# Testing Documentation

# Methodology

To test our web app, we will be using a range of techniques firstly we will undertake unit testing both static and dynamic. In static unit tests we will give a commentary on the unit and what it should perform and what it’s required to do. In dynamic tests we will write automated tests that we can run to see for inputs of all cases especially edge and erroneous ones that the correct result is given by the system. This allows us to follow a test-driven development methodology of completing a unit then testing it meaning we can re-run tests as we add more and more functionality to check the old units still pass.

We will then move on to testing the system as a whole by doing a combination of integration tests to automate tests of combined units mostly in the back-end which are interconnected. As well as manual black box testing the whole system so we can test how user interfaces also interact with the code.

In black box testing we will ask people outside of our team as well as ourselves to perform specific tasks acting as a user with no knowledge of the web apps actual code. Such as to try to login unsuccessfully, then register an account and play a game all in one session to see how it would respond with all the units integrated together with natural user inputs. We will also at this stage be able to test front-end validation error messages for example if you try to make an account but that username already exists it sends the user a friendly alert to try again and not a stack trace. This will allow us to test the system as a whole as well as the user interfaces.

## Testing Tools

|  |  |  |
| --- | --- | --- |
| Tool | Use | Reference |
| Jest | Used to test all front-end and back-end units. Provides methods to test units and expect certain results. | Jestjs.io. n.d. Getting Started · Jest. [online] Available at: <https://jestjs.io/docs/en/getting-started> [Accessed 19 January 2021]. |
| Vue-Test-Utils | Used to test the Vue files in conjunction with jest in the front-end. It allows you to mimic Vue files using a wrapper. | Vue-test-utils.vuejs.org. n.d. Introduction | Vue Test Utils. [online] Available at: <https://vue-test-utils.vuejs.org/> [Accessed 19 January 2021]. |

## Devices Tested On

Device: Apple MacBook Pro 15

Display resolution: 2880 x 1800, 16:10 ratio

Operating system: macOS Catalina

Processor: 2.6 GHz 6-Core Intel Core i7

Memory: 16 GB 2400 MHz DDR4

Device: Apple MacBook Pro 13

Display resolution: 2560 x 1600 pixels

Operating System: macOS Catalina

Processor:

Memory: 8 GB 2133 MHz LPDDR3

Device: Lenovo Y50-70

Display resolution: 1920 x 1080 pixels

Operating system: Windows 10 Home

Processor: 2.6 GHz Intel Core i7

Memory: 8 GB 1600 MHz DDR3

Device: ASUS Vivobook 15

Display resolution: 1920 x 1080 16:9 ratio

Operating system: Microsoft Windows 10 Home

Processor: AMD Ryzen 5 3500U with Radeon Vega

Memory: 8GB: 2400 MHz DDR4 4G+4G

# Unit Tests

## Back-end

### Hash unit (hash.js)

This unit is where users’ passwords are salted and hashed. The hash algorithm in use is SHA-256 and the salt is an SHA-256 hash of the username. The salt is added to the end of the password and the whole thing is hashed.

Static Testing

The code for this method is required to generate a salt and to combine it with the password to then hash. The method in reality takes an input of a username and password combination. It then generates a salt. It does this using the crypto library to hash the username using SHA-256. It will then generate a hash again using the crypto library. It then hashes the password with the salt added on the end using SHA-256. It then returns the completed hash. Meeting what it is required to do.

Dynamic Testing

The dynamic tests for this unit will test for a variety of accepted input sizes the correct hash is made. Remember it first hashes the username to make a salt then adds that to the end of the password and hashes that new combination together all using sha-256. These tests can be ran again using hash.test.js

