Hindia Mohammed

STORAGE PORTAL

A React-Redux Web Application Utilizing Polarion for Test Data Management

Technology and Communication
2020

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Hindia Mohammed
Title	Storage Portal: A React-Redux Web Application Utilizing Polarion for Test Data Management
Year	2020
Language	English
Pages	40
Name of Supervisor	Timo Kankaanpää

This thesis project is a product of the Thermofluids and Simulation (or CFD) team in Wärtsilä. The team has had to undergo a painful and time-consuming process of acquiring test data – and in turn boundary values – to perform a data analysis and simulation. As a result, it proposed to design and implement an application, i.e. Storage Portal, that accepts test data complying with the ASAM ODS standard format and direct it to Polarion. This in turn allows automatic querying and centralization of test data.

The project is a client-server solution. The client side was built using React with Redux managing the state of the React components. A Polarion project was created in the internal network following the ASAM ODS standard to receive the data coming from the client. The client and the server interact using Polarion's API and their communication follows the SOAP protocol.

The application created functions that were highly useful and necessary for automating the querying of test data in Polarion by acting as an interface between the test runner and Polarion. The data is automatically filtered using Polarion, which has proved logging and querying test data to be an effortless, centralized and standardized task.

Test data management is facilitated and half-automated by this application. The system benefits from existing and new technologies to provide the presented solution.

Keywords	Storage Portal, React, Redux, SOAP, ASAM ODS, Polarion

CONTENTS

ABSTRACT
ABBREVIATIONS 5
LIST OF FIGURES AND TABLES 7
1 INTRODUCTION 8
1.1 Wärtsilä 8
1.2 Problem Statement 9
1.3 Objectives 10
1.4 Implementation Plan 10
2 TECHNOLOGIES AND TOOLS 12
2.1 React 12
2.2 Redux 12
2.3 Polarion 13
2.4 SOAP 15
2.5 Jest 15
2.6 Enzyme 16
2.7 VS Code 16
2.8 Bitbucket 16
2.9 Kubernetes 17
3 SYSTEM DESCRIPTION 18
3.1 Software Requirements Specification 18
3.2 BPMN 20
3.3 Use Case Diagram 20
4 APPLICATION DESIGN 22
4.1 Software Architecture 22
4.2 Package Diagram 23
4.3 Sequence Diagrams 23
4.3.1 Add Test Point 23
4.3.2 Edit Test Point 24
4.3.3 Delete Test Point 25
4.3.4 Add Test Campaign 25

4.4 Class Diagram 25
5 SOLUTION IMPLEMENTATION 28
5.1 Development 28
5.1.1 Storage Portal 28
5.1.2 Performance Data Management 29
5.2 Data Management 29
5.3 Testing 34
5.4 Deployment 35
6 CONCLUSION 38
7 FUTURE WORK 39
REFERENCES 40

ABBREVIATIONS

ACCDOM	ACCount DOMain
ADAL	Active Directory Authentication Library
ALM	Application Lifecycle Management
API	Application Programming Interface
ASAM	Association for Standardization of Automation and Measuring Systems
AWS	Amazon Web Services
AZ	Availability Zone
BPMN	Business Process Modeling Notation
CFD	Computational and Fluid Dynamics
CD	Continuous Development
CI	Continuous Integration
CSS	Cascading Style Sheet
Corp	Corporation
DevOps	Development and Operations
EC2	Elastic Compute Cloud
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
JS	JavaScript
KOPS	Kubernetes Operations

LG Large

MS Microsoft

MPS Marine and Power Solution

NOx Nitrogen Oxides

NPM Node Package Manager

ODS Open Data Services

OS Operating System

PC Personal Computer

SOAP Simple Object Access Protocol

SRS Software Requirement Specification

UI User Interface

UML Universal Modeling Language

UPN User Principal Name

URL Uniform Resource Locator

VCS Version Control System

VS Visual Studio

WS Web Service

WSDL Web Services Description Language

XL Extra Large

XML Extensible Markup Language

LIST OF FIGURES AND TABLES

Figure 1. Google trends data for JS framework and libraries. /5/ 13
Figure 2. State management with Redux /6/ 14
Figure 3. Polarion's ALM /7/ 14
Figure 4. BPMN Diagram 21
Figure 5. Use Case Diagram 21
Figure 6. Client-Server architecture 22
Figure 7. Package Diagram 24
Figure 8. Adding Test Point 24
Figure 9. Editing Test Point 25
Figure 10. Deleting Test Point 26
Figure 11. Adding Test Campaign 26
Figure 12. Class Diagram 27
Figure 13. GUI of Storage Portal 29
Figure 14. Test Point Custom Fields 30
Figure 15. Work Items in Polarion 30
Figure 16. Login Form Schema 30
Figure 17. Required Fields Error Message 31
Figure 18. Displaying Test Points from *store* 32
Figure 19. Edit Mode of Record Component 32
Figure 20. Editing Test Points in *store* 32
Figure 21. Polarion Documentation for *logIn* API function 33
Figure 22. *workitem* Constructor 33
Figure 23. SOAP call using jQuery *soap* 34
Figure 24. Jest test log 35
Figure 25. Testing reducer for ADD_PATH_TO_FILE action type 36
Figure 26. Commit history 37

Table 1. Functional Requirements 18
Table 2. Non-functional characteristics 19

1 INTRODUCTION

This project aimed to build a web-based client-server application for Wärtsilä's MPS department, which embraces CFD and Performance teams. The application proposed by the CFD team will enable to log test data into Polarion. Then using Polarion, performance team members, who have permission to access data, can filter test data necessary for analysis.

The application was mainly built with JavaScript libraries named React, Redux and Soap. The server side of the application, a Polarion webservice, is already implemented in Wärtsilä's network. Hence, the application utilized the Polarion API and WSDL provided by Polarion to send and receive data using the SOAP protocol.

Furthermore, a Polarion project was created with a framework to receive the data coming from the client application. This project lets the application create a new work item every time a form using **Storage Portal** is submitted.

