Computer Science Clinic

Final Report for
Cradlepoint

Cradlepoint 2021-2022 Final Report

May 6, 2022

Team Members

Jessica Kwok (Fall Project Manager)
Dylan McGarvey (Spring Project Manager)
Matt Tran
Natasha Wong

Advisor

Melissa O'Neil

Liaisons

Lynn Gates
Aaron Ebling
Alex Terrell
James Hawk
Chirs LaPeters

Contents

Chapter 1

Introduction

In this chapter, we briefly introduce the company Cradlepoint, and specifically their Center of Excellence team. Then, we move on to discuss the problem statement and the solution we designed. Finally, we summarize our yearlong development process. This includes our original schedule of deliverable, the actual timeline that happened, and reasoning behind the difference between the two.

1.1 Company Background

Cradlepoint is one of the leading solutions providers for LTE and 5G wireless network problems.[1] It provides a wide range of physical products including cellular and Wi-Fi antennas, private LTE and 5G solutions for businesses, and other network accessories. In addition, Cradlepoint also offers a cloud-based management and control platform, networking technologies, routing and security, and other functionalities to support its network equipment. Its customers ranges from private businesses to the public sector.

Of particular significance, Cradlepoint's Sales Engineering *Center of Excellence* (COE) team, formed by a group of specialists, architects, corporate *system engineers* (SE), and lab designers, consults with customers to provide solutions based on their specific situations and use cases. The COE does so by executing customized *proof-of-concept* (POC) test plans and uses the results of these tests to provide workable insights and compelling proof points for the technical architects and operators to design and implement network solutions.

[1] As noted by Cradlepoint in their project description.

Deliverable	Date
Minimum Viable Product (MVP)	December 7, 2021
Midyear Update	December 7, 2021
MVP Documentation and Technical Debts	February 1, 2022
Refined Final Product (RFP)	April 1, 2022
Software Feature Freeze	April 8 2022
RFP Documentation and Tests	April 14, 2022
Coding Freeze	April 22, 2022
Project Poster	April 22, 2022
Final Report	May 6, 2022

Table 1.1 Original Schedule of Deliverables

1.2 Problem and Desired Solution

Due to the valuable insights that these POC test plans can provide to both Cradlepoint and its customers, Cradlepoint has seen a dramatic growth in demand for POC testing. Currently, much of the work the COE team has done to create, collect, manage, and present its POC test plans occurs manually. This complicated testing process involves multiple stakeholders, including customers, partners, SEs, and POC engineers. To complete a test cycle, different stakeholders need to frequently exchange documents, ask for and provide approval and feedback, and present results. Thus, the current process is time consuming and resource intensive for the company. To ameliorate this problem, Cradlepoint wishes to automate its current manual process via a web based portal that will manage the workflow of the POC testing process from start to finish.

1.3 Development Process

The original schedule of deliverables can be seen in the table above (Table 1.1). In the Fall Semester, we mainly focused on creating a *Minimum Viable Product* (MVP) for our web application, which involved creating most of the user interfaces, API endpoints, and integration between them. Due to the complexity of the database schema, some of these tasks required a longer amount of time than expected, and required revisions to get a version of the schema that everyone agreed upon. The unexpected length of these tasks resulted in pushing back some of the bug-elimination and re-

ducing of technical debts from the MVP to the start of the Spring Semester. After those tasks were accomplished, we started working towards the *Refined Final Product* (RFP) in the remaining of Spring Semester. Our original focus for the RFP was to be on the Review and Iteration Process, but this was pushed towards being an area of future work (see Section 6.5.1) due to changes in priority. Instead our focus was put towards hosting our application through *Amazon Web Services* (AWS), refactoring APIs to be more RESTful, and creating functionality for cloning from libraries and data deletion.

1.4 Conclusion

With this context in mind, this document aims to provide a thorough source of knowledge for all technical aspects of our web application. In chapter 2, we outline the technologies that we utilized in order to make our application work. Following this, chapters 3 and 4 discuss our backend schemas and hosting respectively. In chapter 5, we shift our focus to our frontend architecture and various technologies. Finally, in chapter 6, we critically highlight areas of future work in which we bring attention to items and tasks that we believe should be accomplished next in the future development of the web application. Thus, this document hopes to leave its reader with a well-rounded understanding of all the aspects of the development of the web application.

Chapter 2

Technology

This chapter covers the primary technology stacks that were used in the project, as well as brief descriptions of what they each do. Then, it documents the most frequently used frontend libraries, along with descriptions of how they were used.

2.1 Description of Stack

Our technology stack consists primarily of React built through the NextJs framework which provides middleware and backend routing capabilities in addition to React's conventional frontend features.[1] Furthermore, we rely on MongoDB to serve our database and a number of Amazon Web Services such as S3 and Amplify to support our hosting needs for image and file storage and the website itself, respectively.[2] [3] Each of the frontend and backend features are described in further detail in their respective sections.

2.2 Frontend Libraries

There are a number of frontend libraries we have leveraged to ease and simplify the building process of the web application. Below is a list of these libraries:

[1] Next.js: https://nextjs.org
[2] MongoDB: https://www.mongodb.com
[3] Amazon Web Services: https://aws.amazon.com

- Material UI: provides many of the simple UI components we customize or directly use.[4]
- React Modal: provides the modal component we use throughout the app.[5]
- Toastify: provides error pop ups.[6]
- React Table: provides the table UI we use, which supports robust functionality such as sorting and organizing data within the table.[7]

2.3 Conclusion

The chapter serves as a brief guide for developers to know where to look up resources and documentations regarding the specific libraries when making changes. In the following chapters, we cover how some of the primary tech stacks were used, both in the backend and the frontend.

[4]Material UI: https://mui.com/
[5]React Modal: https://reactcommunity.org/react-modal/
[6]Toastify: https://aleab.github.io/toastify/
[7]React Table: https://react-table.tanstack.com/

Chapter 3

Backend

This chapter shows all of the information necessary to understanding the backend services of our application. The design's layout and structure is explained within this chapter. This section also demonstrates how we interact with the database and modify any of our stored data.

3.1 Connecting to MongoDB

We have created a file in the utility folder, `./util/mongodb.js` which creates a client utilizing the Mongodb package for Nodejs, and returns that as a result. The returned object is then used for any actions involving the database.