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected outcome | Actual outcome |
| 1 | Testing a standard input within all bounds | In Range | Username:  ethan  Password:  pass | 832f6101280c3d3fc3529c172b98421347a4e01965a26540ba46b469d268dafe | 832f6101280c3d3fc3529c172b98421347a4e01965a26540ba46b469d268dafe (Pass) |
| 2 | Testing smallest input username (1 char) and password (6 char) | Edge | Username: a  Password aaaaaa | 624407e6be53a4833e7b674db32660f1cf1a97d20d9f09f227868a523d86b4d4 | 624407e6be53a4833e7b674db32660f1cf1a97d20d9f09f227868a523d86b4d4  (Pass) |
| 3 | Hashing the largest input for password (12 chars) with smallest username (1 char) | Edge | Username: a  Password: aaaaaaaaaaaa | fab3dffd86640104e63f0a24ae4bffead6fe04af38d394d8f503e995e75ff58a | fab3dffd86640104e63f0a24ae4bffead6fe04af38d394d8f503e995e75ff58a  (Pass) |
| 4 | Hashing the largest input for username (10 chars) and password (12 chars) | Edge | Username: aaaaaaaaaa  Password: aaaaaaaaaaaa | 2438404c4981e3fd05aff6ac175d2fd066711acad631988e2de0a9edcc82b1e1 | 2438404c4981e3fd05aff6ac175d2fd066711acad631988e2de0a9edcc82b1e1  (Pass) |
| 5 | Hashing the largest input for username (10 chars) and smallest password (6 chars) | Edge | Username:  tenChars10  Password:  sixCha | 637bcb8521628ba618fe64a78b953a06d1aef3a8b5d9afb361cbeca5577bb794 | 637bcb8521628ba618fe64a78b953a06d1aef3a8b5d9afb361cbeca5577bb794  (Pass) |
| 6 | Hashing an input with special characters and numbers | In Range | Username:  Sp3cial!  Password:  Ch£rs99 | 7e58cc25020d23fb9212a7e5affa7e5d49749490b0bab053370e197bd58adafe | 7e58cc25020d23fb9212a7e5affa7e5d49749490b0bab053370e197bd58adafe  (Pass) |
| 7 | Form validation should not allow values to be empty but if they are the hash sould still work | Erroneous | Username: ‘’  Password: ‘’ | cd372fb85148700fa88095e3492d3f9f5beb43e555e5ff26d95f5a6adc36f8e6 | cd372fb85148700fa88095e3492d3f9f5beb43e555e5ff26d95f5a6adc36f8e6  (Pass) |
| 8 | Form validation should not allow values too long but if they are the hash should still work | Erroneous | Username:  LongUsernameLong  Password:  LongPasswordLong | ecb5e7c545f697d3fb1ef1852cabe85edf9c74dfce6735308c3acca5d6238e72 | ecb5e7c545f697d3fb1ef1852cabe85edf9c74dfce6735308c3acca5d6238e72  (Pass) |

### usernameExists.js

This unit is responsible for checking if a specific username exists or not in the mongoDB database. This is for during the registration period as we want users to all have unique usernames.

Static Testing

This method is required to check if a username exists so that we don’t have 2 users with the same username. The code for this method is an async function which takes the input of a string of the username to check. Async functions allow us to use await to allow promises to be resolved. Firstly, we make a connection to the database using a promise and the database URI. The promise is resolve once it reaches the then() at this point we are connected to the database. Once connected we assign a variable to a new promise and await its completion this new promise is our query. The query is a findOne() in the database taking the input of the username to find. If it finds it returns ‘exists’ if it doesn’t find a document with that username it returns null. The method is invoked in the frontend during registration using an Axios post request to index.js which calls the function and sends the result of it back to the frontend using a response. The frontend will alert the user if it returns ‘exists’ and not null to prevent the next step of registration adding their data to the database.

Dynamic Testing

The dynamic test will test for a variety of input sizes and character types too see if that username exists in the database or not. These tests can be ran using usernameExists.test.js

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Testing a normal username within bounds that is already in the database | In Range | Username: testing | ‘exists’ | ‘exists’  (Pass) |
| 2 | Testing a normal username within bounds returns null as is not in the database | In Range | Username: UNIQUE | null | null  (Pass) |
| 3 | Testing a username passed exists of the smallest username (1 chars) | Edge | Username: T | ‘exists’ | ‘exists’  (Pass) |
| 4 | Testing an existing username passed to the function returns exists of the largest username (10 chars) | Edge | Username: 10CharExis | ‘exists’ | ‘exists’  (Pass) |
| 5 | Testing a new unique username passed to the function returns null of smallest username (1 chars) | Edge | Username: U | null | null  (Pass) |
| 6 | Testing a new unique username passed to the function returns null of the largest username (10 chars) | Edge | Usename: 10CharUniq | null | null  (Pass) |

### loginAuthentication.js

This unit is part of the backend and authenticates user logins with the database. It takes parameters of the user’s username and hashed password. It takes these 2 parameters and checks for a matching result. It will then return the database entry that matches those credentials or null if there is no match.