After this chapter introduces the project in more details, Chapter 2 follows describing the technologies and tools used to design, develop, implement, test and deploy the application. The requirements specification of the project, which has been agreed upon by the client and the developer, is laid out in Chapter 3 with the aid of diagrams. Chapter 4, then, introduces the technical details of the system. It unfolds the sequence diagram, package diagram and the class diagram of the client software. The solution implementation of the system from the development stage to the deployment stage is discussed in fine details in Chapter 5. The last chapter concludes the paper with raising the results, the limitations, the future and the suggested improvements to the application.

1.1 Wärtsilä

In 1834, after the governor of the county of Karelia approved the construction of a sawmill in the municipality of Tohmajärvi on 12th of April, Wärtsilä was established. It has now become a leader in providing smart technologies and complete lifecycle solutions for the international market in the marine and energy industries.

Wärtsilä delivers vessels and power plants with maximised environmental and economic performance. It achieves this by emphasising sustainable innovation, total efficiency and data analytics. /1/

In 2018, with approximately 19,000 employees working in more than 80 countries, Wärtsilä's net sales totalled 5.2 billion Euros. The company operates in over 200 locations. /2/

As of January 2019, Wärtsilä consists of two businesses: Marine Business and Energy Business. System Analysis team, Performance team and Assembly and Test Run team are of the organizations found in Marine Power Solutions (MPS) department which lies under Marine Business. To provide most competitive offering in Marine and Energy markets, these teams work very closely to develop the right technologies, products and integrated solutions. Thermofluids and Simulations (or most commonly known as CFD) is one of the many units under System Analysis team.

1.2 Problem Statement

The three teams mentioned earlier share test data very often. Whenever the System Analysis team requires test data, which has been recorded by the Assembly/Test Run team, the Performance Engineers are the best colleagues to identify useful data for the experts in the Analysis team. Hence, the performance team is tasked with the allocation of the important test points. However, data allocation for the Performance team has been very daunting and an unnecessarily difficult task, as there is no naming convention in place nor a central database system for all the test points required by the Analysis team.

Moreover, legacy data is spread everywhere but would be very insightful if machine learning algorithms were applied to those legacy data sets to train models and predict engine behaviors, and to reduce NOx emission.

1.3 Objectives

The objective of this thesis was to design and implement a web application that takes test campaigns, with test points carried out during engine tests, from the test runner and sends them to Polarion.

In the back end, the project created in Polarion was expected to receive the validated data from Storage Portal the client application. The Polarion project was necessary to allow querying and centralizing test data. Its custom fields were planned to be named in accordance with the ASAM ODS standard. The standard was created to simplify the universal interpretation of data acquired from testing, evaluation, and simulation applications. /3/

1.4 Implementation Plan

In order to meet the objectives of the thesis project, the following implementation plan has been laid. It contains a set of expectations to be met by the project.

The first on the list was having close communication with the stakeholders, i.e. the CFD team, Performance team, Wärtsilä's Polarion experts, Wärtsilä's cloud solution experts and the thesis supervisor. This was to ensure the application was designed in the way the performance team is going to find it valuable. The GUI, naming of fields and lists, access groups, technologies and tools (framework, components, test environment, server, VCS, CI/CD pipeline, authentication method) and milestones were agreed upon during online or/and physical meetings. The meetings were expected to be held at least once a week within the CFD team, once every 2-3 weeks with the thesis supervisor and whenever necessary with the other stakeholders involved.

Since there are already too many applications the Performance team uses, however advantageous Storage Portal maybe, the project needed to demonstrate its use case. This was planned to be proven by manually entering a portion of a few test campaigns from the database, which is not following any standard and is difficult to query data from, into Performance Data Management – a Polarion project created

for accepting data from the Storage Portal with fields following the ASAM ODS standard.

Technically, the project was expected to be setup using create-react-app *node package*. The application being a React application with its *state* managed with Redux. Upon landing on the URL, the user will be redirected to an authentication page implemented by Wärtsilä to verify permissions. After logging into Storage Portal, the user inputs data through the *input fields* provided. The user will also be able to pick data, extracted from Polarion and *settings.json* file, for fields that are not input field. Data validation was proposed to be performed with JS libraries Formik and Yup. These supporting JS libraries were planned to be downloaded using NPM by installing NodeJS.

At any point, Workitem.js class will be subscribing to the Redux store. Therefore, when the user submits the Test data, soap function will be called with the workitem object, which is created with the latest state. The soap function invokes Polarion's API function createWorkItem. This will then create the Test Campaign, which can be accessed inside Polarion in Performance Data Management.

The client was planned to be developed using VS Code with Prettier as a linting tool. Its version controlled with Bitbucket; it will go through the CI/CD pipeline in Bamboo while being inspected by SonarQube for code quality. The test environment was set to be Jest with Enzyme. Since Storage Portal is a stateless application with no data persistence, it was planned to be containerized with Docker**.** The Docker container was then expected to be deployed on Wärtsilä's Kubernetes services.

2 TECHNOLOGIES AND TOOLS

In this section, the main and subsection-worthy technologies and tools used in developing, implementing, testing and deploying the application are introduced. The reasons for choosing them is also discussed.

2.1 React

React (also known as React.js or ReactJS) is a JS library for building user interfaces. It is maintained by Facebook, a community of individual developers and companies.

As can be seen on its GitHub page, it currently has been used 3.1 million times, is watched by 6.6 thousand GitHub accounts with 143.2 thousand stars, 124 releases and 1.3 thousand contributors. /4/

React makes building interactive user interfaces effortless by using condensed React Components that can be built to manage their own state. These React Components are simply either JS classes or functions that return a description of how a UI should appear. React web applications can also render new data without reloading the page.

For this project React was chosen, over other JS frameworks, because it was proven to be much better than the other two popular ones, depicted in the Figure 1 below, in terms of speed, efficiency, lightweight components, simple API, and overall performance.

2.2 Redux

Redux is an open-source JS library for managing application state. It is most commonly used with libraries, such as React or Angular for building user interfaces. It helps to write applications that behave consistently, run in different environments and are easy to test.

Simply put, it helps to manage the data using Actions, which specify the changes that need to happen, Reducers, which are functions that define how every action

alters the state of the Store, and a Store, which as the name suggests contains the state of the entire application. Figure 2 demonstrates this paragraph.

