**Programming Project A453**

Contents

[Task 1 1](#_Toc450685885)

[Planning 1](#_Toc450685886)

[Language and Technologies 1](#_Toc450685887)

[Task Overview and Solutions 2](#_Toc450685888)

[UI Planning 2](#_Toc450685889)

[Coding 3](#_Toc450685890)

[Testing 8](#_Toc450685891)

[Evaluation 9](#_Toc450685892)

[Task 2 10](#_Toc450685893)

[Planning 10](#_Toc450685894)

[Coding 10](#_Toc450685895)

[Testing 11](#_Toc450685896)

[Evaluation 12](#_Toc450685897)

[Task 3 12](#_Toc450685898)

[Planning 12](#_Toc450685899)

[Coding 13](#_Toc450685900)

[Result Browser 13](#_Toc450685901)

[Class Management 17](#_Toc450685902)

[User Management 19](#_Toc450685903)

[Evaluation 24](#_Toc450685904)

[Appendix 25](#_Toc450685905)

Although a large amount of code has been included in this document, the full program consists of over 1,500 lines of JavaScript, CSS, and HTML, which was too much to include here. The entire program has been fully commented, so I recommend viewing that in parallel.

Included with this document is a collection of screen recordings, used both for testing and for a general overview of the functioning product – please see the */GIFs/* folder for these.

# Task 1

## Planning

### Language and Technologies

Because the program will likely be used for schools/teaching, then I want to make it as easy as possible for the students to access wherever they are, whether it be at home or at school. This means I have chosen to build my program (known as Arithmetic) on a JavaScript/MongoDB framework known as Meteor. I chose Meteor as it well suited to the quick creation and distribution of browser-based apps, with built in reactive rendering, to allow me to easily port the application to run on mobile devices. The integrated package system makes it quick and easy to import new features and tools available online, and once I have finished writing my app, it can either be distributed as a binary, or published to Meteor’s own backend – more information about the Meteor stack can be found in the appendix (1.1).

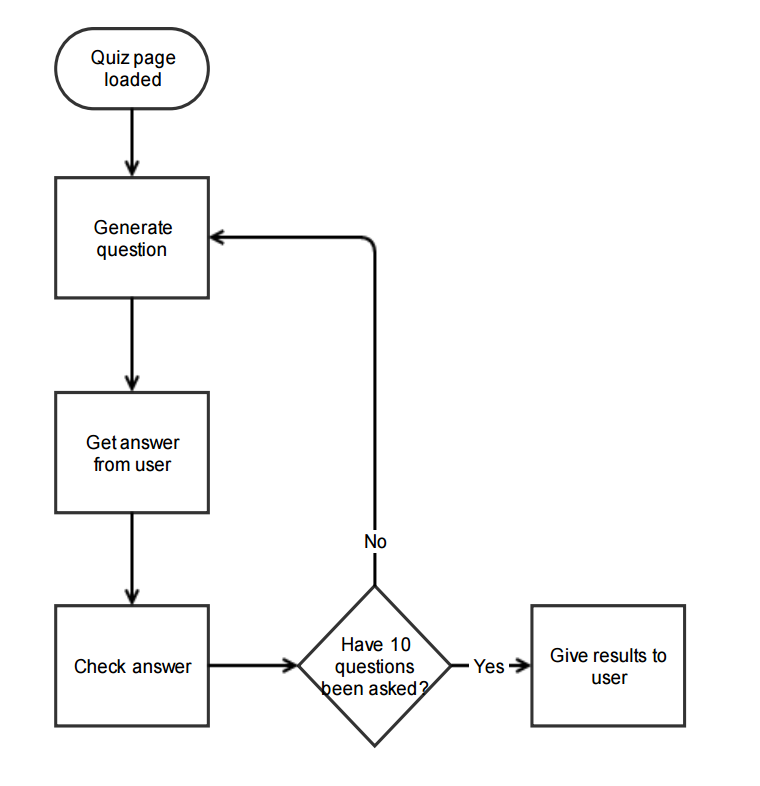
I originally intended to use a C variant, and utilise the WINAPI to create native Windows applications. This would then interact with a SQL server for data storage/retrieval. Having later found Meteor, I decided to use that instead. JavaScript is much simpler, and already uses technologies I am familiar with (HTML/CSS for interfaces & MongoDB). This also meant that application could be cross-platform (web application) – much more suitable for an education environment where deployment and management of applications can be difficult.

Because the teacher wants to use this system to manage a class of students, a simple account system would be useful to keep track of individual student’s performance over time, as well as letting students submit work in their name from anywhere. Meteor includes a full account management system built in, which I will utilise to create a secure, encrypted system to prevent unauthorised users from accessing the student/teachers data.

For storage, the framework includes a persistent MongoDB database system. This is automatically set up, without any manual actions required, or external databases to be set up. MongoDB is also well suited for quick scalability, so if the system were to be rolled out to a larger user base, it won’t have a problem. This also means that I will be able to store student results for as long as required, as they require very little space, and can be easily searched through.

For the interface, Meteor will be running its own web server, through which Arithmetic can be accessed. This means I will be using HTML and a templating engine called Handlebars, which will allow me to pass data from the JavaScript logic into the webserver, so displayed data will be updated live without the need to refresh.