Static Testing

The requirements for this method are to authenticate valid user logins and to highlight invalid logins. This method is an async function meaning we can use await to allow promises to resolve before trying to retrieve values of variables. It takes parameters of the user’s username and hashed password. It then using the database URI makes a connection to the database in a promise and resolves that promise using a then statement. We then assign a variable to a promise query to query the database for that username and hashed password combination. It uses a findOne query to do this as any usernames in the data should be unique as well as their hashed password which was also salted. The promise is resolved by returning the result. If it was a successful login it will return the document in the database with those parameters if not it returns null. This method is called by the front-end sending a login request to the back-end using an Axios post request which is picked up by index.js which will call this method once hashing and salting the password. Index.js will return to the front-end the result of loginAuthentication.js using a response. The front-end using this result will either allow the user to proceed or ask them to try again.

Dynamic Testing

This method takes the input of the username and hashed password. It will return a matching document in the database or null if one does not exist. The hashes are sha-256 so are always of the length 64 characters. These tests can be ran using loginAuthentication.test.js

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Testing a valid username and hash | In Range | Username:  LoginTest  Password:  f6179ed4db729720155d82b03b033f27bbb4b64e3e9651a1b1dce6a7675fb78b | Returned document where username is LoginTest | Returned document where username is LoginTest  (Pass) |
| 2 | Testing a valid username and invalid hash | Erroneous | Username:  LoginTest  Password:  incorrectb729720155d82b03b033f27bbb4b64e3e9651a1b1dce6a7675fb78b | null | null  (Pass) |
| 3 | Testing an invalid username and valid hash for another account | Erroneous | Username:  InvalidLoginTest  Password:  f6179ed4db729720155d82b03b033f27bbb4b64e3e9651a1b1dce6a7675fb78b | null | null  (Pass) |
| 4 | Testing an invalid username and hash | Erroneous | Username:  FailedLogin  Password: b8179ed4db729720155d82b03b033f27bbb4b64e3e9651a1b1dce6a7675fb78b | null | null  (Pass) |
| 5 | Testing an valid username and hash with username of shortest length (1 char) | Edge | Username: T  Password: 4845ee18f0cfd3109768026bf1cde9f06f2488b59bf2743bdbe0d0112e641c12 | Returns document where username is T | Returns document where username is T  (Pass) |
| 6 | Testing an valid username and hash with username of the longest length (10 chars) | Edge | Username: 10CharExis  Password: e5938d4780f861f2fab317d6d1ee1ff1fd150e91740681027fc2b4334c605e2f | Returns document where username is 10CharExis | Returns document where username is 10CharExis  (Pass) |
| 7 | Testing an valid username and but invalid hash with username of shortest length | Edge and Erroneous | Username: T  Password: | null | null  (Pass) |
| 8 | Testing an valid username and hash with username of the longest length | Edge and Erroneous | Username: 10CharExis  Password: | null | null  (Pass) |

## register.js

This unit in the back-end is responsible for adding new users to the database. It takes the parameter of the user account made earlier by a constructor. This object is inserted into the database as a document.

Static Testing

This unit is required to add new user details into the database to be used by other methods such as ones for logging in. The method is an async function so we can use await to allow promises to resolve so we don’t try and user the returned values before the promise resolves. It takes the parameter of an object this object is the user account to add. It is made using a constructor before calling this method. The first part of the method is using the database connection URI to connect to the database using a promise which once resolved allows us to begin database transactions. Once connected we set a variable to a query promise. The query uses the save() method which inserts a document into the database. This promise resolves by returning the document added to the database. This variable is returned by the method so the back-end can send a response to the front-end alerting them it was successful.