The test data inputted by the user using Storage Portal is saved and managed in a Redux state before being sent to Polarion.

2.3 Polarion

Polarion is an end-to-end, enterprise-grade and browser-based application lifecycle management platform.

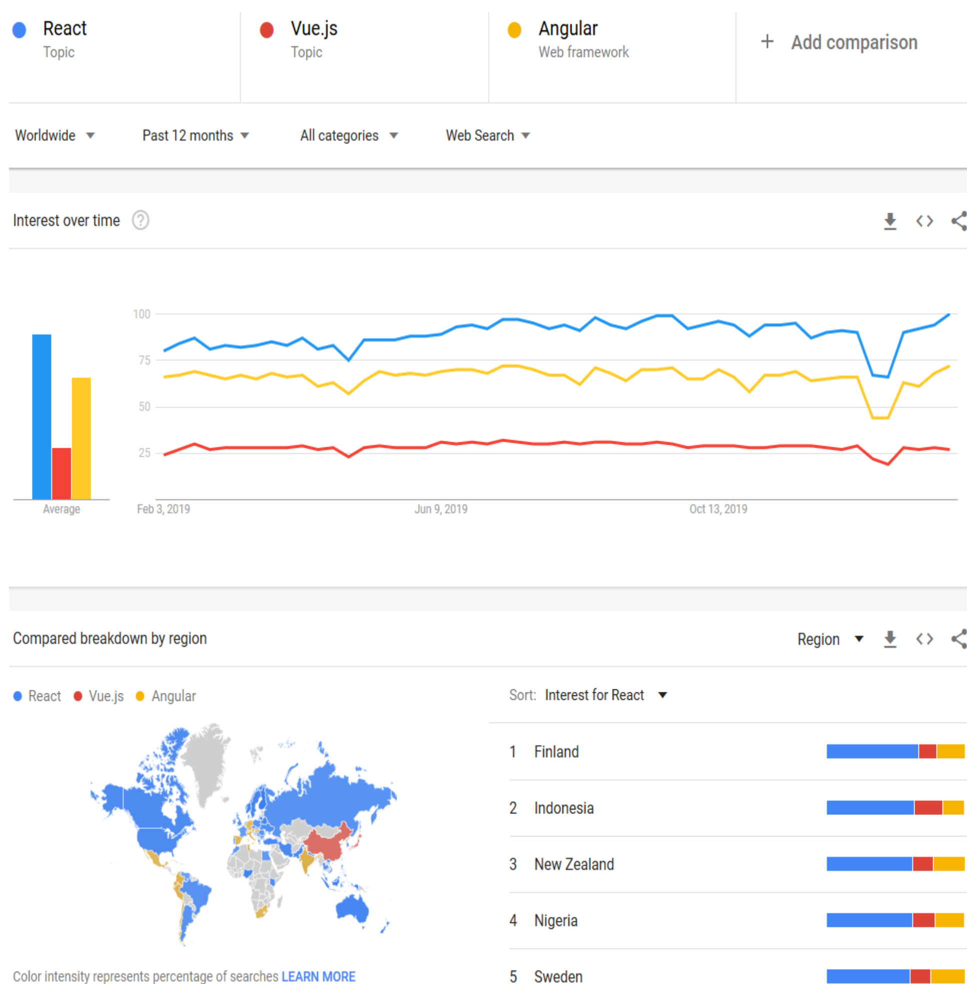

Figure 1. Google trends data for JS framework and libraries. /5/

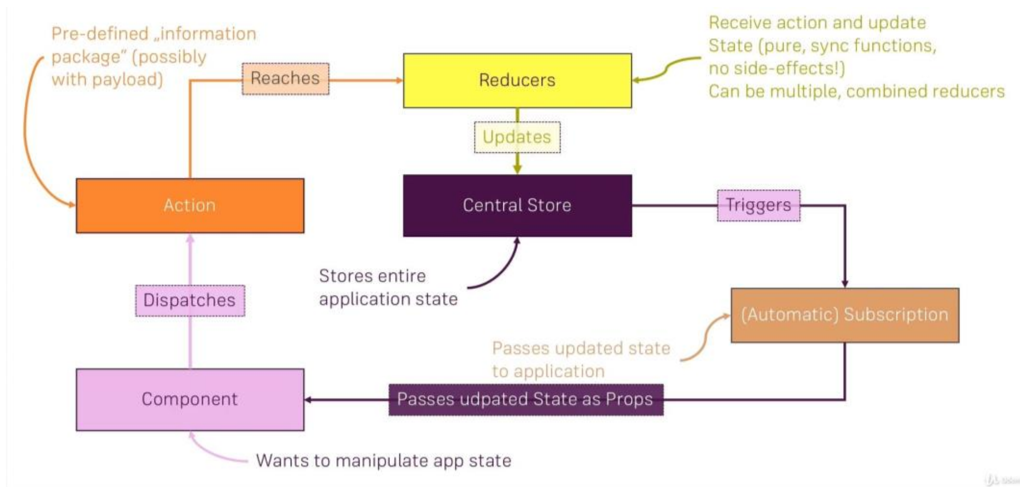

Figure 2. State management with Redux /6/

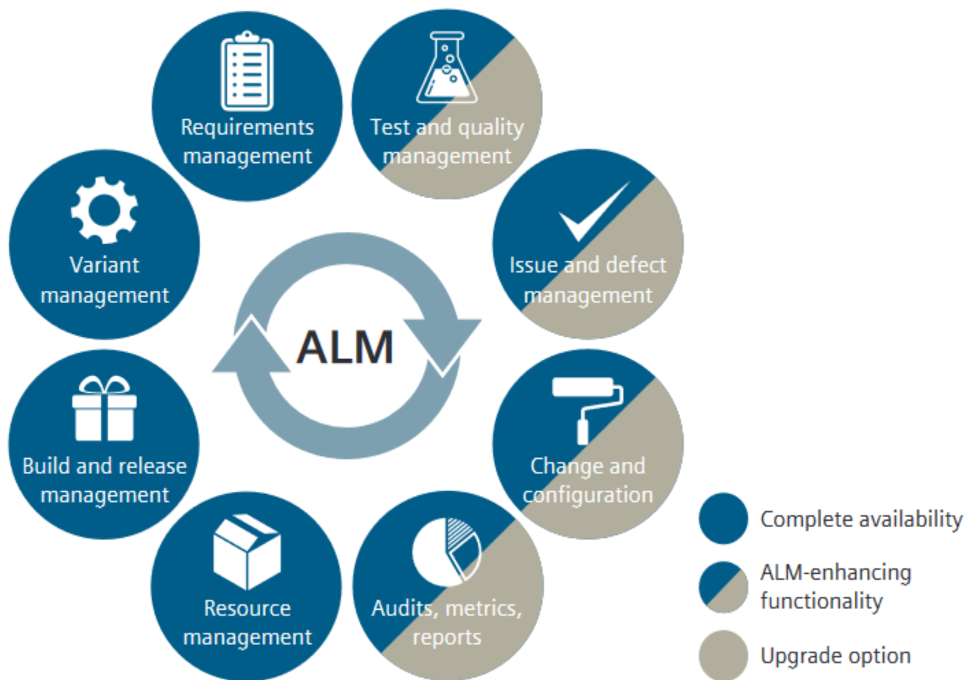

Figure 3. Polarion's ALM /7/

Managing some specific development activity throughout its life cycle is performed by collecting artefacts and data, which reside on the system in a repository folder. This folder is called a Project.

The general term applied to granular artefacts that need to be worked on or processed by people through a workflow and lifecycle is called a Work Item. Work Items most often represent things such as requirements, tasks, change requests, test

cases, and defects/issues. However, it is possible to define Work Item types to represent anything that needs to be tracked and managed through a workflow-controlled process.

The Projects and Work Items in Polarion can be accessed using Polarion's API, which allows interaction with its web service.

Polarion is in use at Wärtsilä and is advantageous to utilize its web service for this project. The client machine can fetch data from Polarion and link Work Items. Furthermore, the user can pull data using Polarion's built-in query function from the GUI.

2.4 SOAP

SOAP is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on application layer protocols, most often HTTP.

SOAP allows developers to invoke processes running on different OSs to authenticate, authorize, and communicate using XML. Since Web protocols such as HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms.

SOAP also supports WS-Security for enterprise-level protection. When dealing with crucial private information, which is the case for this project. it is very practical to use SOAP. Communicating with Polarion requires SOAP as of 2020 hence it was necessary to work following this protocol.

2.5 Jest

Jest is one of the most popular JS test frameworks with 29,532 stars on GitHub at the time of the writing of this thesis. It was built and is continually being maintained by Facebook. Recommended by React, it comes with its own valuable assertion library. It is easy to use and requires zero configuration.

Storage Portal is bootstrapped with Create React App which uses Jest as its test runner. Besides this fact and the details mentioned above which points towards its performance and convenience, Jest has been selected to be the test framework for the client application.

2.6 Enzyme

Enzyme is a JS testing utility for React to test output of a React component. Some React components are more worthy of testing than others based on what one wants to accomplish with an application. In this kind of scenarios, when performing isolated unit tests, we use testing utility libraries.

Enzyme is a test runner and assertion library agnostic. In addition, it allows reading and setting states of a component as well as mocking children components. It is well documented and recommended by Jest hence it is very sensible to reap its benefits for shallow and in-depth testing of Storage Portal.

2.7 VS Code

VS Code is a source-code editor developed by Microsoft for macOS, Windows as well as Linux machines. It supports debugging, Git control, syntax highlighting, intelligent code completion, and code refactoring.

