1.2.1 Unit Testing

For the unit testing we chose to test several important server calls. These are crucial because it is how the client side and server side of our application communicate. Without these working bug-free the application would not be able to do much.

All of the server calls interact with the database in some way, either to retrieve data or to update it. In order for testing to work consistently we needed a database that would be constant. For this we made a much smaller database purely for testing purposes. When running tests locally, this database will be used instead of the official one. That way we will know exactly what information is supposed to be stored in the database and can expect consistent results from our tests.

Laravel has an official unit testing framework, but it expects all of the mvc to be used. Since we replaced the view with React, this framework was not ideal. Instead we have our own testing script, coded in javascript that runs all of our tests. Neither the testing script nor the database will be included in the release version of our application, since they are only for testing.

For development we created a javascript object, realServerBridge, which we used to abstract all of the server calls. We used this to make the server calls when testing.

Login

Our login server call is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **URL** | **Purpose** | **Type** | **Input** | **Returns** |
| /login | Login verification | POST | {username:’’, password:’’} | {"success":"true","username":"JASONB","isAdmin":"true"} |

It expects a JSON object including a username string and a password string. It will return a JSON object with a string holding the success state, the username, and a string saying whether or not the user is an admin. The test cases are as follows:

|  |  |  |
| --- | --- | --- |
| **Description** | **Input** | **Expected output that was tested** |
| Call is made with a username that exists in the database with the correct password for that user | {username: ‘User’, password:’password’} | success=’true’  username=’User’  isAdmin=’false’ |
| Call is made with a username that does not exist in the database (sending an empty username and password is treated the same way as this) | {username:’notauser’, password:’password’} | success=’false’ |
| Call is made with existing user, but the wrong password | {username:’User’, password:’notthepassword’} | success=’false |
| Call is made with the admin’s username and password | {username:’Admin’, password:’password’} | success=’true’  username=’Admin’  isAdmin=’true’ |

We tested by making the server calls and then verifying the output we received. No set up or tear down was needed to keep the test consistent.

Set Preferences

Users can specify preferences for their schedule, such as a desired course load or a day of the week they want free. The server call to update the preferences for a user is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **URL** | **Purpose** | **Type** | **Input** | **Returns** |
| /editpreferences | Edit user’s preferences | POST | {username:’’,cload:'',dayoff:'',preftime:''}} | {"success":"true","username":"JasonB","courseload":"5","dayoff":"Monday","preftime":"Mornings"} |

It expects a JSON object including strings for the username, course load, day off, and preferred time and will return a JSON object including the success state, as well as the info that was sent. Our client side of our project can handle invalid data entered for preferences by replacing it with default information, so for testing we were not concerned with that. The test cases are as follows:

|  |  |  |
| --- | --- | --- |
| **Description** | **Input** | **Expected result** |
| Call is made with a username that exists | {username:’Jason’, cload:’5’, dayoff:’Monday’, preftime:’Mornings’} | Data updated in the database |
| Call is made with a username that is not in the database | {username:’notauser’, cload:’5’, dayoff:’Monday’, preftime:’Mornings’} | Output:  success=’false’ |

For the first test case we had to also use the server call to retrieve the preferences in order to check that the information had actually been updated.

The methods for both cases expect a username in the cookies, so this had to be done as set up for the cases. This had to be removed for the tear down and the preferences had to be set back to blank values, so the tests could be consistent.

Registration

Registration was important to test. If this didn’t work it could cause secuirity issues or could make it so that people could not even sign up for the site and could never access it. The server call is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **URL** | **Purpose** | **Type** | **Input** | **Returns** |
| /register | Register user | POST | {username:’’, email:’’, password:’’} | {"success":"false","username":"SprinkKing","error":"usernametakenalready"} |

It expects a JSON object containing strings for the username, e-mail, and password and will return a JSON object containing strings for the success state, the username, and an error message. The success will be false only if a user with that username already exists. The error will be blank if the call was successful.

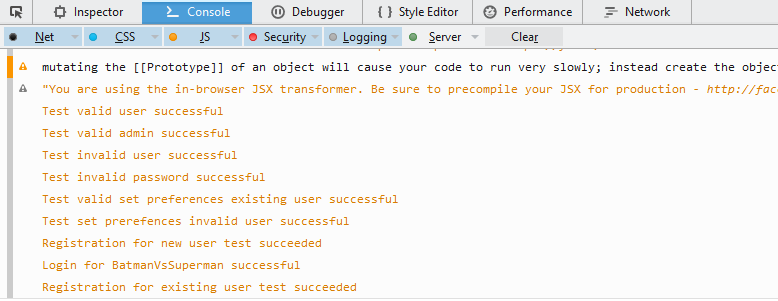
The test cases are as follows:

|  |  |  |
| --- | --- | --- |
| **Description** | **Input** | **Expected output that was tested** |
| Call is made with a new username | {username:’BatmanVsSuperman’, email:’justiceleague@gmail.com’, password:’password’} | success=’true’  Also the following login call should succeed, showing that the user was added in the database |
| Call is made with an existing username | {username:’Jason’, email:’jason@hotmail.com’, password:’password’} | success=false |

No set ups were needed. For tear down we had to remove the BatmanVsSuperman user from the database, so the test would work again next time.

Running the tests

All of the tests succeeded. Here was the console after they ran:



Unit Testing Code

For any server calls that expect information back from the server, a callback method must be sent to the server bridge object. This is due to the asynchronous nature of AJAX calls. When the data is finally received, it will be sent as the only argument to that method and then it can be tested. This is why we send functions as an argument for the server bridge methods.