Dynamic Testing

The dynamic testing of this method will test different input sizes and character types are added to the database correctly when creating a new user. Note that the password file is a salted sha-256 hash so will always be 64 characters. You can run these tests using register.test.js

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Testing inserting an account | In Range | \_id: new mongoose.Types.ObjectId(),  firstname: 'Unit',  lastname: 'Test',  username: 'unitTestAccount',  password: 'fab3dffd86640104e63f0a24ae4bffead6fe04af38d394d8f503e995e75ff58a',  email: 'testing@test.com' | Matching document is returned after being added to database | Matching document is returned after being added to database  (Pass) |
| 2 | Testing inserting the smallest values possible accepted by the form | Edge | \_id: new mongoose.Types.ObjectId(),  firstname: 'U',  lastname: 'T',  username: 'u',  password: 'fab3dffd86640104e63f0a24ae4bffead6fe04af38d394d8f503e995e75ff58a',  email: 't@t' | Matching document is returned after being added to database | Matching document is returned after being added to database  (Pass) |
| 3 | Testing inserting the largest username (10 chars) value possible accepted by the form | Edge | \_id: new mongoose.Types.ObjectId(),  firstname: 'IHaveALongFirstName',  lastname: 'IHaveALongerLastName',  username: '10CharsAdd',  password: 'fab3dffd86640104e63f0a24ae4bffead6fe04af38d394d8f503e995e75ff58a',  email: 'longUsername@testLong.com' | Matching document is returned after being added to database | Matching document is returned after being added to database  (Pass) |
| 4 | Testing inputs with special characters | In Range | \_id: new mongoose.Types.ObjectId(),  firstname: 'Test!',  lastname: '?tester',  username: 'SpecialCha£s',  password: 'fab3dffd86640104e63f0a24ae4bffead6fe04af38d394d8f503e995e75ff58a',  email: 'specialChars\*@testSpecial.com' | Matching document is returned after being added to database | Matching document is returned after being added to database  (Pass) |
| 5 | Testing inputs with numeric characters | In Range | \_id: new mongoose.Types.ObjectId(),  firstname: 'Test1',  lastname: '2tester',  username: '3',  password: 'fab3dffd86640104e63f0a24ae4bffead6fe04af38d394d8f503e995e75ff58a',  email: '35@35.com' | Matching document is returned after being added to database | Matching document is returned after being added to database  (Pass) |

### updateScore.js

This unit in the back-end is responsible for updating or creating documents that hold user scores with new high scores so that another unit can make an up-to-date leader board.

Static Testing

This unit is required to update a user’s current high score in the database and if one doesn’t exist add one. The method is an async function so we can use await to allow promises to resolve. The method takes as a parameter an object made with the Scores constructor with the details of the score achieved, the users username, the game it was achieved in and what stream the game was testing. The first part of the method uses a promise to connect to the database using the URI. Once resolved it sets a variable to a new query. This query finds a document with the same username, stream and game details if one doesn’t exist a document is created with those details as we have set ‘upsert’ to true. The next part of the query updates the score only if its higher that the one currently in the document. The resulting document from this operation is returned by the function.

Dynamic Testing

The dynamic unit tests will test that when the function is called to update scores if a previous score does not exist one is created. Then if one does its only updated if the score is higher. You can see and run these tests using updateScore.test.js

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Testing inserting a new score | In Range | username: 'TestCase',  game: 'blocks',  stream: 'Software Testing',  score: 80 | A new document is added with the input objects attributes | A new document is added with the input objects attributes  (Pass) |
| 2 | Testing inserting a lower score than the one in the current document | In Range | username: 'TestCase',  game: 'blocks',  stream: 'Software Testing',  score: 50 | The object currently in the database is not updated as the score is lower 50<80 | The object currently in the database is not updated as the score is lower 50<80. The score of the returned document is 80  (Pass) |
| 3 | Testing inserting a higher score than the one in the current document | In Range | username: 'HighTest',  game: 'blocks',  stream: 'Software Testing',  score: 100 | The object currently in the database is updated as the score is higher | The object currently in the database is updated as the score is higher. The score of the returned document is 100  (Pass) |
| 4 | Testing error score of null does not overwrite other scores | Erroneous | username: 'TestCase',  game: 'blocks',  stream: 'Software Testing',  score: null | The document returned keeps its original score 80 and is not replaced with null | The document returned keeps its original score 80 and is not replaced with null  (Pass) |

### retrieveTest.js

This method is called when the front-end requires the affinity test questions of a specific stream.