However, the main reason for the preference is that it mainly supports React IntelliSense, out of the box, which includes code completion and hinting. Additionally, it supports Node.js development and allows JS extensions, such as linters. This makes it convenient and quick to develop clean code for the client application.

2.8 Bitbucket

Bitbucket is a web-based version control repository hosting service for source code and development projects. It was acquired by Atlassian in 2010 and integrates other Atlassian software such as Jira, Confluence and Bamboo. Moreover, it comes with extensions such as integrated CI/CD, which simplifies building the CI/CD pipeline.

Wärtsilä has been using Bitbucket as a VCS. This, coupled with the benefits mentioned earlier, is enough reason to choose Bitbucket as a VCS to control the source code.

2.9 Kubernetes

Containers are a good way to bundle and run applications. In a production environment, the containers that run the applications must be managed and ensure that there is no downtime. Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation. /8/

The Docker container of Storage Portal is deployed on Kubernetes via KOPS. The Kubernetes is running on AWS on EC2 instances. It is chosen to be deployed with KOPS due to stability, support for multiple-AZ master nodes and ability to use native network.

3 SYSTEM DESCRIPTION

As mentioned above the application aims to solve the problems in the implementation plan. To develop the software system, one should have clear understanding of what the user expects the functions of the software to be. In addition, clearly defined requirements are essential signposts that leads to a successful project. This chapter covers the agreements between the stakeholders.

3.1 Software Requirements Specification

This subchapter describes the characteristics that this product must have to meet the needs of the stakeholders and the business itself. SRS is a detailed description of a software system to be developed with its requirements based on the agreement between the user and the software developer. It constitutes of all the necessary requirements essential for the project development.

These requirements are divided to Functional, which describes what the features and functions of the product are, and Non-functional, which describes the general characteristics of the system.

Table 1 specifies the functional requirements of the system as expected by the user. Table 2 specifies the Non-functional requirements of the system as expected by the user.

Table 1. Functional Requirements

Reference	Description	Priority
F1	Fetching Test Requests and Work Requests from Polarion	1
F2	Fetching Engine Types, Test Types and Test Platforms	1
F3	Allowing navigation between Test Points	1
F4	Displaying recorded Test Points dynamically	1
F5	Allowing editing Test Points	1

F6	Allowing deleting Test Points	1
F7	Contains text editor for Test point Info and Test description	3
F8	Verifying ACCDOM users	1
F9	Adding the new Test Campaign to Polarion	1
F10	Allowing multiple Test Requests, Work Requests and File Paths	1

Priority levels:
Must have–1
Recommended to have–2
Nice to have–3

Table 2. Non-functional characteristics

Characteristic	**Description**
Usability	The UI is optimized so the user should never have to think how to operate with the application.
Safety	Only authorized personnel can operate the system, the user has to login to use the application.
Response time	The system should respond to every command in 2 seconds.
Aesthetic	The UI should be built to be appealing and clean following Wärtsilä's design guidelines.
Maintainability	The system should be maintainable by Wärtsilä's developers hence it should be well documented.