### Task Overview and Solutions

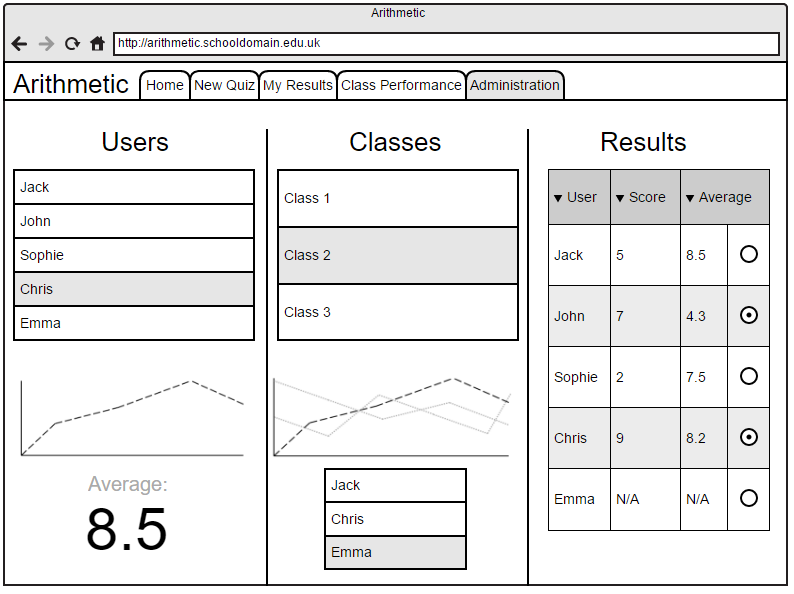
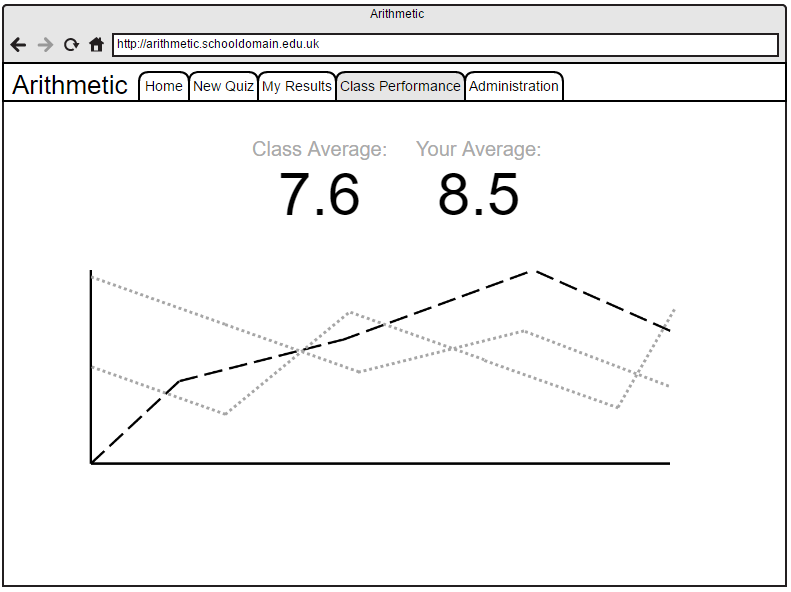
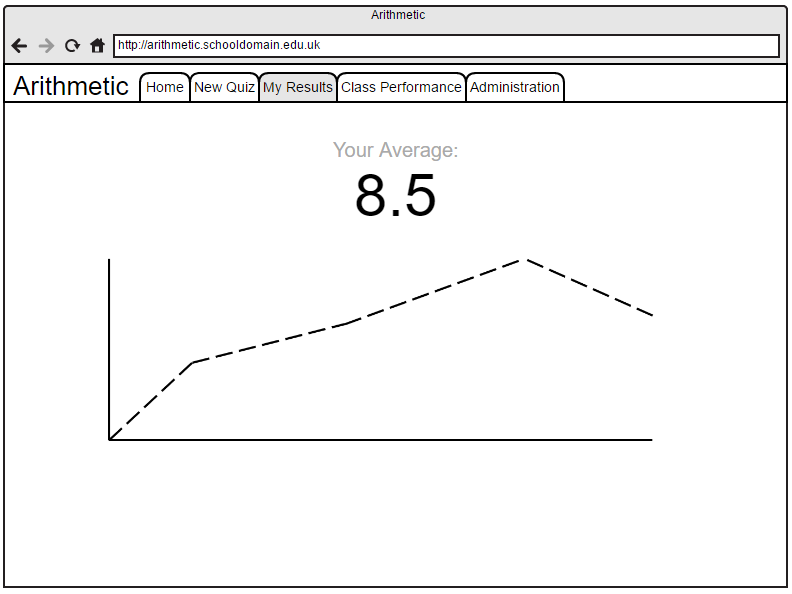
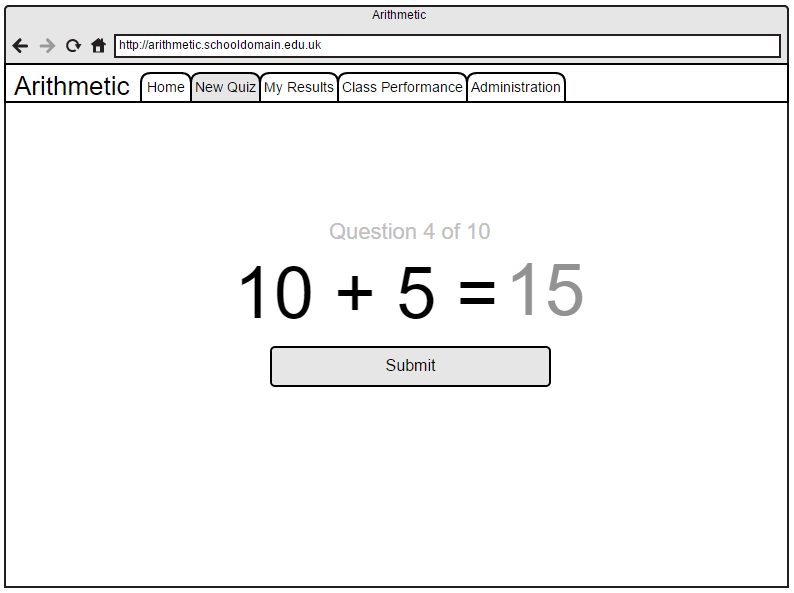
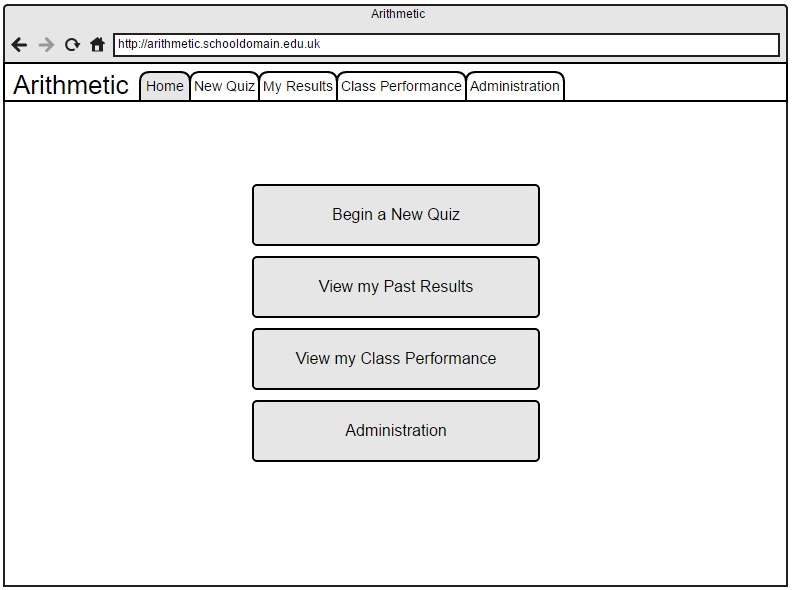
The task requires the program to be able to generate basic arithmetic questions, and then display them to the student. The student then needs to submit an answer. This will be repeated 10 times until the quiz has been completed, whereupon the results need to be marked and feedback given to the student on their performance. The flowchart (Figure 1) shows the basic logic that I will need to complete this task. This can then be easily expanded later to complete the other two tasks.

The main requirements of Task 1 are:

* Generate a quiz using random questions
* Each question should use two numbers, addition, subtraction, and multiplication.
* Use positive numbers between 1 and 10 – “basic arithmetic”
* Get the student’s name
* Give feedback on their answers to the question
  + Show the question, their answer, and the correct answer to the question. Can also show whether the current question was answered correctly or not.
* Score the quiz out of 10.

### UI Planning

I created a series of wireframe mock-ups of basic ideas for how the finished product should look. The layout should be fairly simple and easy to use for all ages. Becauise it will be displayed in a web browser, I will be using traditional navigation bars to navigate around the application. These will always be present to the user always knows how to change pages and where to do so. I can also display information about who is currently logged in in the navigation pane too. I chose to name the program simply ‘*Arithmetic’.*



## Coding

Due to the nature of the framework I am using, individual functions will not run chronologically – many are called depending on which page of the application is being viewed, and the actions of the user (e.g. pressing buttons or entering text).

To create a working application in Meteor, I need to create a ‘Template’ (HTML file), and then associate some logic (JavaScript) to run when that template is called (Web page is viewed) and interacted with (e.g. buttons pressed or forms submitted).

The template for the quiz is relatively short – I only need to display the question, question number and provide a way for the user to submit their answer. I have also written CSS to improve ease of use, and give feedback as appropriate. I chose to use the HTML/CSS/JS framework Bootstrap to aid with UI development – this made creating responsive, grid-based pages much easier. Having made the basic HTML file, I then used CSS to make the page look good, as well as work well on mobile devices. This increases ease of use (further cross-platform support), and makes the program simpler by directing user’s attention towards the important elements (e.g. the question and its answer).



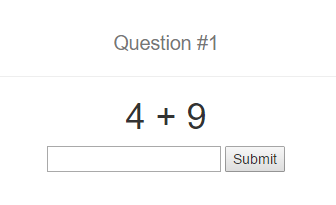
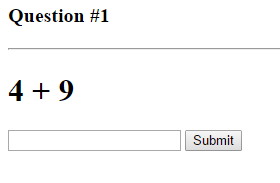
Next, the logic for that template needs to be written. The following code is run as soon as the page has been loaded (Identified by the Meteor callback *.rendered*). I first call the function *nextQuestion()* (explained later), and then resets a series of variables so any previous quiz data is not present. Meteor uses a Session key/value store stored on the client. This means any user viewing the page will have their own variables independent of the server and each other, which can be accessed and manipulated by the server when needed. 