Static Testing

This unit is required for a given input stream as a string return the affinity questions for that stream. It does this by taking the stream as an input connecting to the database. Then once connected it queries the database for questions of that stream inputted and returns all matches so they can be used as questions in the front-end.

Dynamic Testing

Testing for the 3 valid input streams and invalid input the correct documents are returned.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Testing valid input of Software Testing | In Range | ‘Software Testing’ | An array of documents with the stream being Software Testing | Array of documents of Software testing stream  (Pass) |
| 2 | Testing valid input of Technical Operations | In Range | ‘Technical Operations’ | An array of documents with the stream being Technical Operations | Array of documents of Technical Operations stream  (Pass) |
| 3 | Testing valid input of Business Intelligence | In Range | ‘Business Intelligence’ | An array of documents with the stream being Business Intelligence | Array of documents of Business Intelligence stream  (Pass) |
| 4 | Testing an invalid input for stream | Erroneous | ‘Not a stream’ | No documents returned | No documents were returned  (Pass) |

## Front-end

### TextBlocks.vue

This is one of our games where users are required to put blocks of text back into order to match a definition.

Static Testing

This unit is responsible for the html and game logic for the text blocks game. It is made up of 3 parts the Vue template html code, the JavaScript code and the css. The html template first defines the navigation bar which has the FDM logo and astra as well as the links to other pages. There is then the title, reset icon and time remaining. Below that is the game container. Within that is the html code for displaying the blocks that are all draggable.

In the script section is the JavaScript code it imports axios to retrieve questions and add scores. It also imports vue-draggable to make draggable components and the vue router so we can route to other pages. In the data function the variables for the correct answer, the current block order and the hint that can be used in the html and the script. There is then the mounted function which runs when the page is loaded and sets the variables answer, question and hint by using axios to communicate to the back-end to get them. In the methods section is the method which is invoked by the html every time an element is dragged it checks if the dragged blocks are in order or not if they are the game is complete.

Dynamic Testing

The dynamic tests for this unit check for various orders of text blocks and inputs including erroneous ones that the correct result is generated. The variable gameFinished is either if you have completed the game is turned to true or it remains false. This method for checking would usually be called once a block is dragged and dropped but in the testing environment we simulate by predefining the order they are dragged into and call the checker() function which checks the order.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Blocks in the correct order | In Range | blockOrder = [1, 2, 3, 4, 5]  correct = [1, 2, 3, 4, 5]  gameFinished = false | Game will be completed so  gameFinished = true | gameFinished = false  (Pass) |
| 2 | All blocks in the incorrect order | In Range | blockOrder = [2, 1, 4, 5, 3]  correct = [1, 2, 3, 4, 5]  gameFinished = false | Game will not be completed so  gameFinished = false | gameFinished = false  (Pass) |
| 3 | Blocks in partial correct order | In Range | blockOrder = [1, 2, 4, 5, 3]  correct = [1, 2, 3, 4, 5]  gameFinished = false | Game will not be completed so  gameFinished = false | gameFinished = false  (Pass) |
| 4 | Block order is empty making it an erroneous input | Erroneous | blockOrder = []  correct = [1, 2, 3, 4, 5]  gameFinished = false | Game will not be completed so  gameFinished = false | gameFinished = false  (Pass) |

### Login.vue

Static Testing

The file is required to take the users input in the form and send it to the back-end to authenticate. Depending on the back-ends response the user is logged in or told the credentials are incorrect.

The html section of the file is made up of the login form as well as the links to other pages such as registration, admin login and guest login. Users will enter their username and password and submit it to be validated.

The script section is made up of the 2 methods loginPost() and validUser(). On submission of the login form loginPost() is called it makes an axios post request to the back-end with the username and password. The back-end will authenticate the login for us. The response of this post request is used by validUser() to check the returned data is valid. It will either be null for an invalid login or the JSON document of that user from the database minus their hashed password.