3.2 BPMN

BPMN is a graphical notation used for business process modeling. We use BPMN to draw business process diagrams. These diagrams present the activities and tasks of a process and their relations. The diagrams use flowchart concepts to represent the logic of business processes. /8/

In BPMN, Circles represent a trigger that starts, modifies or completes a process. An activity or task performed by a person or system is shown by a rectangle. Sequence Flow is represented by a straight line with an arrow. It shows the order of activities to be performed. If the flow is across organization boundaries such as departments. It is represented by a dashed line with an arrow at the end. The BPMN for this project is depicted in Figure 4.

3.3 Use Case Diagram

UML is a modelling toolkit that can be used to build diagrams. In UML, a use case diagram can summarize the details of a system's users (also known as actors) and their interactions with the system. Use cases are represented with labelled oval shapes. Stick figures signify actors and a line between the actor and use case models the actor's participation in the system.

The use case diagram for this project is depicted in Figure 5. As can be seen, there are two targeted actors in this system. The Mechanic (or sometimes Test Engineer) is the individual who uses Storage Portal to log data related to test. The Performance Engineer is the individual who uses Polarion to query the test data but is also able to, in some scenarios, log test data using the Storage portal.

There are six self-describing use cases as shown above. The main use cases are, however, *Add Test Campaign* and *Find Test Campaign.*