Figure 2 Quiz page without CSS styling

Figure 3 Quiz page with CSS styling

Next, the event logic is needed. When the user triggers an event (e.g. clicking a button or submitting a form), the *.events* callback is triggered, running a function corresponding to the event.In the template, the user’s answer is entered into a *<form>,* and then a submit button is pressed. Meteor intercepts this submit as an event, through which I can add my own logic. In the code below, I first use the built in *preventDefault()* function to prevent the usual behaviour of HTML forms (would usually refresh the page). We can then retrieve the answer from the textbox (*event.target.answer.value)* before resetting it back to an empty box. Now we can check whether or not the user’s answer is correct using the *eval* function. This will execute a given string (in this case, the question), and therefore calculating the answer to the question. This means I don’t have to store the question, the actual answer, and the user’s answer all together, as the actual answer can be calculated on-the-fly. This also ensures that the user’s answer is always being compared to an answer we *know* is the correct answer to that question, and also simplifies Session storage of the ongoing quiz.

A simple *if* statement can then be used to check the answer against the question. Depending on whether their answer was right or wrong, we can change the background of the *<body>* and move onto the next question. Again, I am using Session variables to store the results and the user progresses through the quiz, using the function *addToAnswerLog()* (shown below). Finally, I can check whether or not the user has answered 10 questions. If so, we can go to the results page, and if not, we can generate another question and repeat the process.

Having received the user’s answer, we can then check it against a number of criteria. First, I used an *if* statement (line 5) to check whether the user has actually entered anything. This can be done by comparing the *.lenth* of the string to 0. If nothing has been entered, we can log this to the console.

Next we can check whether any invalid characters have been entered (line 7). I chose to use REGEX for this, as it is very flexible, and the syntax is somewhat simple. For the detection of non-digit characters, I simply had to use *‘[\D]’.* Having detected any non-digit characters, I can then log this to the console.

Finally, I can check whether the number given is correct. I can then use the same *eval()* function to calculate the correct solution to the answer, and then compare the user’s answer to that. If correct, we can flash the screen green, and if wrong, we can flash it red. Regardless of whether it is right or wrong, we then store the question and the user’s answer in the *answerLog.*



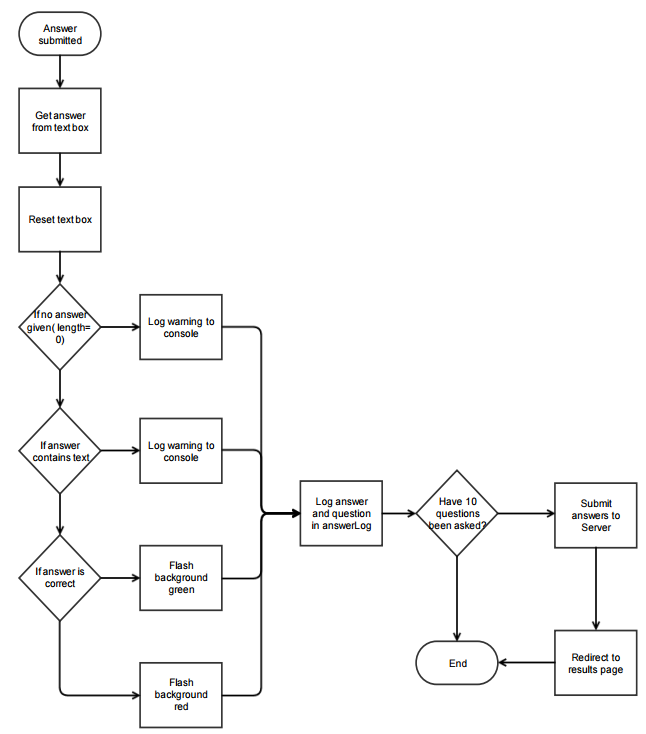


Figure 4 - An illustration of the above code

The *nextQuestion()* function is also fairly simple – it calls the server method to generate a question, and then stores the returned string in the Session. I can also log any errors to the console should they occur during the generation. Calling server functions can be tricky in Meteor, as they are run asynchronously. This means that when I call it I will need to provide a callback so I can be sure that the generated question will be returned before the question is stored in the Session.



On the server, the following code will then be ran. This will generate two random integers, and then pick an operator for the numbers (such as + or -). One problem created by generating random numbers is that very occasionally, two questions may be very similar or even exactly the same. Whilst I could iterate over the *answerLog,* this is probably unnecessary for the rarity of that event. That would also mean I would have to pass all past answers to the server, which will also decrease performance further.



The Session’s answer log is just a simple array with each item containing a sub array of the question and the answer. After each question, the data is pushed into this array and then the Session’s *answerLog* is updated. I chose to use an array because I know that I will be storing 10 sets of questions and answers, which I will need to iterate over later on. A set of individual variables would make this very inconvenient.



To show the question and question number to the user, I am using *.helpers* to pass information into the templating engine, Handlebars. In the templates, I can insert placeholders, which will trigger this function to retrieve the data needed. For both the question and the question number, all I need to do is just return the Session variable set in the previous question generation functions. This data is updated live, so I only need to do this once, and needn’t worry about updating or refreshing the page.



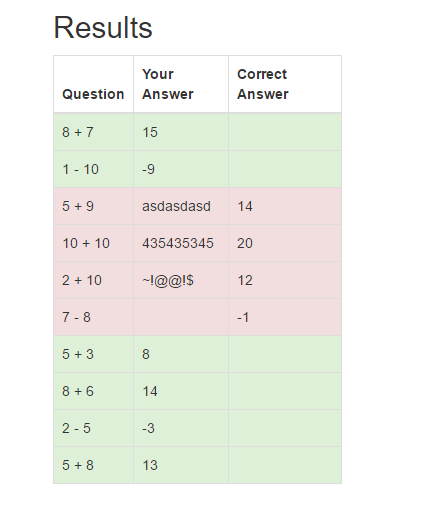
Once the quiz has been completed, we need to give the results to the user. To do this, they will need to be redirected to the results page (*‘/results’),* where we can use the Session key/value store to retrieve the *answerLog* arraystored by the quiz.



The results page is fairly simple compared to the quiz page, as we only need to display a table of the questions and answers to the user – no *.events* or *.rendered* callbacks needed. I am using a plugin called *reactiveTable* to generate the table automatically – all I need to do is pass the array and a few configuration options, which are set using the above helper. This helper is then passed into the package in the Handlebars code in the Template.



All if this together produces the results table shown to the user, with the question, their answer (you can see the testing inputs used) and the actual answer to that question. In later tasks, I can store the results of the quiz in a database, and these tables can be generated as needed.



## Testing

For the first test, I entered a correct answer, a wrong answer, a negative answer, a wrong answer only using letters, and a wrong answer using symbols.

**--Please see /GIFs/Tests/Task 1 – Test 1.gif for a recording of this test—**

As you can see from the recording, the first second and third tests worked as they should. Given a wrong (including negative) number would trigger the wrong answer flash, and a correct answer gave the correct answer flash. We can also see that the questions are being generated properly, and the question counter goes up with each question. One problem that is immediately clear is that any answers containing letters did **not** trigger the wrong answer correctly. Looking back at the code for the *Template.quiz.events* I can see why – the REGEX to check the input only checks for non-digit characters, and if one is found, the console logs that it was incorrect, but does not trigger the wrong answer lines. To fix this problem, I can simply remove the lines that check for letters or an empty input, as these will be marked wrong by the *eval()* anyway. Either way, feedback wasn’t given to the user about their answer (unless they were reading the developer’s console in the browser). Later on, I could notify them to only submit numerical answers and let them retry the question. Another issue raised by only using *eval()* is that worded numbers (e.g. ‘ten’) would also be marked wrong without any feedback, though this would encourage them to use the correct format, something that would have to be taught in later years anyway.

Having made this change and retesting, we can see that the quiz is functioning properly now.