For more details on testing the authentication of the user login in the back-end which tests username and password combinations see the section on loginAuthentication.js

Dynamic Testing

The dynamic tests firstly check loginPost() posts the username and password in the form to the back-end. As well as if validUser() which takes the response from the back-end correctly identifies valid and invalid users.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Axios Request made on submission  (loginPost()) | In Range | The html wrapper with the form in it | Axios request calls from the form the username and password using document.getElementId | Axios request is made and username and password are called from the form  (Pass) |
| 2 | Request returns valid user so they can proceed and no alert is called  (validUser()) | In Range | response = {  data: {  username: 'testing',  firstname: 'first',  lastname: 'last',  email: 'mail@test'  }  } | Alert not called as it’s a valid user | Alert was not called  (Pass) |
| 3 | Request returns invalid user so they cannot proceed so an alert is called  (validUser()) | Erroneous | response = {  data: {  }  } | Alert is called/displayed as its an invalid user login | Alert was called and displayed  (Pass) |

### Register.vue

Static Testing

This unit is required to take the user input using a form and send it to the back-end to check if the username is unique then if so add it to the database.

The html section of the Vue file includes this form with the necessary front-end validation requiring each field to not be empty. As well as specific username and password lengths.

The JavaScript section includes the methods for posting to the back-end to see if a username is unique and not in the external database. If it is not unique the next function will alert the user. If it is unique that function will post to the back-end for it to add the new user details and hash the password in the external database. Once posted the user is routed to the next page. For more information on the related back-end methods see the tests for hash.js, usernameExists.js and register.js

Dynamic Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Axios Request made to check if username exists by usernameUniquePost | In Range | The html wrapper with the form in it | Axios request calls from the form the username using document.getElementId | Axios request is made and the username is called from the form  (Pass) |
| 2 | Alert is called if back-end responds the username is not unique | Erroneous | response = {  data: ‘exists’} | Alert is displayed as the username already exists | Alert was called  (Pass) |
| 3 | Alert is not called if back-end responds the username is unique and an axios request is made to add the new users data | In Range | response = {  data: {  }  } | Alert is not called as its an valid user name so axios request is made that gets 5 elements from the form | Alert was called and axios gets the 5 elements from the form  (Pass) |

### AdminLogin.vue

Static Testing

This file is required to send a user input to the back-end for authentication and based on the back-ends response allow or deny the login.

The html section of the file is made up of the admin login form as well as the links to the user login page. Admin’s will enter their username and password and submit it to be validated on submission of the form.

The JavaScript part of the code deals with the submission of details to the back-end and routing the user to the next page if it’s a valid user. The first method adminPost() sends an axios post request to the back-end to validate the submitted form data. It returns the response of the back-end to validAdmin(). The admin login is valid if the returned document has the data value admin to be true and they are taken to the next page. If not it is invalid and the user receives an alert.

Dynamic Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Test that adminPost() calls the username and password from the form to be used by axios | In Range | The html wrapper with the form in it | Axios request is made and calls the username and password from the form | Axios request was made and calls the username and password from the form  (Pass) |
| 2 | Testing alert is not called and the admin is routed to next page when back-end returns it’s a valid admin | In Range | response = {  data: {  admin: ‘true’  }  } | The alert should not be called as they are a valid admin | Alert was not called  (Pass) |
| 3 | Testing alert is called when back-end does not return they are an admin | Erroneous | response = {  data: {  }  } | Alert should be called and displayed as they are not a valid admin | Alert was called and displayed  (Pass) |

### AffinityTest.vue

This unit is responsible for being the page in which users can answer questions to see if they would be a good fit for a specific stream.

Static Testing

The file is required to generate the page and logic of affinity tests for each stream. The question and the possible answers are displayed on clicking one of the options a function is run to see if it was the correct or bonus answer and accumulates points. It will then move on to the next question if there’s no questions remaining then the affinity test ends and the user is told if they fit that stream.