3.2 API Routes

Each of the API Routes can be found in the `./pages/api` folder. In Next.JS, each of these files will create an API route which can be called with `[host]/api/[pathToFile]`. Additionally, many of these routes have exported their main logic to a separate function, either within the file itself, or to an Entry file in the util folder, such as `./util/addEntry.js`, which contains logic for all of the add API Routes. The reason for separating the logic into another function is so that the default exported function can be utilized when the API route is called using an http request, whereas solely the API Route's logic is directly imported into a page when rendering. Importing the function directly makes the application run faster by not having to await an API call from its own backend, and eliminating the use of an

HTTP request when not needed.

We have detailed the various endpoints needed to allow our frontend to communicate with our backend, in this supplemental document.[1]

3.3 Data Schema

In this section, you can find the schema that was used for each collection, including:

- engagements
- statusCodes
- testPlan
- testPlanLibrary
- testCases
- testCaseLibrary
- tests
- testLibrary
- results
- user (created but not implemented in the application yet)
- userTypes (created but not implemented in the application yet)
- device
- BOM (not a collection, a customized object used in other collections)

[1] Endpoints: https://bit.ly/3oeUADE

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
createdOn	[Date]	
testPlanId	[Object ID]	ID of the active test plan
statusCode	Int	Relates to Status Codes in the collection
SFDC	String	Should be a url
customer	String	
description	String	
POC_Engineer	String	POC Engineer Responsible
SE	String	Site Engineer Responsible

Table 3.1 engagements

Field	Data Type	Notes
_id	Int	Each status is represented by an integer
status	String	Status to be displayed
description	String	More detailed description of the status

Table 3.2 statusCodes

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
createdOn	[Date]	
summaryBOM	Array[Object]	Array of objects containing BOM Objects (shown later)
testCases	Array	Array of Ids of the associated test cases
engagementId	[Object Id]	Id of the associated engagement
deviceConfig	String	Should be replaced with filename later, and be a download link
isActive	Boolean	Whether or not the test plan is active
version	String	Version of the test plan
authors	Array	Array containing all of the authors of this test plan
customerFeedback	String	
description	String	
name	String	

Table 3.3 testPlan

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
createdOn	[Date]	
summaryBOM	Array[Object]	Array of objects containing BOM Objects (shown later)
testCases	Array	Array of Ids of the associated test cases
engagementId	[Object Id]	Id of the associated engagement
deviceConfig	String	Should be replaced with filename later, and be a download link
version	String	Version of the test plan
customerFeedback	String	
description	String	
name	String	

Table 3.4 testPlanLibrary

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
name	String	
description	String	
config	String	Should be replaced with filename later, and be a download link
topology	String	Filename in the S3 Bucket
testPlanId	[Object ID]	Id of the associated Test Plan
tests	Array[Object ID]	Array of IDs all associated tests
BOM	Array[Object]	Array of objects containing BOM Objects (shown later)

Table 3.5 testCases

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
name	String	
description	String	
config	String	Should be replaced with filename later, and be a download link
topology	String	filename in the S3 Bucket
testPlanId	[Object ID]	Id of the associated Test Plan
tests	Array[Object ID]	Array of IDs of all associated tests
BOM	Array[Object]	Array of objects containing BOM Objects (shown later)

Table 3.6 testCaseLibrary

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
name	String	
description	String	
testCaseId	[Object ID]	Id of the associated Test Case
resultStatus	String	Status of the results
results	Array	Array of Ids of the associated results

Table 3.7 tests

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
name	String	
description	String	
testCaseId	[Object ID]	Id of the associated Test Case

Table 3.8 testLibrary

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
createdOn	[Date]	
testId	[Object ID]	ID of the associated test
resultStatus	String	Enum of: Pass, Unknown, or Fail
evidence	String	Unique filename in the S3 bucket
description	String	

Table 3.9 result

Field	Data Type	Notes
firstName	String	
lastName	String	
userType	int	Referenced from userTypes collection, specifically the code
email	String	
userId	String	Unique identifier for a user

Table 3.10 user

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
createdOn	[Date]	
code	string	A numeric code to represent different usertypes, used for lookup
userType	String	Actual type of the user

Table 3.11 userTypes

Field	Data Type	Notes
_id	[Object ID]	Mongo datatype for IDs
deviceName	String	
codeVersion	String	
SKU	String	
deviceType	String	Should be physical or software

Table 3.12 device

Field	Data Type	Notes
isOptional	Boolean	Whether or not the device is optional
_id	[Object ID]	Identifier for specific BOM entry
quantity	int	Amount of that device needed
deviceId	[Object Id]	Id of the device from the devices collection

Table 3.13 BOM

3.4 Usage

The elements described in this backend section are typically utilized through the frontend. Users will interact with the interface that we have developed, and call the methods and routes described here, in the backend section. Additionally, the API routes covered in this section may be further utilized in order to integrate into other systems. Our GET routes could be utilized to pull information about the data that the application stores, and any PUT routes could be utilized to programmatically add data into the application. Having established the usage of our backend as well as its routing and schemas, we can now discuss how this is hosted.

Chapter 4

Hosting

This chapter outlines the way in which we hosted and recommend this application to be hosted in the future, including the services utilized. Relevant configurations for these services are included in their sections where needed.

4.1 Set up

First, note that because of compatibility issues with AWS Amplify at the time that we attempted hosting, we are running the latest version of Next.JS 11.

1. Host the code base for the application in a GitHub repository (Other services such as bit bucket would work as well, using just the files themselves may also work, but this guide is written assuming that GitHub is the hosting repository in mind).

2. Obtain a GitHub account that has admin access to the repository, since this is required by AWS in order to host it on Amplify.

3. Log into AWS, and ensure that you are in the region where you would like the application to be hosted. We will now set up both the S3 bucket, and the Amplify application.

4. First, we set up the S3 bucket. Navigate to the S3 section of AWS. Create your bucket, and make sure to allow public access.