**--Please see /GIFs/Tests/Task 1 – Test 2.gif for a recording of this test—**

## Evaluation

|  |  |  |
| --- | --- | --- |
| Criteria | Met? | Comments |
| Generate a quiz using random questions | ✔ | Quiz questions are created using the *Math.random()* function for both the numbers and the operation. |
| Each question should use two numbers, addition, subtraction, and multiplication. | ✖ | I did not include multiplication questions – I will need to alter my program to do this. |
| Get the student’s name | ✔ | I used Meteor’s built in user account system – each student will have their own account. |
| Give feedback on their answers to the question | ✔ | A results page is generated after every quiz, showing what they answered correctly and where they went wrong. |
| Score the quiz out of 10. | ✖ | Whilst I did give a table, I didn’t include an actual score out of 10 – I will need to alter my program to do this. |

2 of the required criteria were not included, so I corrected my program to include these mistakes.

For the first problem, I had to change my random operation chooser to instead randomly pick a number between 1 and 3, and from that number use a specific operation. The new code is shown below.



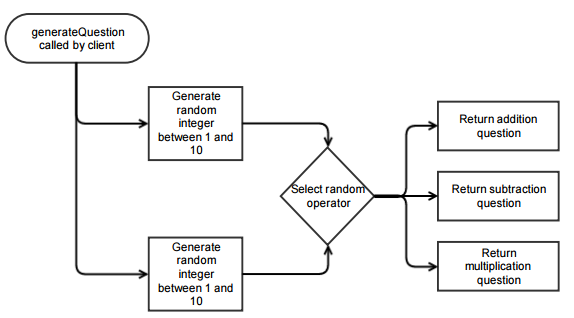


Figure 5 - An illustration of the above function

I then tested this by cycling through a quiz to check that all three types of question were being asked:

**--Please see /GIFs/Tests/Task 1 – Test 3.gif for a recording of this test—**

As the test shows, all three operators are being included in the question types.

Showing the score on the results page was also a fairly simply addition – I already kept the score out of 10 as a Session variable anyway, so all I had to do was add the Handlebars variable and the associated helper:

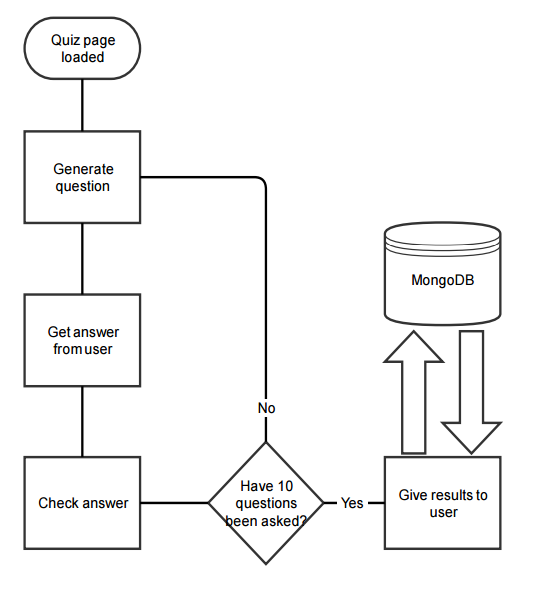
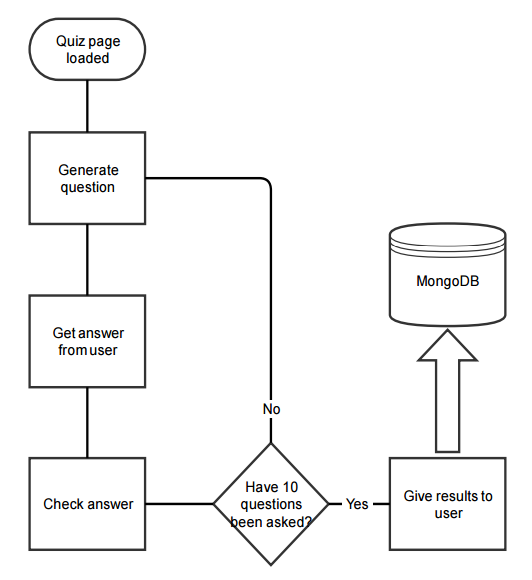




Now that those two criteria are fully added, Task 1 has been completed.

# Task 2

## Planning

Task 2 requires the addition of a results storing solution. Another reason why I chose to use Meteor was because it runs a fully configured MongoDB database server alongside the webserver and NodeJS server. MongoDB (Mongo) is a document oriented NoSQL database system - it is much more flexible than SQL databases, and queries are often much simpler – a more in-depth discussion of MongoDB vs SQL can be found in the appendix 2.1.

Integration into my existing program will be fairly simple – all I need to do is call a built in function to insert the data into the database.

The new flowchart for my program is shown on the left – having completed the quiz, the program needs to insert the data into the database, and should also keep track of the class the submitting user belongs to.

The task criteria are as follows:

* Record and store quiz data

Data will be stored in MongoDB, along with the user who submitted it, and the date & time of submission.

* Record class information (Class 1-3) so data can be separated and compared.

## Coding

Since the database already exists (created by Meteor automatically), all I need to do is create the *collection* (Similar to *tables* in SQL databases). *Mongo.Collection()* constructs the collection, or if it already exists, creates a new connection to it. Specifically, an in-memory, non-persistent implementation of Mongo, called *Minimongo,* instance is created, which serves as a cache and does the actual interaction with the MongoDB server. Any commands or queries are executed on the client’s copy of the database (Through publications and subscriptions), and simultaneously sent to the server and executed there too.



Now that the collection has either been loaded or created, we can interact with it. The code below shows how we can use the *.insert({})* function to add data to the collection. This function is running on the server for increased security (more on this later), so the client will need to call the *submitAnswers* function and pass the data. I have chosen to store the ID of the user, their username, the date/time of submission, number of correct answers, and finally the array of questions and answers in the database. I chose to include the ID and the username for two reasons – firstly, when passing the object into the template to be displayed to the user, I am unlikely to need the raw ID, but instead the username. This saves having to lookup the username of the user for every quiz result, which could be a lot for large queries. Secondly, in the future, should the ability to change usernames be created the results can still be searched by the creating user’s ID, which will never change. Together, all of this data will provide enough data to generate reports or look in-depth at a user’s performance.

Using a proper database system is much more efficient and faster than a text file or CSV file. The inherent organisation of a database makes querying and data lookup much easier and faster. Should the teacher be so inclined, they could have a centralised database to store the entire school’s quiz data – the Mongo server is available for any program to write to, should they want to manage their data another way.



To implement this new functionality in the client, we just need to add the *Meteor.call()* function, and pass the required data. This happens before the user is redirected, but this shouldn’t impact performance, as server functions are executed asynchronously and are non-blocking.

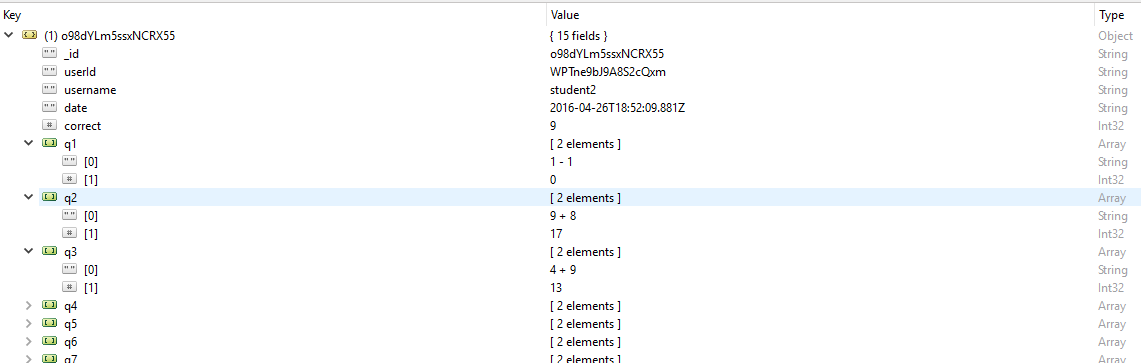


Along with information about the quiz, I included the ID of the submitting user – this will stay the same regardless of Class, and by looking up the ID associated with that quiz, I can in turn get the class of that user. By having classes be an attribute of a user, then can be moved into different classes or roles, with their quiz results staying the same. By not hardcoding each quiz to each class, the system will be much more versatile should a user move between classes.