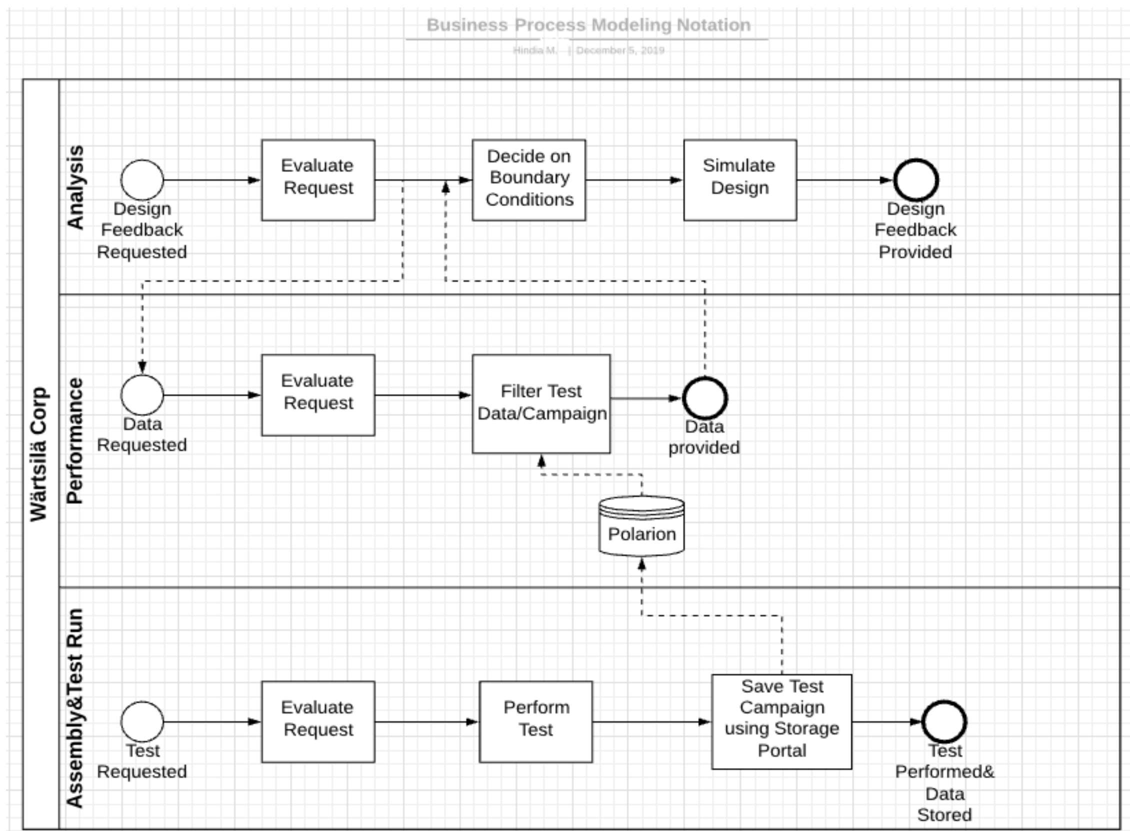

Figure 4. BPMN Diagram

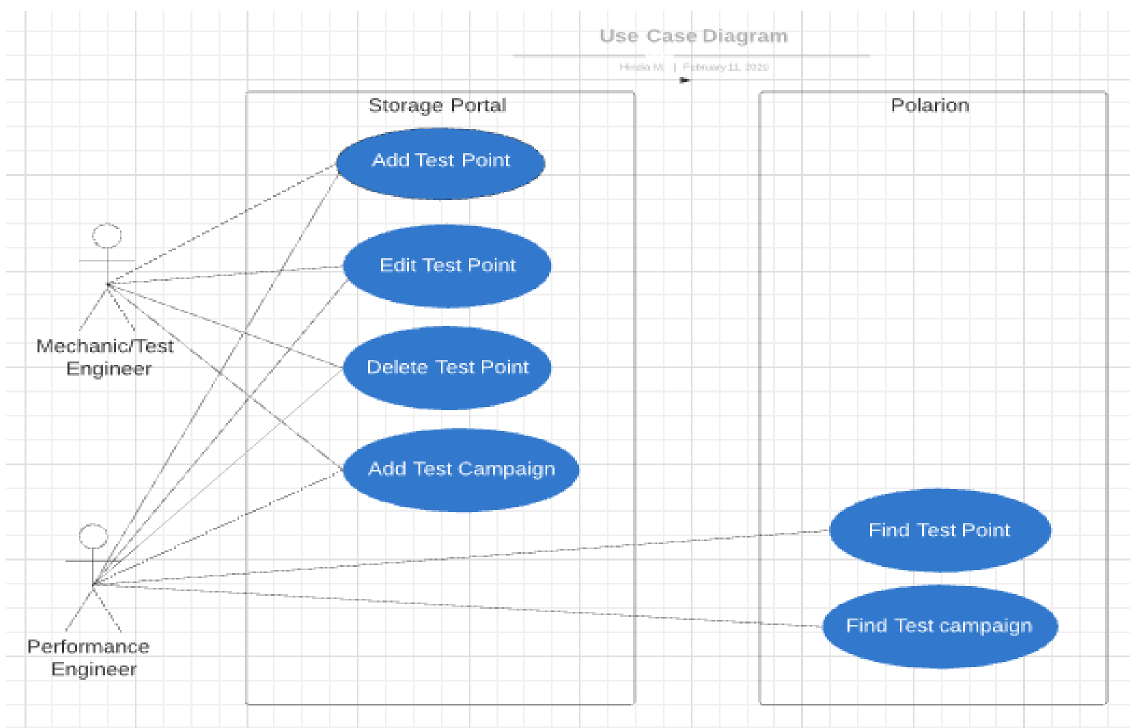

Figure 5. Use Case Diagram

4 APPLICATION DESIGN

In this chapter, the abstract overview of the application architecture, design and sequence is provided. It uses figures to demonstrate how the application appears for the user and how the application works in the logical realm.

4.1 Software Architecture

Software architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them.

As mentioned, this system has a server, Polarion's webservice, and a client, Storage Portal**,** involved. This type of software architecture is identified to be Client-Server model.

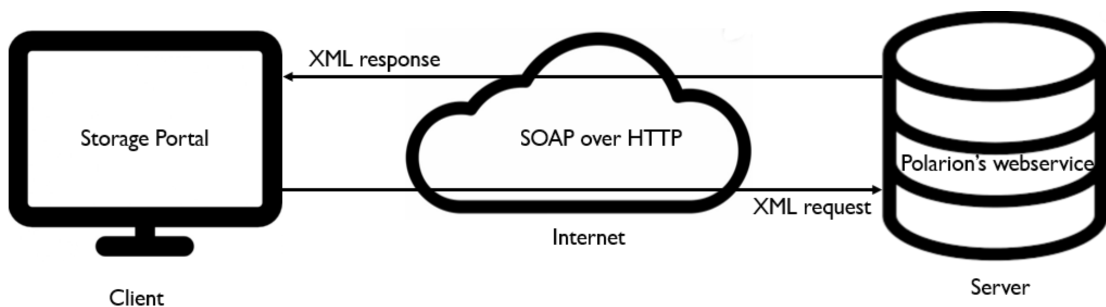

Figure 6. Client-Server architecture

The Client and server exchange messages in a request–response messaging pattern. The client sends a request, in an XML message format, and the server returns a response, in an XML message format. To communicate, the computers must have a common language, and they must follow rules so that both the client and the server know what to expect. The language and rules of communication are defined in the WSDL, which is an XML-based interface description language that is used for describing the functionality offered by a web service, of each web service provided by Polarion. To formalize the data exchange even further, Polarion implements an API. The main webservices used by the Storage Portal are Project webservice, Tracker webservice and Session webservice.

Storage Portal follows the React-Redux architecture. It contains three Container (smart) Components, which manage data and performs functions, and one Presentational (dumb) Components, which deal solely with the UI.

4.2 Package Diagram

A package diagram depicts the dependencies between the packages that make up the application. Packages, a collection of logically related UML elements, appear as rectangles with small tabs at the top left. Elements inside a package are denoted by rectangles. A private element, denoted by '-', is not visible at all to elements outside the package. Dotted arrows depict dependencies. The arrow points towards the element/package, which can bring an effect.

As shown in the diagram in Figure 7, the main packages of the client application are assets, components, containers and store. These packages and the main elements of the application, such as *soap.js*, *workitem.js*, have the relationship portrayed with third party packages.

4.3 Sequence Diagrams

In UML, an interaction diagram, which presents how and in what order operations are carried out, is known as a Sequence Diagram. It represents specifics of a UML use case to show how objects/classes - in this case React Components - interact with each other to complete a process. In this section, the key use cases of the application are illustrated using sequence diagrams.

4.3.1 Add Test Point

One of the main actions the user takes is to add test points. After adding the test point, the user expects the data to be displayed in the Records section of the UI within a table. This process is depicted in Figure 8.