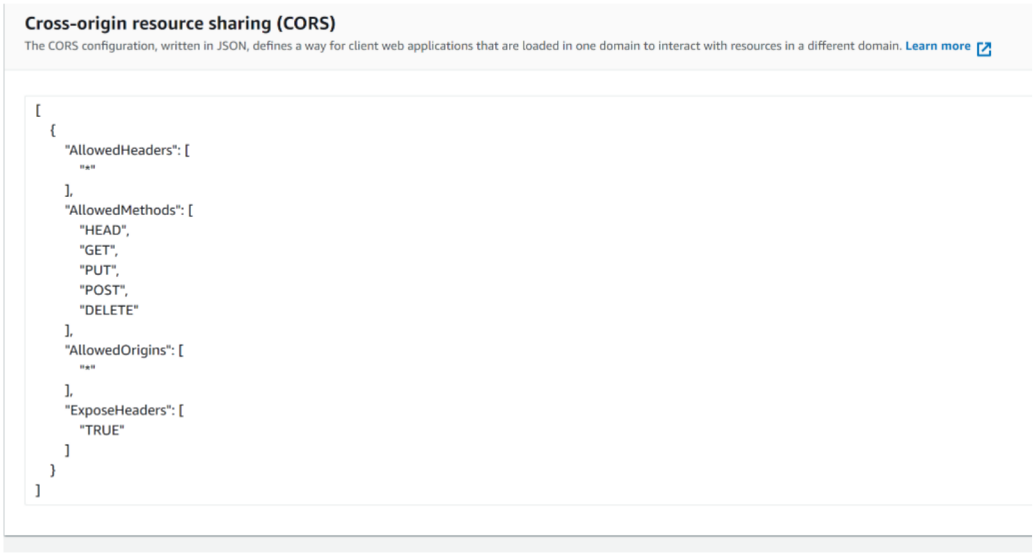

Figure 4.1 Set CORS Permissions

5. Set the CORS permission to be as follows:

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "HEAD",
            "GET",
            "PUT",
            "POST",
            "DELETE"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [
            "TRUE"
        ]
    }
]
```

This section can be found under the Permissions tab of the bucket, and should should look like the above figure.

6. Finally, set up a new user, to be used by the application, and give it read and write permissions to the bucket. Note down the Secret Key because this will not be accessible later.

7. Now, we will begin hosting on Amplify. Navigate to the Amplify application on AWS, and make sure that you are in your desired region.

8. Select new app, then host app, and choose to host from GitHub.

9. Connect your account which has admin access to the repository and select the application.

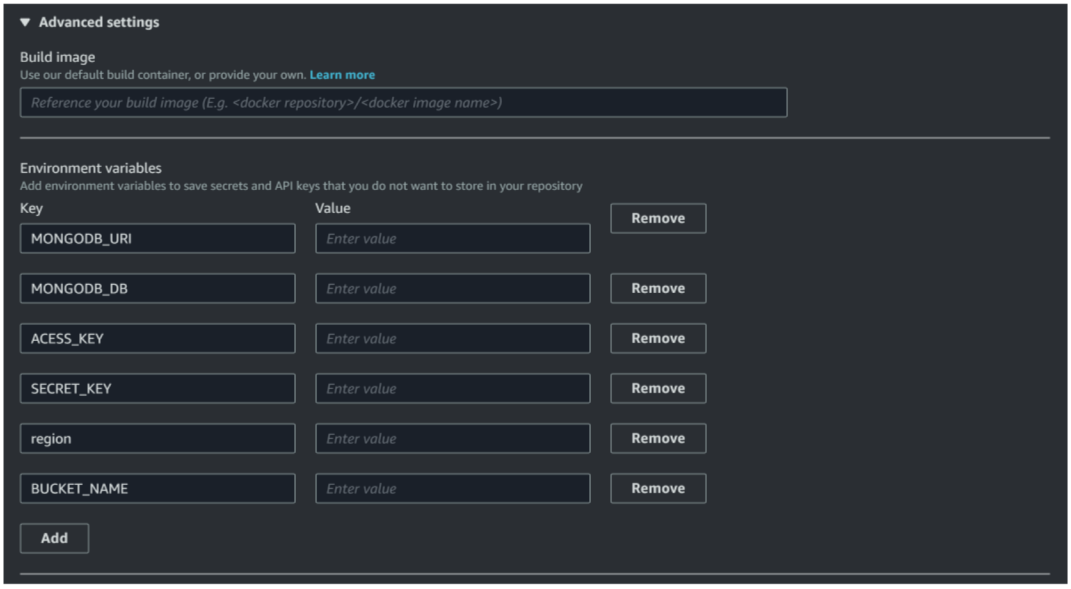

Figure 4.2 Set CORS Permissions

10. Under advanced settings, add in the values for the following environment variables, depending on how they're hosted for you.