## Testing

To make sure that the data is being inserted into the database correctly, I used a tool called *Robomongo* to query the MongoDB server separately of Meteor. Having first submitted a few quizzes, I queried the *Data* collection. The screenshot below shows that data is indeed being stored in the database.

Expanding one of the stored objects shows that the fields are being populated correctly.



## Evaluation

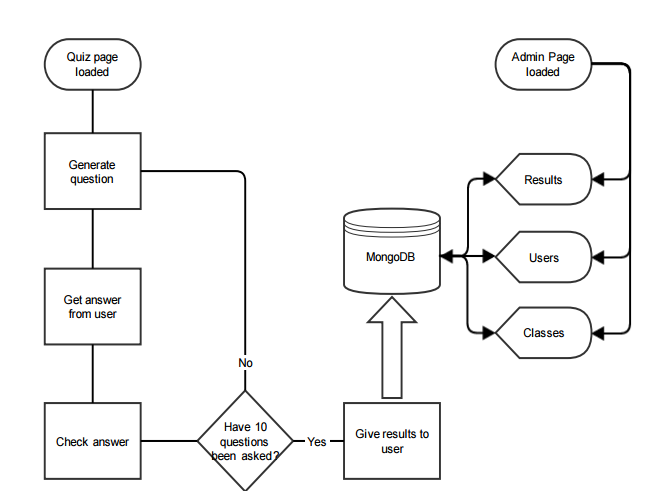
|  |  |  |
| --- | --- | --- |
| Criteria | Met? | Comments |
| Record and store quiz data | ✔ | I can see from the database query that data is being stored correctly by MongoDB. |
| Record class information (Class 1-3) so data can be separated and compared. | ✔ | By including information about the user, I can get information about the class they are assigned to. By not having the class hardcoded into the database object, users can move between classes without the need to alter all their submitted quiz results. |

For the class system, I used another Meteor package called *alanning/meteor-roles.* As I am using an account system, I can set a ‘role’ for each user, signifying which class they are in. By doing the class sorting at a user level, users can be easily moved between classes without the stored quiz results having to be moved too. This increases efficiency and performance. Due to the size of each data object, many hundreds of thousands of quiz results can be stored and accessed without taking up much space at all.

# Task 3

## Planning

Task 3 is a much bigger challenge than Tasks 1 and 2, as I will need to create a fully functional administrator interface, through which teachers can access individual student’s data, as well as seeing their performance over time. Instead of including every requirement in a single page, I have chosen to create three new ‘tabs’ for administrators to use. Each tab will provide a different set of features, such as a results browser.

The new admin interface (data flow shown left) will need the following features:

* Results browsing page

Results sorting

By username (alphabetical)

By score

By date/time

By class

* User management page

User management

Creation

Deletion

Password changing

Recent (3) results

User sorting

By recent (3) average score

By total (all results) average score

Alphabetically

Class management page

Moving users between classes

Viewing users in classes

Class and Teacher roles

Classes 1-3

Teachers for administration page access

For the results page, I will need to query the database for results matching a set of criteria (class, username etc.) and then sort (by date/time, score etc.). I will also need to update my current quiz submission system to calculate user’s averages, both recently (using the past 3 results), and overall (all available results). This is more efficient that calculating the class/user’s average every time they need to load results, as that will be a lot more often than quiz submissions, especially with many users.

The Class system will need to be fully fleshed out, and I will also need to provide a way for the administrator/teacher to move users between classes. This will also be linked to the user management page.

Teachers will need to be able to create/delete/manage users, so a user management page will be needed. There it should list all the users on the system, their results/averages and group/class. Again, I will need to query the Users database and alter as necessary.

## Coding

## Result Browser

The results browser needs to give the teacher access to all results stored on the database. There will be 3 elements on the page – the search box, class filtering & sorting buttons, and the results themselves. Handlebars already includes built in functionality to iterate over an object and display data attributes from that object, so I need to pass an object containing all the data I want to display into Handlebars.

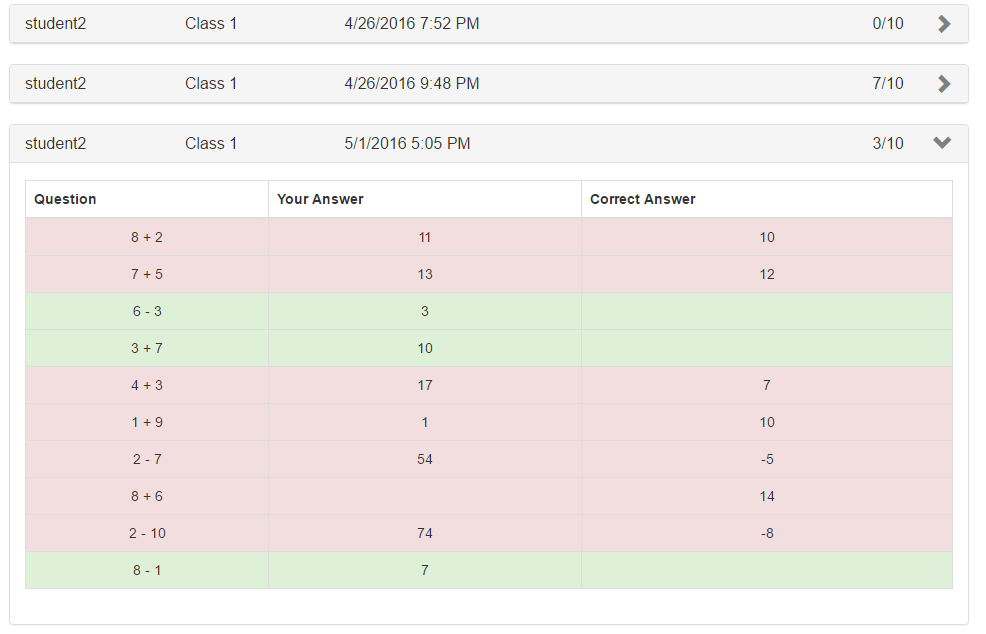
This is initially fairly simple – just use a blank query against the database to return every result stored, convert into a Handlebars readable format using the already made *convertResults()* function, and pass into the template. The helper is as follows:



The template itself is pictured below – using an if statement, I can check whether or not there are any results returned from the query – if not, I can tell the user that no results have been found. If results have been found, then I can create a loop to iterate over each result, creating a dropdown down box with information about the quiz in the title (date, student etc). Inside that dropdown box, I can use the *reactiveTable* package again to create a table for that result (same as the table shown at the end of the quiz).



The final result is shown below – each result has its own dropdown box which can be expanded to show more information about that quiz. This data is also updated automatically – new results will be automatically added to the page without the need to refresh.



Whilst I am able to get results, this page currently shows the results of all users at once, and doesn’t provide any way to sort or filter the results. To implement class filtering, I will need to go through each result, and remove any that the teacher does not want to see.

I added three toggle buttons to the top of the results page. Clicking each button updates a Session Boolean value accordingly. These can then be accessed in the *results* helper (the same one that returns the object containing all the results to the template) to remove results belonging to classes that the teacher does not want to see.

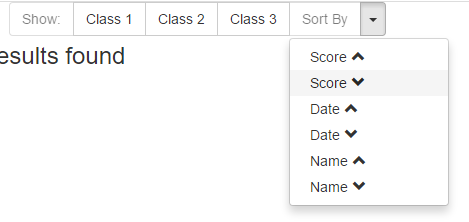
I added a for loop just before the results are given to Handlebars that will iterate (in reverse) through each result, compare the submitter of that quiz’s class to the *showClassX* Booleans controlled by the buttons, and *splice()* it from the object if the user does not want to see it. I have to iterate in reverse because if I didn’t, the *splice()* function would mess up the loop’s counter variable, *i (*there would be fewer objects in the array than expected).



