdom UK," http://qwizdom.com/uk/, 2017.

Qwizdom is the software that this project is primarily based off.

[2] "Architecture of laravel applications - laravel book," http://laravelbook.com/laravel-architecture/, 2017.

Describes the architecture of the Laravel framework in more detail

[3] "Validation - laravel," https://laravel.com/docs/5.4/validation, 2017.

Documentation for how server side validation works within Laravel

[4] "Csrf protection - laravel," https://laravel.com/docs/5.4/csrf/, 2017.

Documentation for how Laravel handles protection from cross site request forgery attacks

[5] gaddygab and georaldc, "[question] why dusk page test uses databasemigrations instead of databasetransactions? issue #110 laravel/dusk," https://github.com/laravel/dusk/issues/110, February 2017.

Describes why migrations have to be used rather than transactions, posted as an issue to the test frameworks Github page

[6] "Controllers - laravel," https://laravel.com/docs/5.4/controllers#resource-controllers, 2017.

A list of the routes automatically bound when a controller is specified as a resource controller.

[7] "Action cable overview - ruby on rails guides," http://edgeguides.rubyonrails.org/action_cable_overview.html, 2017.

Describes how RoR uses WebSockets. Good for a comparison of other languages and another framework.

[8] "Broadcasting - laravel," https://laravel.com/docs/5.4/broadcasting, 2017.

Documentation for broadcasting within Laravel, which is what it calls WebSockets

[9] "What is pusher? building real-time laravel apps with pusher," https://pusher-community.github.io/real-time-laravel/introduction/what-is-pusher.html, 2017.

Guide on what pusher is and how to set it up with Laravel, though this guide was not used for that purpose, merely to gain an understanding of how Pusher worked.

[10] "Writing realtime apps with laravel 5 and pusher - pusher blog," https://blog.pusher.com/writing-realtime-apps-with-laravel-5-and-pusher/, June 2016.

Guide that was used to get the basics of Pusher and how to do use WebSockets within Laravel

[11] "Introducing laravel echo: an in-depth walk through," https://mattstauffer.co/blog/introducing-laravel-echo, June 2016.

Guide that was used to get the basics of Laravel Echo and how to do use WebSockets within Laravel

[12] Way, Jeffrey. and calebeoliveria, "Best practices for custom helpers on laravel 5," https://laracasts.com/discuss/channels/general-discussion/best-practices-for-custom-helpers-on-laravel-5, May 2016.

Discussion of how global helper functions should be registered and used and used within Laravel

[13] "Middleware - laravel," https://laravel.com/docs/5.4/middleware, 2017.

Description of Middlware in Laravel, a way of filtering incoming HTTP requests to a controller. An example would be like checking if a user is logged in.