```
MONGODB_URI = URI string from Mongo to connect to the db instance
MONGODB_DB = database name in Mongo that stores all the data
ACCESS_KEY = Access key of the access account
SECRET_KEY = Secret key of that same account
region = region that the S3 bucket is hosted on
BUCKET_NAME = name of the S3 bucket
```

11. Click next, and then deploy and the application should start building.

4.2 Mongodb

1. Set up a Mongodb instance, and make sure to set permissions such that it can be written to and read from.

2. Create a database for use by the application, this will be utilized in our environment variables when setting up the Amplify instance for hosting.

3. An outline of the schema can be found in the data schema section. Create a Mongodb instance with these parameters.

4.3 Conclusion

Once this application is hosted, it should already have the workflow set such that any push to the main branch causes an automatic rebuild of the website. There may exist other services that could accomplish the same tasks as we've done here, but these were the ones that we utilized for this project. At this point, with the application hosted, we can now begin to interact with the website's frontend. The next section outlines the flows behind how the user interacts with the application.

Chapter 5

Frontend Architecture

This chapter is intended to give future developers a more concrete understanding of the frontend architecture. It first covers an exhaustive list of pages that the website consists of, in addition to user actions supported on each of of these pages. Then, it moves on to the User Interface (UI) components that were made to be reused through out the entire website. Finally, it documents each modal and their functionalities, along with the different modal flows that are embedded in the website.

5.1 Pages

The following is an exhaustive list of pages that are currently supported:

- Home: The home page shows all the existing engagements of the users.
  Supported user actions on page:
  - view Engagement details
  - edit Engagement
  - delete Engagement
- Engagement Details (Figure 5.1): After the user selects a specific engagement in the home page, they will see this page, which displays all the details and descriptions of the engagement. In addition, they will also see the active test plan, the archived test plans, and the summary BOM of the active test plan under this engagement.
  Supported user actions on page:

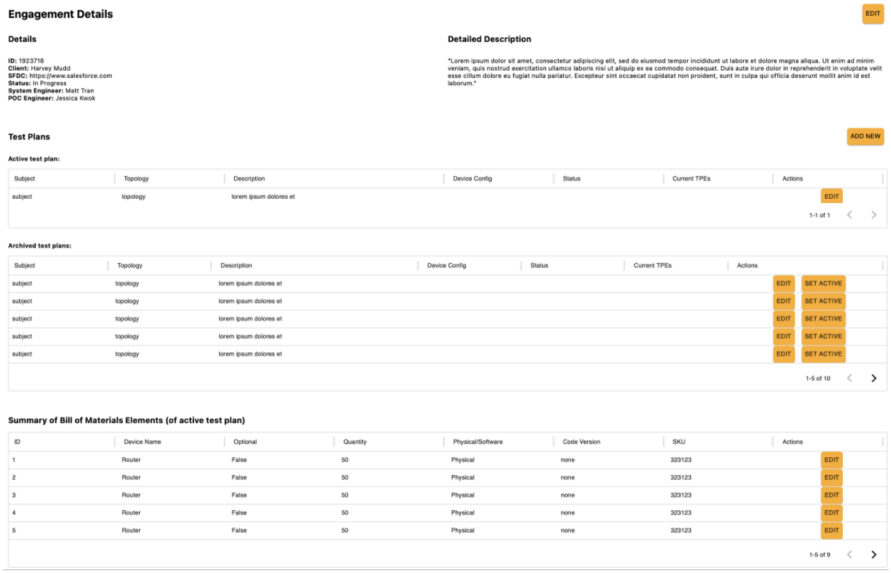

Figure 5.1 Engagement Details

  - edit the Engagement details
  - set archived Test Plan as active
  - add new Test Plan
  - view Test Plan Details
  - delete (non-active) Test Plan

- Test Plan Details (Figure 5.2): This page displays all the details of the test plan. In addition, users will see all the test cases under the test plan and its summary BOM.
  Supported user actions on page:

  - edit the Test Plan details
  - add new Test Case
  - view Test Case Details
  - delete Test Case

- Test Case Details (Figure 5.3): This page displays all the details of the test case. In addition, users will see all the tests under the test case and its BOM. Users can edit the test case details, add new tests, add

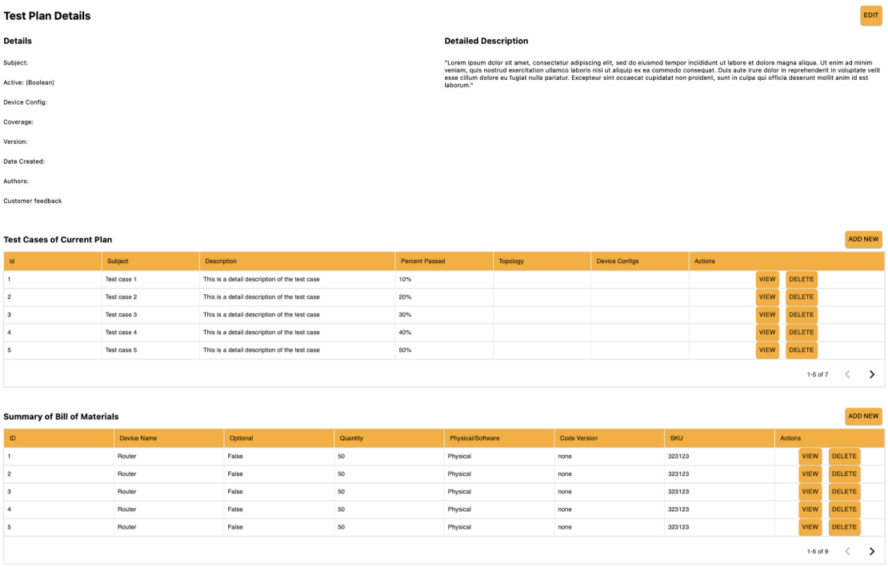

Figure 5.2 Test Plan Details

new devices to the BOM, or click into a test to view further details.
Supported user actions on page:

  - edit the Test Case details
  - add new Test
  - view Test Details
  - delete Test
  - add device to BOM
  - edit BOM device quantity and setting
  - delete device from BOM

- Test Details (Figure 5.4): This page displays all the details of the test. In addition, users will see all the results recorded under the test. Users can edit the test details and add new results.
Supported user actions on page:

  - edit the Test details
  - add new Result
  - view Result details

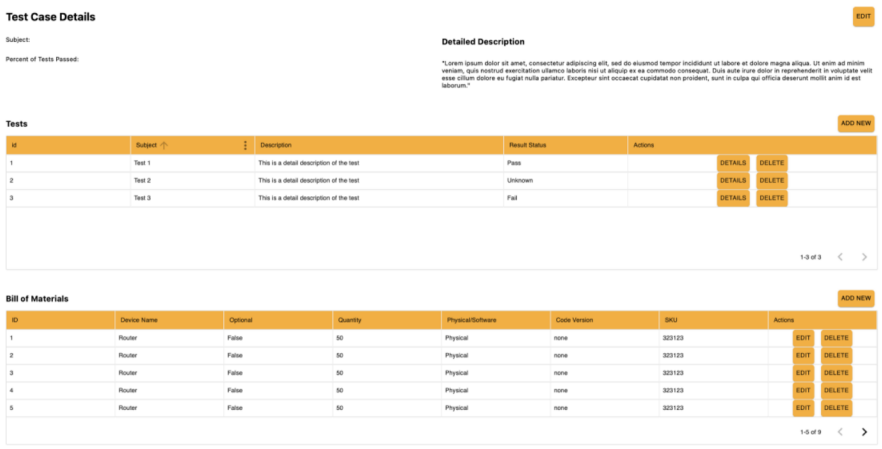

Figure 5.3 Test Case Details

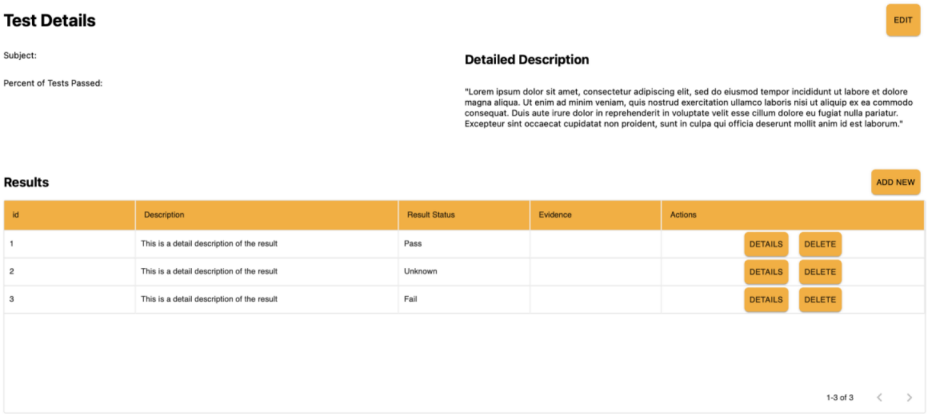

Figure 5.4 Test Details

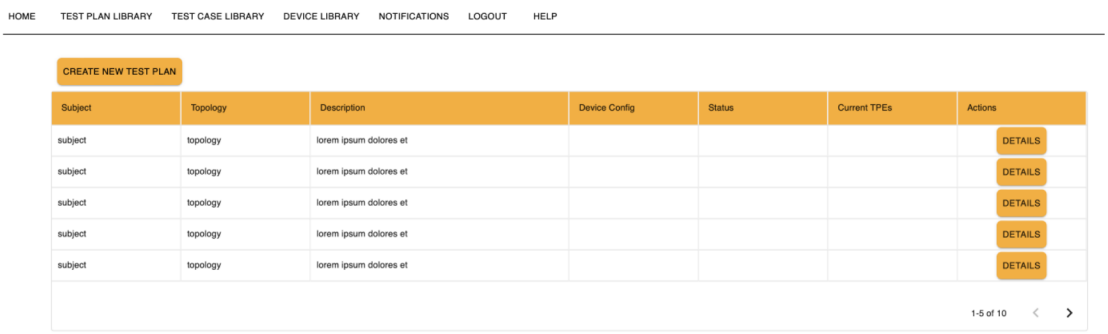

Figure 5.5 Test Plan Library

  - delete Result

