**User Story Scenarios & Acceptance Criteria**

To view suggested user stories/acceptance criteria for actual acceptance test/project review please view the following links:

[System and Unit Test Document](https://docs.google.com/document/d/1dyowr6uAQ8Z9wCdqShwv1fXkmISN3AeqnMZhDGjZe0c/edit)

[DoD - Definition of Done / Acceptance Criteria](https://docs.google.com/document/d/1ixBqLAMYZHZdfAXxZsfOHgVQ69NhqW0GBU5qScKLuoA/edit)

**Known Issues and Potential Solutions**

| **The Problem** | There are many front-end console warnings with the description *“failed to parse source map”.*    Based upon our observations, these warnings are not related to any functionality problems in the front-end or back-end. |
| --- | --- |
| **Location of Problem** | The problem is related to the code for the calendar located within the home page. This is because the “devexpress” library we are using for the calendar utilizes Typescript, but we have written our software in Javascript. |
| **Possible Solutions** | * A possible solution is to rewrite our software in Typescript, rather than in Javascript. This will prevent any problems between the Typescript coe used by the calendar library, and our Javascript code that utilizes it. * We have not seen these warnings related to any functionality issues in our code. In other words, users will be able to utilize our web application without any knowledge about these warnings. We can double check that these warnings do not affect the functionality of the application. Once we know for sure that these warnings are not problematic, we can modify Create-React-App so that it will ignore these warnings. |

| **The Problem** | The console states that we have linter warnings and errors when starting the front-end part of our web application. |
| --- | --- |
| **Location of Problem** | The problem exists throughout all of our front-end source code. |
| **Possible Solutions** | * We need to go through all the front end source files and fix the linter warnings and errors. * We can then install [ESLint](https://eslint.org/) to inform us of any linter warnings moving forward.   + However, the problem with ESLint is that it will not allow the front-end to start until any linter errors are fixed. Thus, we would need to fix all the linter errors first before using ESLint. |

| **The Problem** | Information about the user is stored as React “state”, rather than in the browser storage or as a cookie. Thus, whenever any web page is refreshed, the user will be logged out. |
| --- | --- |
| **Location of Problem** | This problem exists throughout all of our front-end source code. No matter which web page the user is on, if the page is refreshed, they will be logged out. |
| **Possible Solutions** | We would need to store the user information in the [browser’s local storage](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage). Whenever the web application is refreshed, we can program our software to obtain any user information in the local storage. This will allow the user to be logged in, even though they refreshed their webpage. |

| **The Problem** | The front-end and back-end of our web application exchanges information in plain-text. We currently have not taken any security measures to encrypt our HTTP requests and responses. |
| --- | --- |
| **Location of Problem** | The location of the problem exists in the communication between the front-end and the back-end. In the front-end, we send unencrypted HTTP requests, and there exists no mechanisms in our back-end to send encrypted HTTP responses. |
| **Possible Solutions** | * We would allow both our front-end and back-end to communicate with each other in HTTPS. We can obtain HTTPS certificates for free through [Let’s Encrypt](https://letsencrypt.org/). * If using HTTP is not an option, we can utilize [JSON Web Tokens](https://jwt.io/introduction) to secure our HTTP requests and responses. There exists [a Node.js library for JSON Web Tokens](https://www.npmjs.com/package/jsonwebtoken) that we can utilize. In the front-end, we can store the HTTP request payload in an encrypted JSON web token, which can be decrypted by the server in the back-end. |

| **The Problem** | Any string can be written in the “email” field for the Register webpage, even if the string is not an email. Thus, a user can create an account with an “email” that is not actually an email. |
| --- | --- |
| **Location of Problem** | The problem exists in our front-end file “Register.jsx”, which contains the source code for our register page. |
| **Possible Solutions** | * We can utilize a regular expression to detect whether or not a string is a valid email. We could make the regular expression ourselves, or find one online (citing the authors who created the regular expression). * A better solution is to utilize email verification. When a user creates an account, we can utilize an API or external library to send an email to the user’s email. If the email the user entered does not exist, then the account will not be created. |