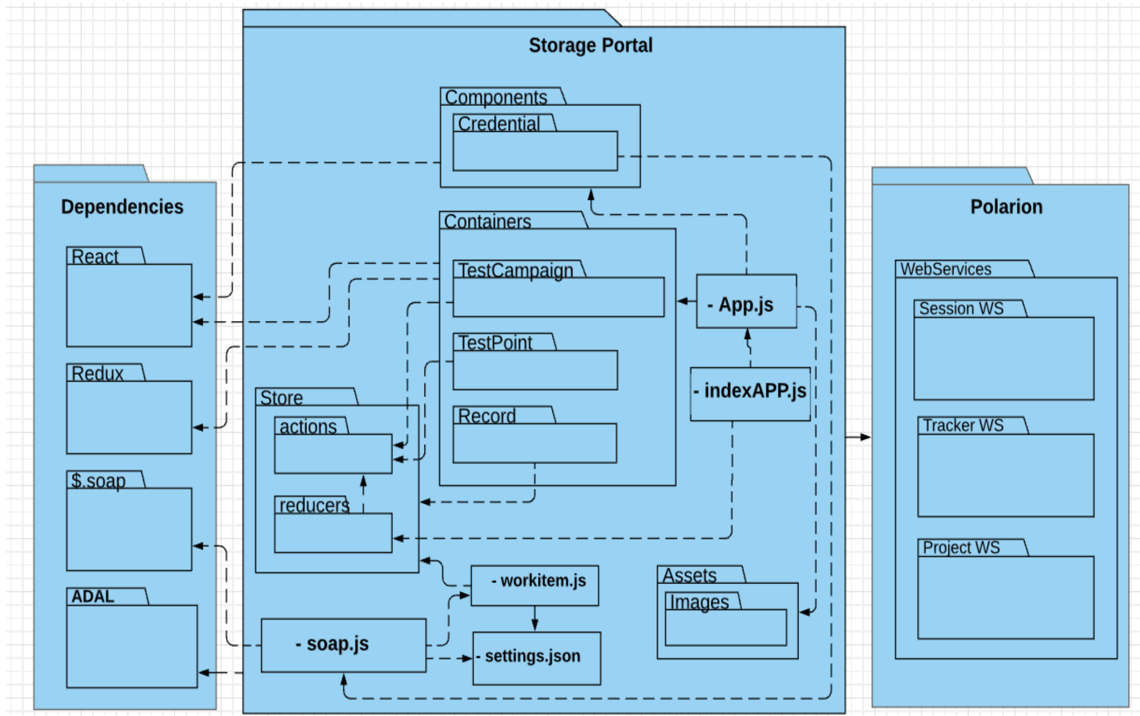

Figure 7. Package Diagram

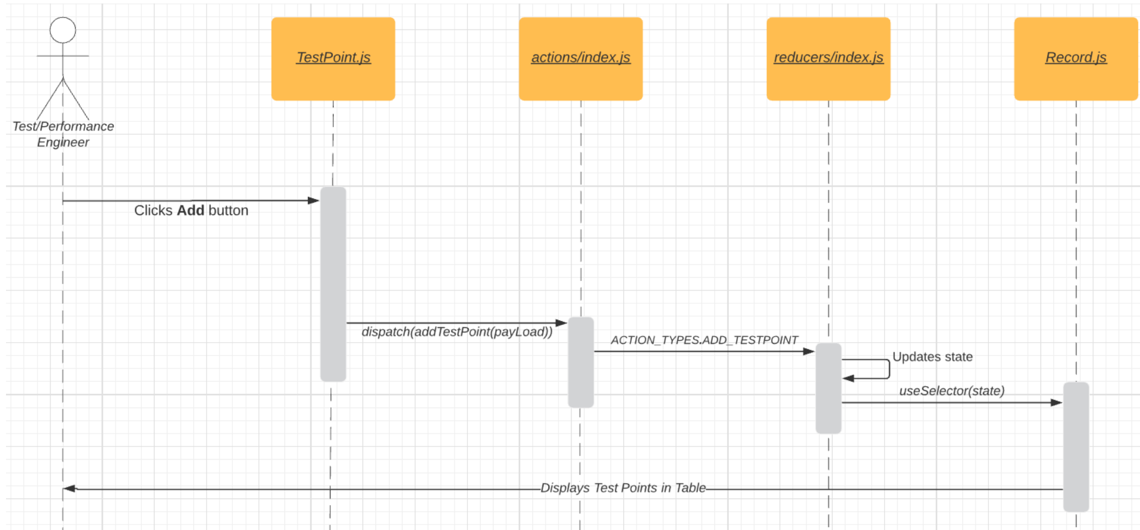

Figure 8. Adding Test Point

4.3.2 Edit Test Point

After the user observes the recorded test points and decides to adjust the data in the table, the user clicks on the edit icon found in every row for each entry of test point.

The consequence of that action begins sequence of events that are captured in Figure 9 below.

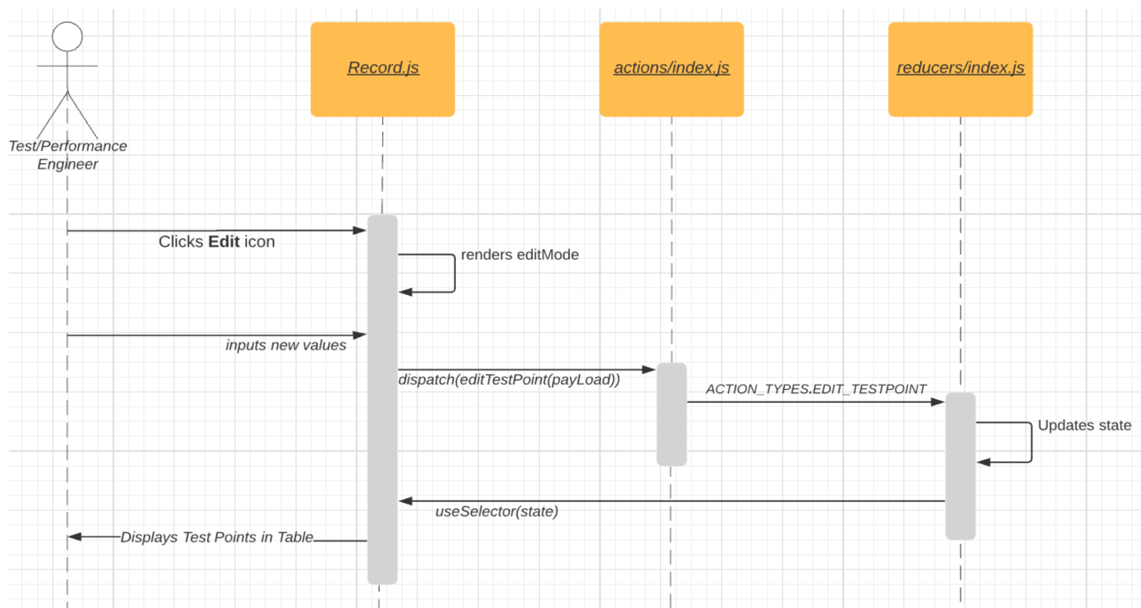

Figure 9. Editing Test Point

4.3.3 Delete Test Point

When the user decides to delete a test point, which is not sent to Polarion, the user clicks on the delete icon found in every row for each entry of test point. Upon that action, a sequence of events that are illustrated in Figure 10 below start to unfold.

4.3.4 Add Test Campaign

The application communicates with WS of Polarion to create work items by submitting the values the user has inputted. This in turn produces a Test Campaign. *Add Test Campaign* is the use case, which encapsulates this process. The sequence diagram in Figure 11 describes this process.