Dynamic Testing

Each time an option is selected optionSelected() is called which updates affinity and will move to the next question if there is one otherwise it will end the test.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | 2 points added if correct option is selected | In range | The html wrapper with the question defining the correct option.  Selecting correct option | Affinity is increased by 2 points | Affinity was increased by 2 points  (Pass) |
| 2 | 1 point added if bonus option is selected | In range | The html wrapper with the question defining the bonus option.  Selecting bonus option | Affinity increased by 1 point | Affinity increased by 1 point  (Pass) |
| 3 | 0 points added if the incorrect option is selected | In range |  | Affinity remains the same | Affinity remains the same  (Pass) |
| 4 | Game ends when no questions remaining | In range | quizComplete: false  3 questions answered 1 by 1 | quizComplete= true only after the third question | It turned true after the third question  (Pass) |

### GameTimer.vue

The game timer is used as a 2-minute countdown within the game components. Its used to make sure games end once 2 minutes are used as well as to calculate scores.

Static Testing

This file is required to be a 2-minute countdown which we can retrieve the current time remaining from.

The html template section displays the time in minutes and seconds which is what will be seen when it is imported into the other games. The script section defines all its global variables in the data function them being related to the time remainins, minutes remaining, seconds remaining and the variable gameTimer which we use to start and stop the timer. There is 4 methods used startTimer which resets the timer variables and sets an interval of 1 second to a function called timeMonitor. This function runs every second and decreases the time left in minutes and seconds until it reaches 0. There is also a function to stop the timer stopTimer this clears the interval stopping the countdown. Then there is another method getTime which returns the timeRemaining usually used by the games to calculate scores.

Dynamic Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Test that time decreases correctly | In Range | timeRemaining =120 | timeRemaining =119  minutesRemaining =1  secondsRemaining =59 | timeRemaining =119  minutesRemaining =1  secondsRemaining =59  (Pass) |
| 2 | Test that when seconds remaining is under 10 its displayed with a leading 0 | In Range | timeRemaining =70 | secondsremaining = 09  (70 – 1 = 69 = 1min 09 seconds) | secondsremaining = 09  (Pass) |
| 3 | When time left is 0 time no longer decreases | Edge | timeRemaining =0 | timeMonitor will not decrease time  timeRemaining =0 | timeRemaining =0  (Pass) |
| 4 | If time was somehow negative it should not decrease in this erroneous state as its less than 0 | Erroneous | timeRemaining =-1 | timeMonitor will not decrease time  timeRemaining =-1 | timeRemaining =-1  (Pass) |

# Back-end Integration tests

There are 3 main integrated components in the back-end they are the hash.js integrated with register.js and hash.js integrated with login.js.

### hash.js integrated with register.js

In the database we don’t want to store users passwords in plain text so we hash them before we add them in the database. So the flow of components to register a new account is to first hash the password, add that to a new instance of the UserAccount object and then to use register.js to add this as a document in the database

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Hash and register a new user within bounds of the form input | In range | Username:  IntName  Password:  IntTest  newUser: {  username:  IntName  Password:  Result of hash  Firstname:  Int  Lastname:  Test  Email:  testing@test  } | Document of the new user is returned and the details all match (excluding password which is removed from the result) | Document of the new user is returned and the details all match  (Pass) |
| 2 | Hash and register a new user of smallest size form input for username and password | Edge | Username:  I  Password:  IntTes  newUser: {  username:  I  Password:  Result of hash  Firstname:  Int  Lastname:  Test  Email:  testing@test  } | Document of the new user is returned and the details all match (excluding password which is removed from the result) | Document of the new user is returned and the details all match  (Pass) |
| 3 | Hash and register a new user of largest size form input | Edge | Username:  Integration  Password:  Integration12  IntTest  newUser: {  username:  Integration  Password:  Result of hash  Firstname:  Int  Lastname:  Test  Email:  testing@test  } | Document of the new user is returned and the details all match (excluding password which is removed from the result) | Document of the new user is returned and the details all match  (Pass) |

### hash.js integrated with loginAuthentication.js

As users documents don’t store users passwords in plain text to authenticate the login we must first hash and salt the users password. Then we can check for a match in the database