- Result Details: This page displays all the details of a Result. Supported user actions on page:
  - edit Result details
- Test Plan Library (Figure 5.5): Displays a table of all existing test plans.
- Test Case Library: Displays a table of all existing test cases (similar to Figure 5.5).
- Device Library: Displays a table of all available devices (similar to Figure 5.5).

5.2 Components

Since there are components that can be reused in different pages and modals, we decided to first build out these common components. The components include

- menu bar
- plain screen
- split screen
- styled buttons

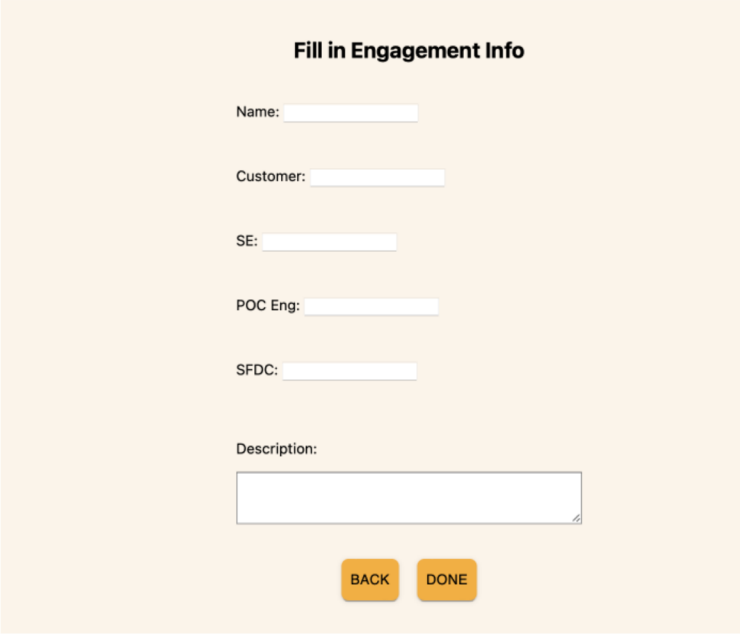

Figure 5.6 `EngagementModalForm.jsx`

- drop-down
- text fields
- tables

5.3 Modals

The following is an exhaustive list of existing modals for reference. When and how the modals are used is explained in the upcoming subsections.

- Modal Forms: forms in which users can fill in/edit the information of the respective entity (e.g. Engagement Modal Form allows users fill in/edit the information of an Engagement, shown in Figure 5.6).
  - `EngagementModalForm.jsx`
  - `TestPlanModalForm.jsx`
  - `TestCaseModalForm.jsx`
  - `TestModalForm.jsx`

Figure 5.7 StartModal.jsx

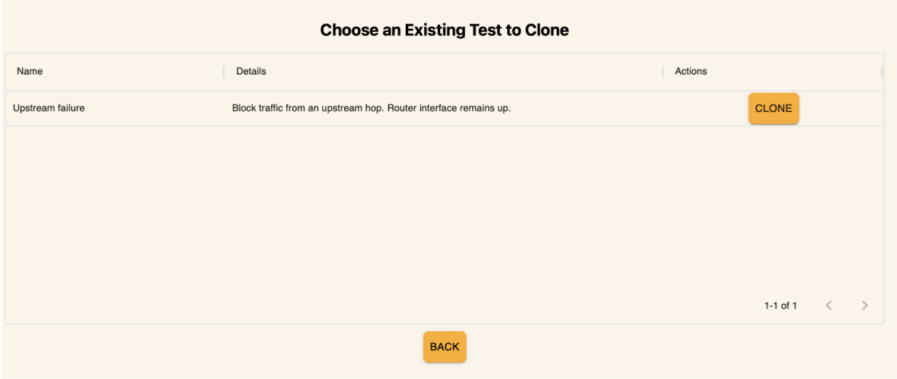

Figure 5.8 `CloneModal.jsx`

  - `ResultModalForm.jsx`
  - `DeivceModalForm.jsx`

- `StartModal.jsx` (See Figure 5.7) Allows users to choose whether they want to create a new [Test Plan, Test Case, or Test] from scratch or through cloning an existing one from the respective library.
- `CloneModal.jsx` (See Figure 5.8) Displays all the library [Test Plans, Test Cases, or Tests] such that users can choose one to clone from.
- `selectDevice.jsx` (See Figure 5.9) Displays all the devices from Device Library such that users can multi-choose device(s) they want to add to the corresponding BOM.
- `selectQuantity.jsx` (See Figure 5.10) Displays a selected list of devices and allows users to enter quantity desired (text-field) and whether it is optional (checkbox) for each entry.
- `DeleteModalForm.jsx` (See Figure 5.11) Allows users to confirm their

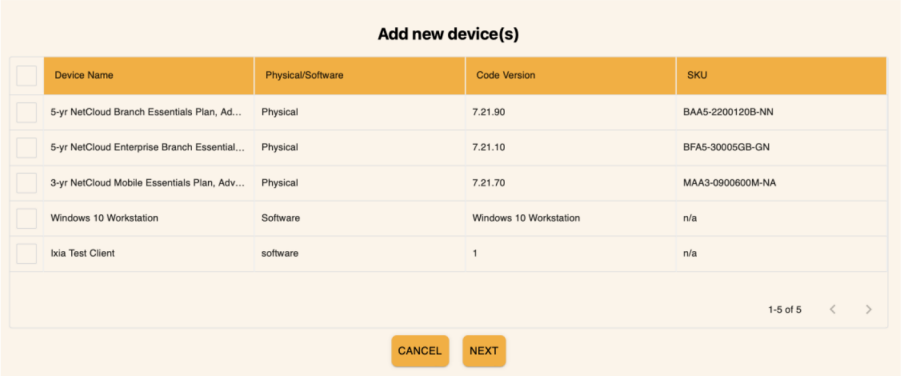

Figure 5.9 `selectDevice.jsx`

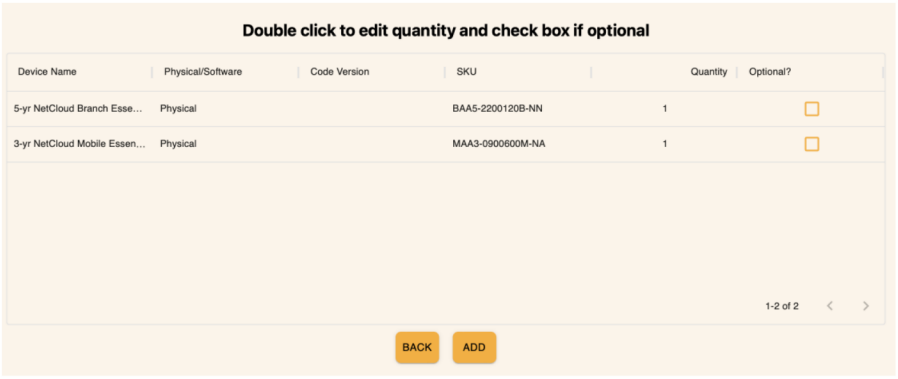

Figure 5.10 `selectQuantity.jsx`

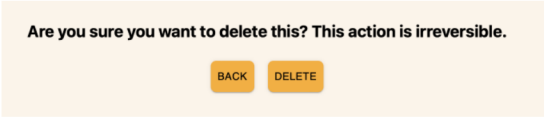

Figure 5.11 `DeleteModalForm.jsx`

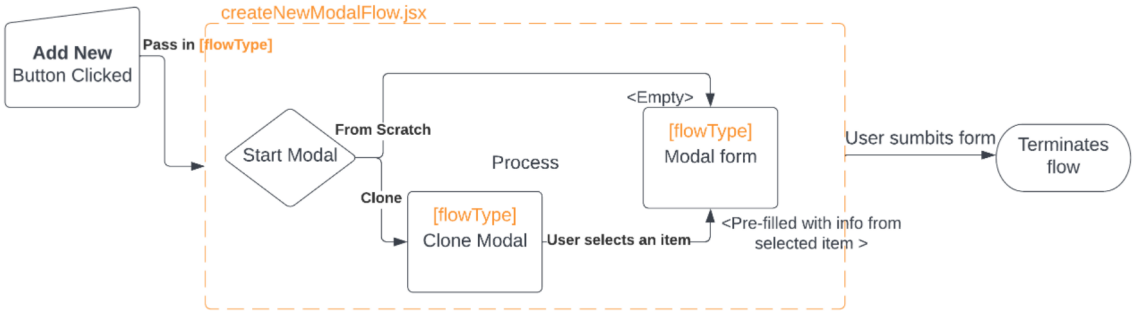

Figure 5.12 Create New Modal Flowchart

decision of deleting an/a Engagement, Test Plan, Test Case, Test, Result, or Device].

5.3.1 Create New Modal Flow

Recall that for Test Plan, Test Case, and Test, they each have a corresponding library that stores the pre-approved existing entries. Thus, when users want to create a new one, they have the option of creating from scratch or cloning an existing one from the library. Figure 5.12 shows the flowchart of the modal flow.

The corresponding details page contains a `CreateNewModalFlow` component which will be triggered once the user clicks on the "Add New" button. It will by default go to the Start Modal, passing in the `flowType` (e.g. Engagement), in which user can decide whether to create from scratch or clone. In the first case, an empty modal form of the corresponding entity will pop-up in which user can fill out the necessary information and submit. In the second case, a modal containing a table of all the library entries of that entity will show up. The user can choose one of them by clicking on the clone button and a modal form that is pre-filled with the information of the selected entry will show up. The user can edit the information and submit the form.

For other entities such as Engagements or Results, since there are no libraries, creating a new entity only has the create from scratch path.

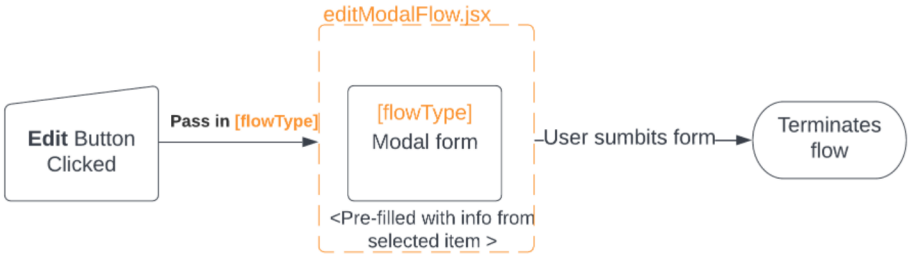

Figure 5.13 Edit Modal Flowchart

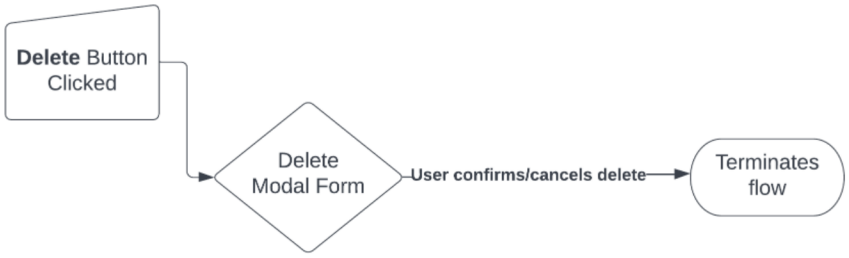

Figure 5.14 Delete Modal Flowchart

5.3.2 Edit Modal Flow

The edit modal flow applies to Engagement, Test Plan, Test Case, Test, and Result. Figure 5.13 shows the flowchart of the modal flow.

