**Programming Project**

# 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 is 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.

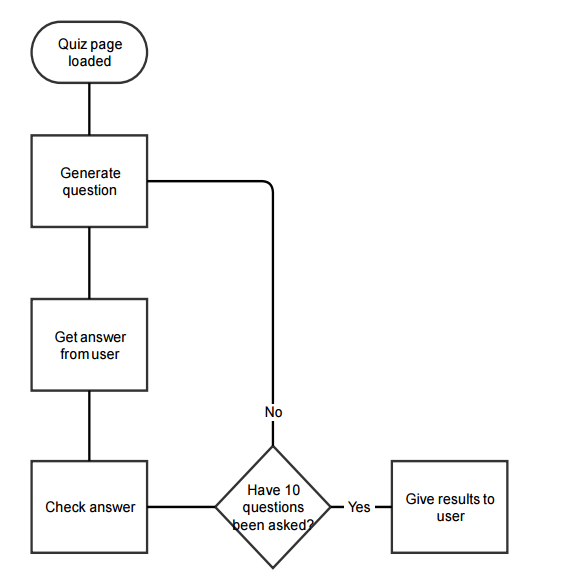
Because the teacher wants to use this system to manage a class of students, a simple account system would be useful to keep track of individual student’s performance over time, as well as letting students submit work in their name from anywhere. Meteor includes a full account management system built 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.

For more information on the Meteor Stack, see <https://www.meteor.com/>

### 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 (left) 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.
* Get the student’s name
* Give feedback on their answers to the question
* Score the quiz out of 10.

## 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.



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. 

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. 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 *addToAnwerLog()* (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.



The *nextQuestion()* function is also fairly simple, but runs on the server rather than the client. This means it will be run asynchronously, so I will need to include a callback in the Server call. I can also log any errors to the console at the same time. After a question has been returned from the Server, I can iterate the question counter, and store the new question in the Session, where the helpers (described below) will display it to the user. 

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.

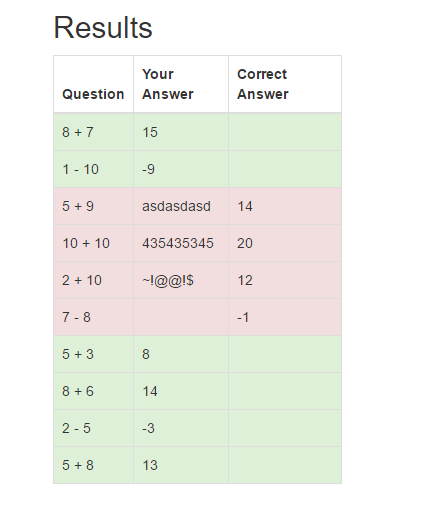


Once the quiz has been completed, we need to give the results to the user. To do this, they will need to be redirected to the results page (*‘/results’),* where we can use the Session key/value store to retrieve the *answerLog* arraystored by the quiz. 

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 blah



## 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/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 why – the REGEX to check the input only checks for letters, 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. Having fixed this, I reran the test to ensure that everything worked as it should.

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