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | Hashing of a valid password that with correct username authenticates login | In range | Username:  IntName  Password:  IntTest | Hash is correctly made and a matching document is found | Hash is correctly made and a matching document is found (Pass) |
| 2 | Incorrect username and password combination | Erroneous | Username:  InvalidName  Password:  IPassword | No document is returned (null) | null returned  (Pass) |
| 3 | Valid login attempt with shortest username and password allowed by form | Edge | Username:  I  Password:  IntTest | Hash is correctly made and a matching document is found | Hash is correctly made and a matching document is found (Pass) |
| 4 | Invalid login attempt with shortest username and password allowed by form | Erroneous | Username:  I  Password:  NotPas | No document is returned (null) | null returned  (Pass) |
| 5 | Valid login attempt with largest username and password allowed by form | Edge | Username:  Integration  Password:  Integration12 | Hash is correctly made and a matching document is found | Hash is correctly made and a matching document is found (Pass) |
| 6 | Invalid login attempt with largest username and password allowed by form | Erroneous | Username:  Integration  Password:  Integration99 | No document is returned (null) | null returned  (Pass) |

# Full System Black Box Testing

This section of our testing will focus on interacting with the system as a whole through the user interface as users.

## Form Validation

On submission of the forms in the user interface client-side checks in the html will be made these include the use of required tags so entries cannot be left blank on top of pattern matching so only certain length username and passwords can be used. Then if these fields were incorrectly filled a user-friendly pop-up should be displayed. We conducted the following tests on the forms as users.

### Registration form

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected outcome | Actual Outcome |
| 1 | All fields filled in within the bounds | In Range | Form filled with all valid characters | Registration allowed to progress | Registration allowed to progress |
| 2 | Testing leaving each field blank one by one | Erroneous | No input | A pop-up appears telling the user to fill in the empty fields | A pop-up appears telling the user to fill in the empty fields |
| 3 | Username inputted of shortest length (1 char) | Edge | Username: T | Registration allowed to progress | Registration allowed to progress |
| 4 | Username inputted of longest length (10 chars) | Edge | Username: TenChars10 | Registration allowed to progress | Registration allowed to progress |
| 5 | Username over the largest length of 10 | Erroneous | Username: ElevChars11 | Pop-up alerts user that the username is of incorrect length | Pop-up alerts user that the username is of incorrect length |
| 6 | Password of shortest length (6 chars) | Edge | Password:  6Chars | Registration allowed to progress | Registration allowed to progress |
| 7 | Password of shortest length (12 chars) | Edge | Password:  12CharsPasswo | Registration allowed to progress | Registration allowed to progress |
| 8 | Password below shortest length | Erroneous | Password:  5Char | Pop-up alerts user that the password is of incorrect length | Pop-up alerts user that the password is of incorrect length |
| 9 | Password above longest length | Erroneous | Password:  12CharsPasswor | Pop-up alerts user that the password is of incorrect length | Pop-up alerts user that the password is of incorrect length |

As well as the form validation if the user tries to use a username already in the system they should not be allowed to and receive a user-friendly alert compared to a stack trace.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected output | Actual output |
| 1 | Test for a valid username accepted form that already exists alert is shown | Erroneous | Username:  Testing | Alert is shown | Alert is shown to user which tells them to try another username (Pass) |

### Text blocks game and game timer

During all the games there is a 2-minute timer this component is imported in to work alongside the games. This will be used to calculate scores as well as to limit the games to that 2 minutes. So when the timer runs out the game will end. But if the task is completed in time the timer will stop and a score will be calculated. This will meet the requirements for the games to only last 2-minutes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test | Type | Input | Expected Outcome | Actual Outcome |
| 1 | On completion of the game the timer is stopped | In Range | Completion of the game | Timer is stopped and pop up is shown | The timer was stopped and the pop up was shown  (Pass) |
| 2 | Timer ends the game once 2-minutes are spent | Edge | Start the game and wait 2 minutes | Timer reaches 0 and pop up appears | Timer reaches 0 and pop up appears (Pass) |
| 3 | On refresh of the game the timer is restarted | In Range | Allow the game to run the click the restart icon. | Game and timer restart | Game and timer restart  (Pass) |