Each details page contains a `EditModalFlow` component. Once the edit button is clicked, the corresponding modal form will pop-up with pre-filled information.

5.3.3 Delete Modal Flow

The delete modal flow applies to Engagement, Test Plan, Test Case, Test, Result, and BOM Device. Figure 5.14 shows the flowchart of the modal flow.

Clicking on the delete button will trigger a `DeleteModalForm` which asks the user to confirm the deletion.

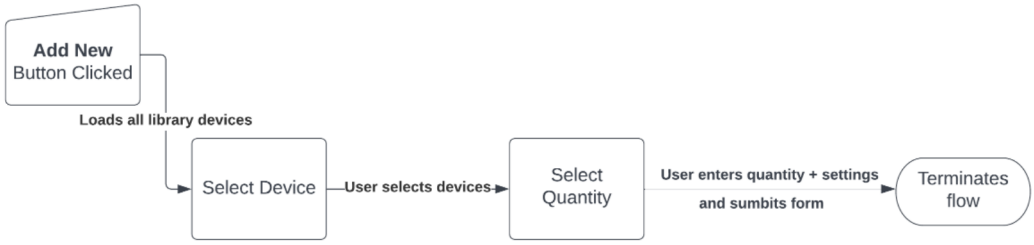

Figure 5.15 Add Device to BOM Flowchart

5.3.4 BOM-related Modal Flows

Since the nature of device and BOM is very different from the other entities (i.e. Engagement, Test Plan etc.), they have their own create new and edit flow (though they have the same delete flow).

Adding a device to BOM can only be done in the test case level. Thus, only the `TestCaseDetails` page contains the `SelectDeviceModal` and the `SelectQuantityModal`. Figure 5.15 shows the flowchart of the modal flow.

Once a user clicks on the Add New button, the `SelectDeviceModal` is triggered and will display a table of all the library devices. Once a user has selected all the devices they want to add to the BOM, clicking next will trigger the `SelectQuantityModal`. All the selected devices will be displayed in a table in this modal where users can edit the quantity and whether or not the device(s) are optional. The flow is terminated once the user submits the form.

Editing a type of device in BOM can also only be done in the test case level. Figure 5.16 shows the flowchart of the modal flow.

Once user clicks on the Edit button, the `SelectQuantityModal` is trigger and will display a table of the selected type of device, along with its current quantity and setting. The users can edit the quantity and whether the device(s) are optional. The flow is terminated once the user submits the form.

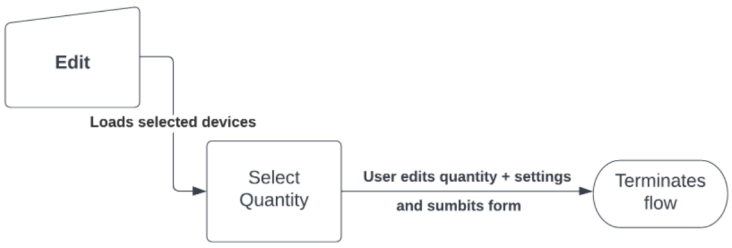

Figure 5.16 Edit BOM Device Flowchart

5.4 Conclusion

This chapter should make it more convenient for users to know where to modify or add new features. It is important to note that these elements are frequently interacting with the backend, mostly via the `serverSideProps` in each page and via submitting forms in the modals. For more details of the backend, refer to Chapter 3. At this point, we have covered most of the architecture of the website. In the following chapter, we move on to describing potential future work for the application.

Chapter 6

Future Work

While all the chapters prior describe how the application works as we've implemented it, this chapter instead outlines the work that could still be done in order to improve the application that we have already built. This work includes fixing bugs that are still present in the application, recommended code refactors, and the implementation of several new features.

6.1 Known Bugs

To the best of our abilities we attempted to fix bugs as we developed the web application. However, there were a few that escaped us in the midst of the time crunch. Below are some of the known bugs we have identified as well as some preliminary potential fixes that we have brainstormed:

1. **Bug:** Result status field for test does not update.
   Description: If a user edits the result status field of a result, the most recent result status filed of the parent test does not update automatically.
   Steps to reproduce: Edit the result status of a result object.
   Potential Fixes:
   - Create an additional field under the test schema (in the database) to keep track of the most recent object.
   - Add a query that retrieves the most recent result object.
2. **Bug:** Navigation directory is incorrect when the back button is used or the page is reloaded.
   Description: The navigation directory on the top left that tracks which

layer the user is in will display the incorrect directory when the page is reloaded or when the back button is used.
Steps to reproduce: Enter an engagement and click into a couple nested layers to build the directory navigator. Then either reload the page or use the back button to see the bug.