**--Please see /GIFs/Tests/Task 3 – Test 1.gif for a recording of this test—**

The test shows that the filtering is functional – only members of the classes selected are being shown in the results browser! One problem with this solution is that, should there be an extremely large volume of quizzes needing to be filtered, the delay will increase as each result needs to be filtered. Given more time, filtering from Mongo would be much more effective and quicker.

Next, I need a way of sorting the results in a number of different ways. I first created a sorting dropdown box containing the possible sorting options:



For each sorting method, I included both Ascending and Descending options.

Unlike in the class filtering, MongoDB is sorting the results as they are being queried. Thusly, I will need to supply a set of options to the query for each method of sorting. Having hooked the dropdown box up to a set of Session Booleans, I can then use a *switch* to set the appropriate settings for each case:



This creates a *sort* object, containing the parameter *sort* (built in Mongo parameter) with the appropriate document to sort by (1 being ascending, -1 being descending).

These parameters can then be passed into Mongo during the *find()* query:



**--Please see /GIFs/Tests/Task 3 – Test 2.gif for a recording of this test—**

The test above shows that the sorting works well, allowing the user to easily sort the data in a number of ways.

I also wanted to include a way to only view a specific user(s) results. I could do this by taking advantage of a built in Mongo search criteria parameter. I added a search box to the results page, along with a clear button to remove any active filters.

Again, due to the nature of Meteor, I could have the results searched live, rather than requiring a ‘search’ button to reload all the results. After each keystroke, the filter is updated, and any changes shown. I also wanted a way to have the user enter multiple usernames to display the results of as many individual users as necessary. To do this, I added more logic to the *results* helper: 

This creates a new object (named criteria), and an array containing a list of usernames that we want to show. Having checked whether there is any input in the box, we can use a for loop to iterate over each comma delimited username (by first using *.split(‘,’)* to split into an array of usernames, then *trim()* to remove any whitespace). For each username, we push a parameter object into the *usernameArray.* Once the array contains the usernames we want, we can set the *.$or* attribute of criteria (This is just the format Mongo needs in order to apply the search criteria – see appendix 3.1 for more information).



The final query is above, containing both the list of usernames wanted, as well as how we want to sort the results retrieved.

**--Please see /GIFs/Tests/Task 3 – Test 3.gif for a recording of this test—**

As the above .gif shows, the username searching box is functioning as it should, and in conjunction with the class filtering.

The completed results browser now contains username search (shown above), class filtering, and result sorting.

### Class Management

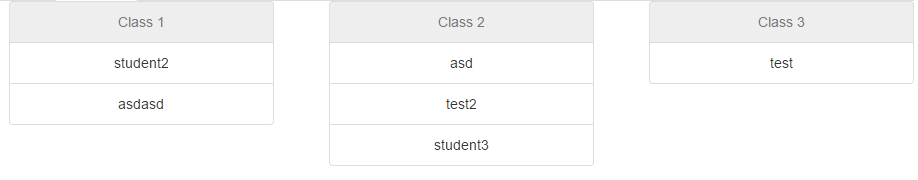
To improve ease of use, I wanted a way of visualising the three classes on the system, and easily moving students between them. I chose to try and implement some way of dragging users between classes. This makes group management intuitive and easy, and can instantly provide a top-down view of class size and the users within each class. Alternatives would be to have a dropdown box for each user to select which group they are in, but this is slow and confusing, especially when moving many users at once. To do this, I found a JavaScript library called *Sortable* which was perfect for what I needed.

First, I needed to create three lists, one for each class, with their respective users underneath. This was also fairly simple, using the same Handlebars loops used to list the results, but instead using the database of users. This created the three lists I wanted, along with a title (the class). The below excerpt was repeated for each class.



I used the Roles package (the same used for all of the class/group management, mentioned earlier) to return a list of users for each class.



Above is how the class lists looked after CSS styling. Now whilst I have a list of all the users, I am not able to drag them between classes yet. I had to activate the *Sortable* library on the page load, using the same *.rendered* callback used on the quiz page earlier. This uses jQuery notation to select all three lists (HTML classes *.sortable1 – 3)* and run the *.sortable* function. I also passed a few settings, namely *connectWith* which allowed items to be dragged between lists (classes) rather than just the order changed. I also need to specify that only items without the *.disabled* HTML classes should be draggable to prevent moving of the titles at the start of the lists.

Unfortunately, this didn’t work properly at first. After some further testing, I discovered that the *.rendered* callback was being called before Handlebars had fully inserted the lists of class members into the DOM (i.e. when the page had been rendered, but not the Handlebars objects). To rectify this, I needed to create a sub-template for each user being dragged.



This can then be called by Handlebars, passing the username as an argument:

****

I could then change the *.rendered* function for this page to be called after *userDrag* templates had been rendered, so the *sortable()* function was being applied properly.

**--Please see /GIFs/Tests/Task 3 – Test 4.gif for a recording of this test—**

As you can see, the dragging is functional, but as of yet, this is entirely visual. I still need to have these movements actually move users between classes at the backend. This was relatively simple – I expanded the *.sortable()* parameters to include introductions on what do to after the user has finished moving an object:

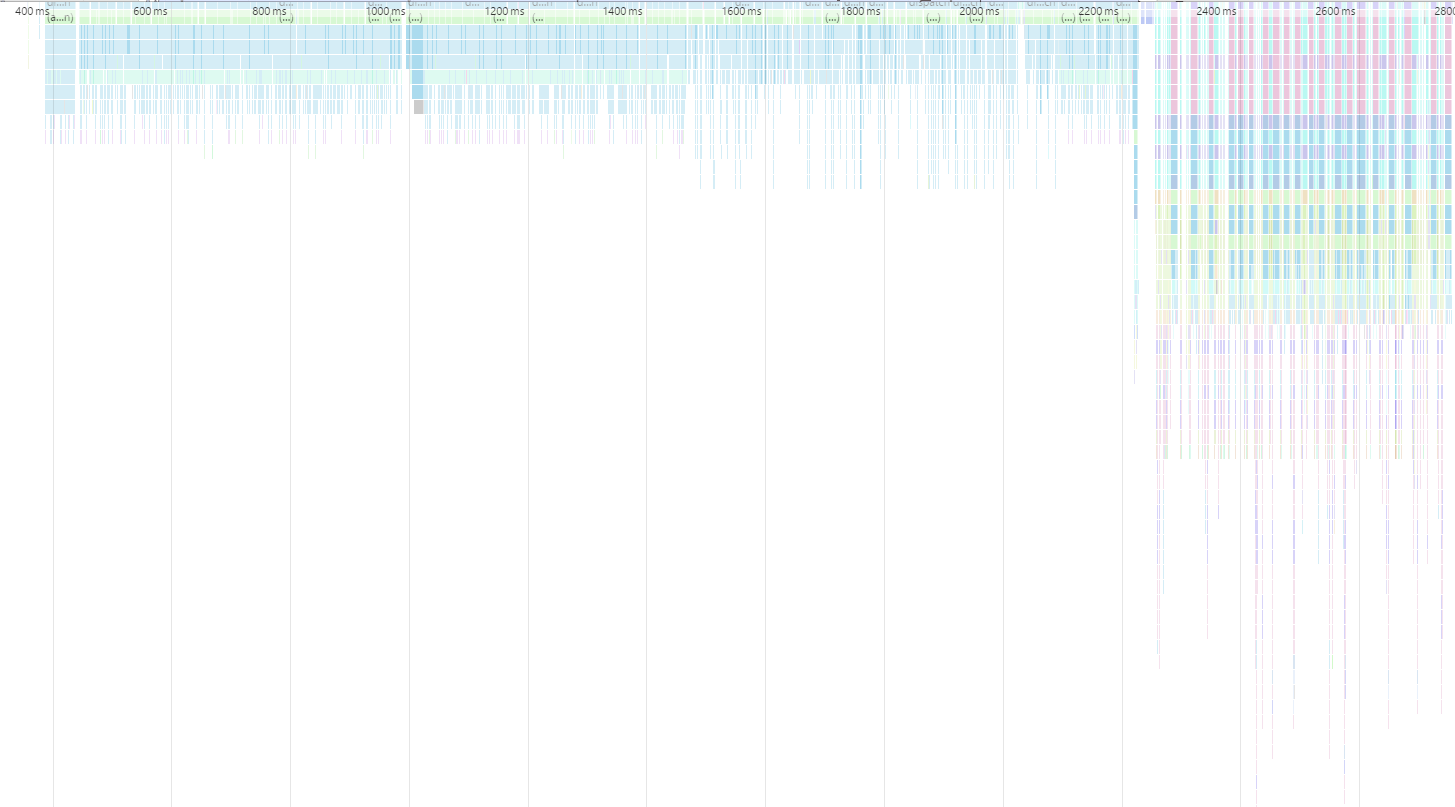


This called a server function (*moveUserToRole)* and passed the text of the object being moved (so the username), and the name of the list they were dragged to (so the class). I did this on the server to ensure it was non-blocking, and to decrease any wait time as much as possible. The server function is as follows:



This just calls a function provided by the *Roles* plugin to first remove the user given any groups they are a member of, and then add them to a different one. One problem with this method is that should it be provided with an invalid destination group, the user may be not be properly transferred, and left without a group. This is partially mediated later on in the user management section, which provides a hardcoded button to move a user to a specific group, and doesn’t rely on their current group to display them, so users of any and all groups will be shown.

To test this, I simply moved users between groups, and checked that they had been successfully transferred using *Robomongo* to manually inspect the user’s role.

Having used this repeatedly, I noticed that after each movement, it was becoming slower and slower. It sometimes took several seconds for a user to be moved. Having looked through my code, I can only conclude this is a problem with the *Sortable* plugin and how it interacts with Blaze/Handlebars. I used Chrome’s JavaScript CPU profiling tool to generate a timeline after moving a user:

Immediately noticeable is the series of pink spikes that coincide with a strange flashing of the user’s dragbox, recorded in the following gif:

**--Please see /GIFs/Tests/Task 3 – Test 5.gif for a recording of this test—**

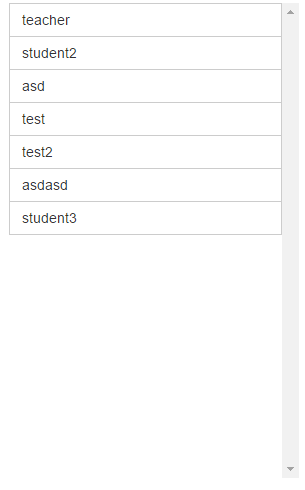
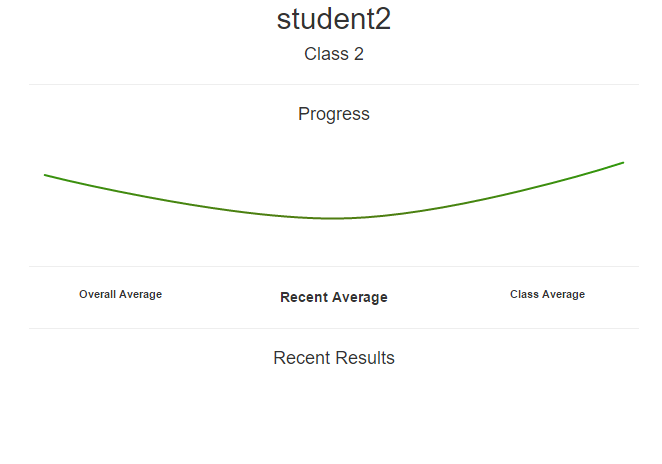
Examining each spike further, they appear to be comprised of 2 series of events – first, the websocket trigger called as the DOM changes, which in turn triggers Blaze to update, and ultimately run the *.rendered* callback. I tested this byhaving some texted logged to the console after each render, and found that it was indeed being called more and more after each drag (so the first time a user was dragged, it would be called only once, but after 5-10, it was being called 20+ times). This also explains the long delay, as this will be calling the *moveUserToRole()* function each time.

Although I had found the reason for the long delay and flashing effect, I was not able to find the root problem. Looking through the CPU profile, I believe it to be an issue with how Blaze and Sortable play off each other, but I lack the knowledge of either to properly investigate the issue in full.

The profile used has been saved to my project folder should you wish to view it yourself (*Chrome JavaScript CPU Profile (User dragging debugging).*

### User Management

Finally, I needed a way for the administrator to manage each individual user, as well as the creation and deletion of other accounts. I also needed to include a way of sorting these accounts by total average and recent average scores, as well as the cumulative average for the class to which they belong.

First, I used the same techniques previously documented to create a list of each user. Each username is its own button, so the administrator can click on whichever they want to view. CSS was also used to have it overflow into a scrollable list, rather than just extending the page.

I then set out the viewing pane with all the data I wanted to show, such as username and group. I also used a package called *Highcharts* to create small graphs that show the progress of a user over time (more on this later). I then created the necessary helpers on the client.

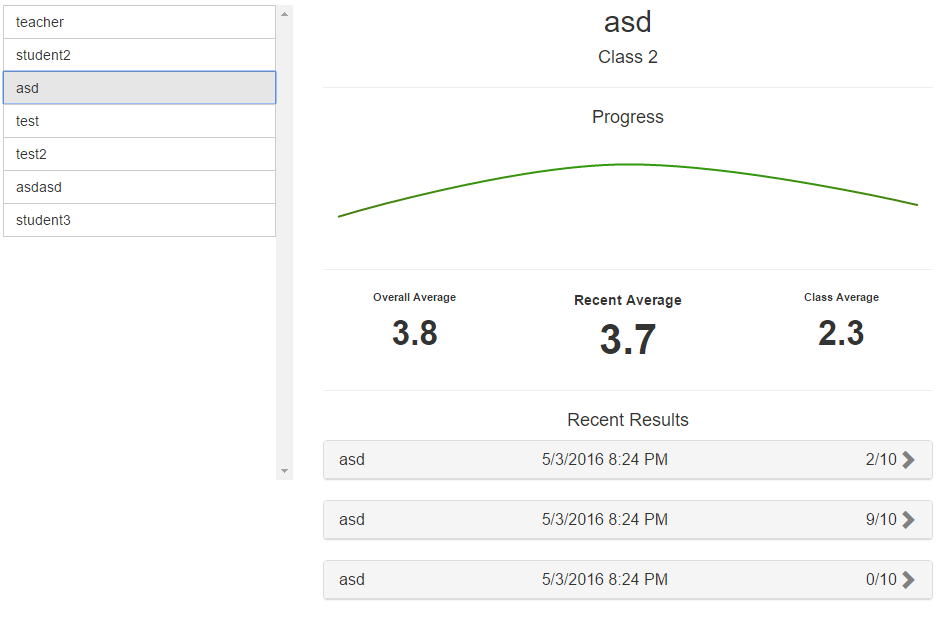
For the recent results, I used a Mongo query:



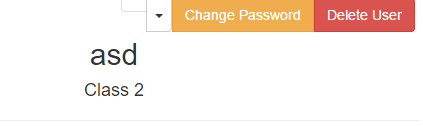
I queried using the username set by clicking on a specific user, and then a *limit:* option so as to only get the most recent 3 results. I also had to update the *submitAnswers()* server function to recalculate both the users total average, and recent average (past 3 results) after every quiz. 

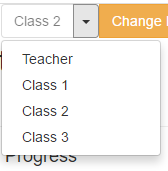
This is done twice – first for the total average (every result they have submitted) and then for the recent average (with the query limited to their past 3 scores).

Finally, all of the user information helpers are as follows:

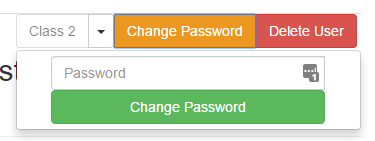


Pictured here is the fully populated user information pane. Whilst this is very useful to see how a user is doing compared to their class, as well as information about their past few tests, it does not let us sort or manage these accounts. To do this, we will need to add some buttons and hook them up to the backend.