| **The Problem** | The “Home” component renders twice whenever the user enters the Home Page. |
| --- | --- |
| **Location of Problem** | The problem is located in the source file “HomePage.jsx”. |
| **Possible Solutions** | We are unsure why this is the case. As a result, we would need to do further research on this issue. However, we have not seen this problem impact the user experience for our web application. |

| **The Problem** | We currently do not have any functionality to handle the event when the user enters an invalid URL in our web application. When the user goes to an invalid webpage, they will see a blank webpage. |
| --- | --- |
| **Location of Problem** | The location of our problem is with how we are utilizing React routes in our source file “Main.jsx”. |
| **Possible Solutions** | We would need to modify our React Routes in “Main.jsx” to redirect all links not explicitly specified in “Main.jsx” to the login page when the user is not logged in. If the user is logged in and goes to an invalid webpage, we can redirect them to a “404 Not Found” webpage. |

| **The Problem** | If the user is on the Dashboard page, they would need two clicks on the “->” button, rather than just one click, to go back to the Home page. |
| --- | --- |
| **Location of Problem** | Since the “->” button goes to previous web pages, the problem lies with how we utilize React Routes. Thus, the bug lies in our “Main.jsx” source file, which contains our code that uses React Routes. |
| **Possible Solutions** | We need to investigate the “Main.jsx” source file with the use of debugging statements. In addition, we can write a test to navigate back and forth through different webpages in our web application. This will allow us to test that our solution works once we have wrote an implementation to address this issue |

| **The Problem** | There are moments in which the Bagel graph will not update for a task that has been recently marked as finished. How this bug can be reproduced is described in the pictures below.    Add at least three tasks. The first task that is marked as completed does not update the Bagel graph.    Marking subsequent tasks as finished will update the Bagel graph to display both the first task and all subsequent tasks.  We have added fixes to this bug. However, even before we addressed this bug, it would not appear all of the time. As a result, we would like to acknowledge the existence of this bug, as the bug may still be present even with our fixes.  **Note:** The bug is not consistently reproducible when following the steps described above. |
| --- | --- |
| **Location of Problem** | The bug involves the Bagel graph, and is thus present in the front-end source file “Bagel.jsx”. |
| **Possible Solutions** | We do not yet know the specific cause of this bug. The best step we will need to take is to investigate the code in the “Bagel.jsx” source file. In addition, creating unit tests that test the functionality of the Bagel allow us to ensure that this bug has been fixed. |

| **The Problem** | All our back-end tests pass when we first start up the database. However, our back-end tests all subsequent times we run it after starting up the database. The reason for this is that our test code tests the actual database, rather than a dedicated test database.  The database must be turned off, and then turned back on, before running the tests. |
| --- | --- |
| **Location of Problem** | The problem is located when running the back-end test source file “tasks.test.js”. |
| **Possible Solutions** | We need to create a test database separate from the main database that connects with our front-end. This test database will need to be reset after running the test file(s), as the tests are dependent on whether or not certain entries are in the database. |

| **The Problem** | Although our back-end testing are working as intended, we have an error which prevents our back-end testing source code from exiting:    We do not know why this issue is happening. However, from our observations, it does not affect our tests from functioning correctly. |
| --- | --- |
| **Location of Problem** | The problem is located when running the back-end test source file “tasks.test.js”. |
| **Possible Solutions** | We need to learn more about the Node.js HTTP library, in addition to the supertest library, in order to investigate why this is the case. |

| **The Problem** | We get an “Axios Network Error” in our front-end tests, even though they pass. This error can be found when running the front-end tests in the “\_\_tests\_\_” folder when typing “npm test”. |
| --- | --- |
| **Location of Problem** | The error is coming from the front-end tests files in the front-end “\_\_tests\_\_” folder. |
| **Possible Solutions** | We originally thought that the problem was because we did not have the back-end and database running before starting the front-end tests. However, starting the back-end and database before running the front-end tests does not remove this issue.  We would need to do more research on the Axios API as a first step into discovering why this issue has occurred. |