Potential Fixes:

- Reconstruct the navigation directory upon every page load, and not in a React state (as it is currently doing now) which loses its state when the page reloads.
- Do a check before every page render to make sure we do not lose the state of the navigation, repopulate the state upon a reload to avoid losing the information.

3. **Bug:** Buttons to add and view library items remain unimplemented.
Description: The add items to library APIs are unimplemented for Test Plan library, Test Case library, and Test Library in the backend. But in the frontend, it would look like it is working.
Steps to reproduce: If a user click on Add New button on one of the library pages, though it will let you go through the create new modal flow in the frontend, when trying to submit it, the backend does not actually call the correct api to add the item to library.
Potential Fixes:

- Implement the APIs: `addLibraryTestPlan`, `addLibraryTestCase`, `addLibraryTest`
- Either create a separate modal flow for adding to libraries, or update the `createModalFlow` so that it supports this functionality

6.2 Additional Data Validation and Testing

Within the docs directory of our project, there is a validation folder containing several json-like files that define MongoDB serverside validation rules, and instructions for how to update them.[1] Currently, these rules are set to only produce warnings instead of errors. MongoDB has documentation which gives more information on what can be done with their schema validation.[2]

[1]Instructions for validation: https://bit.ly/3KOZoI3
[2]Documentation: https://bit.ly/3MV9Dfa

While most of our testing focused on API's and the backend, we originally aimed to do automated Front-End testing with Selenium's Webdriver.[3] Utilizing Selenium would give us a way to test that each page has its required html elements working, such as buttons, modals, fields, and tables. We quickly realized that we would not be able to fully implement this kind of testing, however, if desired, Selenium could be utilized and implemented in the future to continue testing of this application.

6.3 Recommended API Changes

Towards the end of our project, we made several improvements to the various API calls to make them more RESTful, have better error handling, and allow for greater code reuse. These improvements have largely been made only to the edit and add API's, and need to be extended to the get, clone, and delete API's. It may be helpful referring to Section 3.2 for some of the API implementation details before reading the following recommended changes.

For the delete API's, we recommend creating a delete directory, and renaming each relevant API, i.e. `pages/deleteTest.js` to `pages/delete/Test.js`. We then recommend refactoring the delete functions in `utils/deleteEntry.js`, such that each await call has try/catch handling of errors, and returns the proper http response code. Currently, each of these functions is placed within one large try/catch block, and returns the same 500 error message when any of the awaits causes an error.

The clone API's are in a similar state, and we recommend changes identical to those for the delete API's, particularly, changing the names of files and refactoring util functions for better error handling. The get API's, again, require similar changes, however there is currently no util for these API's.

For many of the get, clone, and delete API's, they currently only display a generic 500 status error when something goes wrong, making them very difficult to debug. These changes make it significant easier to debug any problems. There is also significant code-reuse among the API's and having most of their logic all in one util file makes it easier to share common functionality.

[3] See: https://www.selenium.dev/

6.4 Recommended Schema Changes

This section details our recommend changes to the object schemas, that we did not have time to implement. Somewhat in order of increasing implementation difficulty, on the whole, these changes would requires significant refactoring to multiple files in the project on both the frontend and the backend.