I first added three buttons to the top right of the information pane. The first is a dropdown menu to manually move a user between groups without using the dragging system. This also includes a fourth group – teacher – if the administrator was adding another teacher. This would give the user access to the administration page.

Second, is the *Change Password* button. This allows a teacher to manually change the password of that user, should it have been forgotten, to something else. Finally, the *Delete User* button deletes that account, and all associated data. Currently none of these do anything, so let’s first set up the group movement dropdown.

First I added each group to the dropdown, as well as a unique ID that I can associate an event trigger with. I also reused a Handlebars helper previously created to show the current group on the button itself. In the JavaScript, I added 4 event triggers, one for each group, that would move the user to the selected group using the same *moveUserToRole* function created earlier.

For the *Change Password* button, I created a dropdown form with a textbox and a submission button. This functions similarly to the form used on the *quiz* page – using a standard HTML <form>, with an intercepting event in the JavaScript. I then created another server function *changeUserPassword* to update the password using the supplied password. 

The server function is fairly simple – all it needs to do is call the built in *Accounts.setPassword()* function with the given password, and the *logout* parameter set to false (otherwise if the user was changing the password of their current user they would’ve been logged out).



To test this, I changed the password of another account, logged out of mine, and attempted to log into that account. At first this didn’t work, and I realised that I had been setting the new password to a null value by fetching a *.value* of an incorrect HTML #id (Spelling mistake). Having corrected this, I could successfully change the password and login to another account.

The *Delete User* button was also quite simple to implement. I used a package called *BootstrapModalPrompt* to create a warning message to the user to confirm the deletion of the user. If yes is clicked, then the server function *deleteUser* is called.

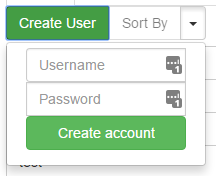


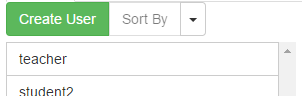
Shown below is the server function, including a check to ensure that at least one administration account(with username ‘admin’) remains on the system (so a user cannot lock themselves out).



**--Please see /GIFs/Tests/Task 3 – Test 5.gif for a recording of this test—**

The above .gif demonstrates the warning to the user, and we can see that the user has indeed been deleted (no longer present in the list of users generated in the user selection pane).

To complete the user management page, we need to include a way of creating new users, and sorting the list of users shown.

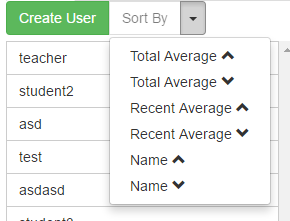
First, I created two more buttons, a *Create User* button, and another *Sort By* dropdown. The *Create User* button shows a dropdown similar to the *Change Password* button, but instead asks for the username and password of the new account. Here is the event called when this form is submitted:

And the associated server-side function:



**--Please see /GIFs/Tests/Task 3 – Test 6.gif for a recording of this test—**

We can see that the user creation function is working correctly, as the new user is immediately added to the user management pane. I have not included any data verification though, but usernames containing symbols should not pose a problem.

Finally, I am implementing the user sorting dropdown. This is similar to the one used on the results page, but includes slightly different options (recent/total averages). This is coded extremely similarly to the one on the results page.

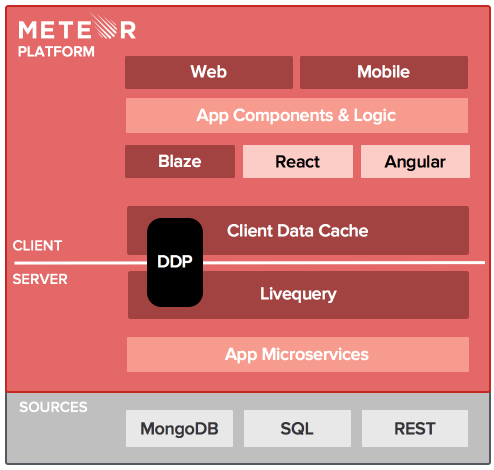
**--Please see /GIFs/Tests/Task 3 – Test 7.gif for a recording of this test—**

As the .gif shows, the list of users are being updated each time, and in the correct order.

## Evaluation

|  |  |  |
| --- | --- | --- |
| Criteria | Met? | Comments |
| “The teacher wants to use the results from students taking these quizzes to log their performance” | ✔ | All of their results are logged in the database. Students are also provided with a ‘Past Results’ screen, detailing their performance over time, and all of their past results. I have also chosen to include all results, rather than just the past 3. This allows for more in-depth analysis of performance and uses very little storage space. |
| “The system should store the last three scores for each student” | ✔ | All results are stored in the database, including the most recent three. I have also included the most recent three results on the information pane of the user management screen. Having used this frequently, I found that including a link to view the entire results given by that user rather than manually looking them up make it a bit more intuitive and useful. |
| “The teacher would like to be able to output the results of the quiz for a particular class...” | ✔ | The result browser contains buttons for filtering specific classes. Whilst this is very effective, I noticed that for large amounts of data, the splicing method detailed in that section can be quite slow. With further investigation I might be able to find a solution that filters at a database level, rather than once all the results have been loaded into memory. |
| Class result sorting: “Alphabetical order with each student’s highest score for the tests” | ✔ | The result browser allows for class filtering, and then sorting of scores from highest to lowest and alphabetically, though does not show only the highest score. Username filtering would allow the teacher to then view that user’s scores and sort by score highest to lowest. |
| Class result sorting: “Highest score, highest to lowest” | ✔ | The result browser provides sorting options for both highest to lowest and lowest to highest score sorting. |
| Class result sorting: “Average score, highest to lowest” | ✔ | The user management pane includes sorting by both recent (past 3 tests) average, and total average (over all tests). As mentioned above, the ease of use could be improved by providing a direct link to the selected user’s full results. The performance graph given on that page is also not very useful at the moment, and would be further improved if given an option to select how many points to plot (e.g. past week, past month etc). |

# Appendix

1. 
   1. Meteor is a JavaScript Application stack built upon a number of technologies. At its heart, Meteor builds upon other Node.js applications, making it easier to create full client/server applications quickly and easily. Although I could have used plain Node.js (a JavaScript engine for building event-driven apps), Meteor includes a number of useful features and frameworks, such as Blaze (Handlebars) and a full package manager. An illustration of the Meteor stack is shown (right).

<https://www.meteor.com>

1. 1. Much of the simplicity gained from using Mongo comes from the use of JSON in queries – this makes organisation and readability much better. For example, if I had a list of contacts through which I wanted to return all contacts with at least one ‘work phone’ given:

SQL:

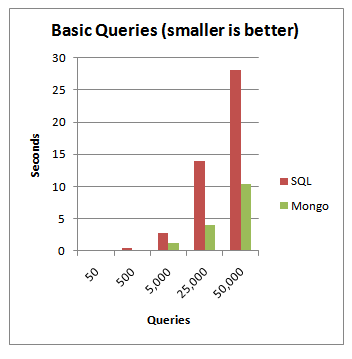
*Select \* from contact A, phones B where A.did = B.did and B.type = ‘work’;*

Mongo:

*db.contact.find({“phones.type”:”work”});*

As you can see, the Mongo query is much more readable, and should I want to add another element to search by, all I need to do is add a comma and another JSON formatted string.

Another aspect of Mongo is its high performance compared to traditional SQL servers – a simple query over 50, 500, 5,000, 25,000 and 50,000 objects is shown below.



Although during testing I am not reaching anything like these numbers of objects, over time, it is not unrealistic to expect a school to achieve 500 – 1,000 quiz results stored across hundreds of students.

1. 1. Along with the query text, Mongo provides a series of operators that can also be set to allow for further result specification. The $or operator acts as the logical OR operation, allowing multiple query objects to be used in conjunction. Each query object is provided as a member of an array ( [{query 1}, {query 2}] ).

<https://docs.mongodb.com/manual/reference/operator/query/>