1. **Change:** Remove redundant "createdOn" fields.
   Description: Most object schemas currently contain a field named "createdOn". According to the mongoDB Documentation, each ObjectId can use a method to get the timestamp from when it was created, making this field somewhat redundant, since the "_id" field essentially store this already.[4]
   Refactors: Remove the "createdOn" field from each schema in which it is located. Then in each relevant add API, use the ObjectId's getTimestamp method to add a field to the returned json response.

2. **Change:** Add a new "lastModified" fields.
   Description: This change proposes instantiating the lastModified field with this method, and updating it with a new Date() timestamp on each edit.
   Refactors: This change would require an addition to each schema of a "lastModified" field. Each relevant edit API's would be refactored to update the "lastModified" field of the returned json response. To display this field, changes would need to be made to each details page to ensure that it is properly displayed.

3. **Change:** Replace "comments" fields with "feedback" field.
   Description: Change the schemas with a String typed "comment" field to instead have an Object typed "feedback", consisting of entries of users, comments, and any nested replies.
   Refactors: This change is not likely to require any changes to current API's, but would require significant changing of each details page. These changes would likely include buttons (and potentially a new modal) to add, edit, and view comments.

[4]Documentation: https://bit.ly/3KSZXAI

6.5 Feature Extensions

We envision that the future work of our web application will feature development of new functionality in two key areas, the review and iteration process, and quality of life improvements. The features listed here will increase usability of the app and should bring it to a more production-ready stage.

6.5.1 Review and Iteration Process Extensions

It is intended that our application will be used to support the review and iteration process for creating and managing POC Engagements. This process involves having Cradlepoint's POC and SE engineers having access to edit and comment on each level of data, and be notified of any major approvals or changes to that data. To fully support this functionality, we recommend the following features to be implemented in the future:

- Creation of user accounts with differing level of permissions.
- User sign-in and logging of changes.
- Assigning tasks to different users along with a notification system to keep track of these tasks.
  - Creation of new API's to allow for email alerts.
  - Integration with Engagement and Testplan status, to send alerts for change status approvals.
- Flesh out success criteria of Test Plans and Test Cases based on the result of tests.
- Creation of more extensive comments, see Section 6.4 for more details.

6.5.2 Quality of Life Extensions

Moving forward, there are a number of additional features we believe would significantly add to the user experience and usability of the web application. Below is a list of these features, in no particular order:

- File Upload and Download buttons for specific object fields such as topologies and configs.

- Export as PDF Button for Engagements to produce a client-friendly summary document.
- Additional Export Json Buttons for TestPlans, TestCases, and other levels.
- View and Edit Buttons for entries in each of the library pages.
  - Extend Edit modals to libraries and potentially create a view button and page to see the lower hierarchy objects attached to a library object.
- A general search feature, to find objects by Name or other fields.
  - Add a specific "tags" field to objects that can be used to find them with the search feature.
- API integration with SalesForce to allow for scheduling and tracking of information through the platform.
- Implement Versioning and Archiving to allow for storage, reuse, retrieval, and backing up of old data.