4.4 Class Diagram

In software engineering, a class diagram in UML is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

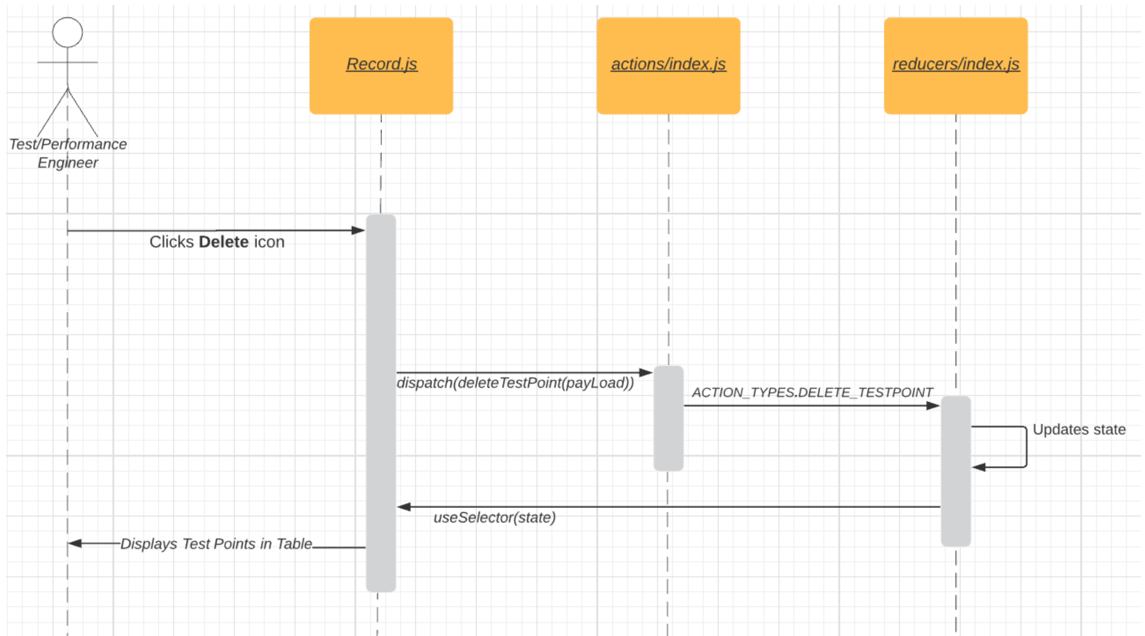

Figure 10. Deleting Test Point

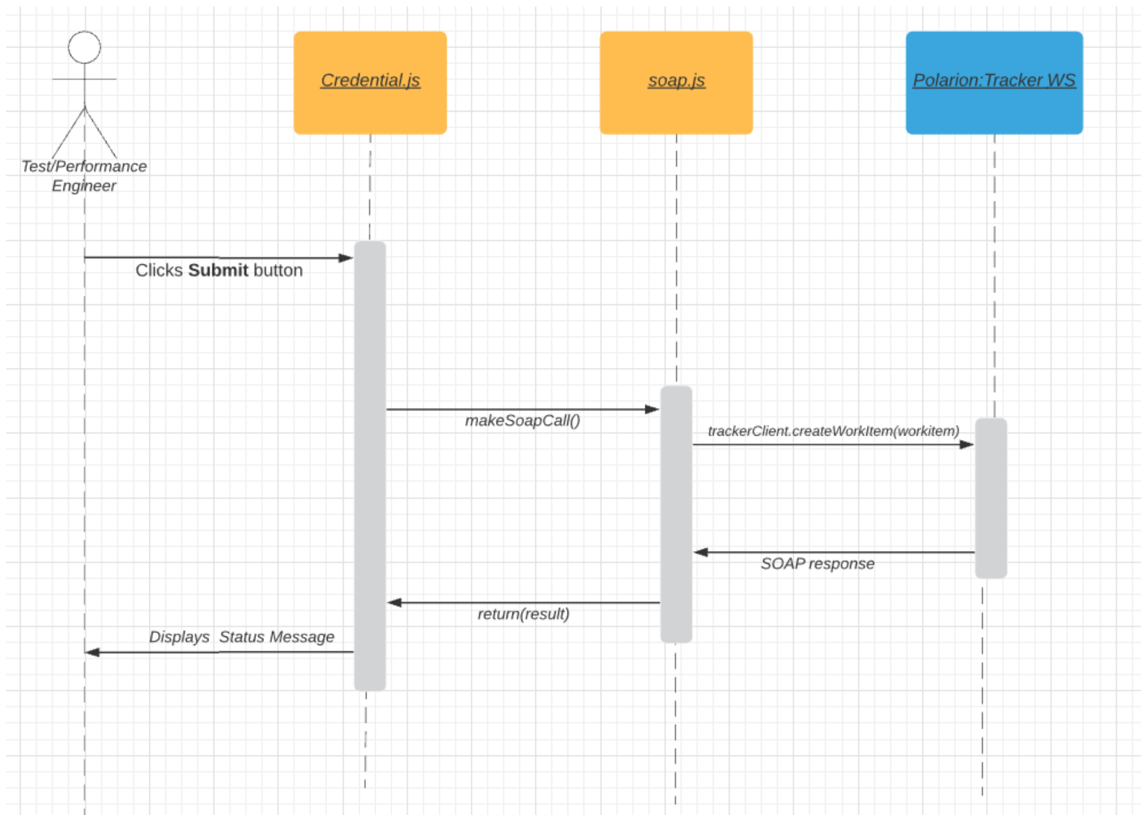

Figure 11. Adding Test Campaign

The Components used in this project are *Functional Components*, however, they are treated as Classes. The class diagram design is adapted from the developer Ashley Kitson with the following core principles:/10/

- Dashed two-headed arrow indicates that the linked components share the store.
- Parent and child components are linked with solid lines. The parent being denoted with an arrowhead.

As can be seen in Figure 12, the true source of state is provided from the Redux store. It is updated using action dispatchers in React Components. However, *workitem* module, which is not a React Component, subscribes to the store to create the work items, which will then be used by the Credential component when making a *soap* call to Polarion.

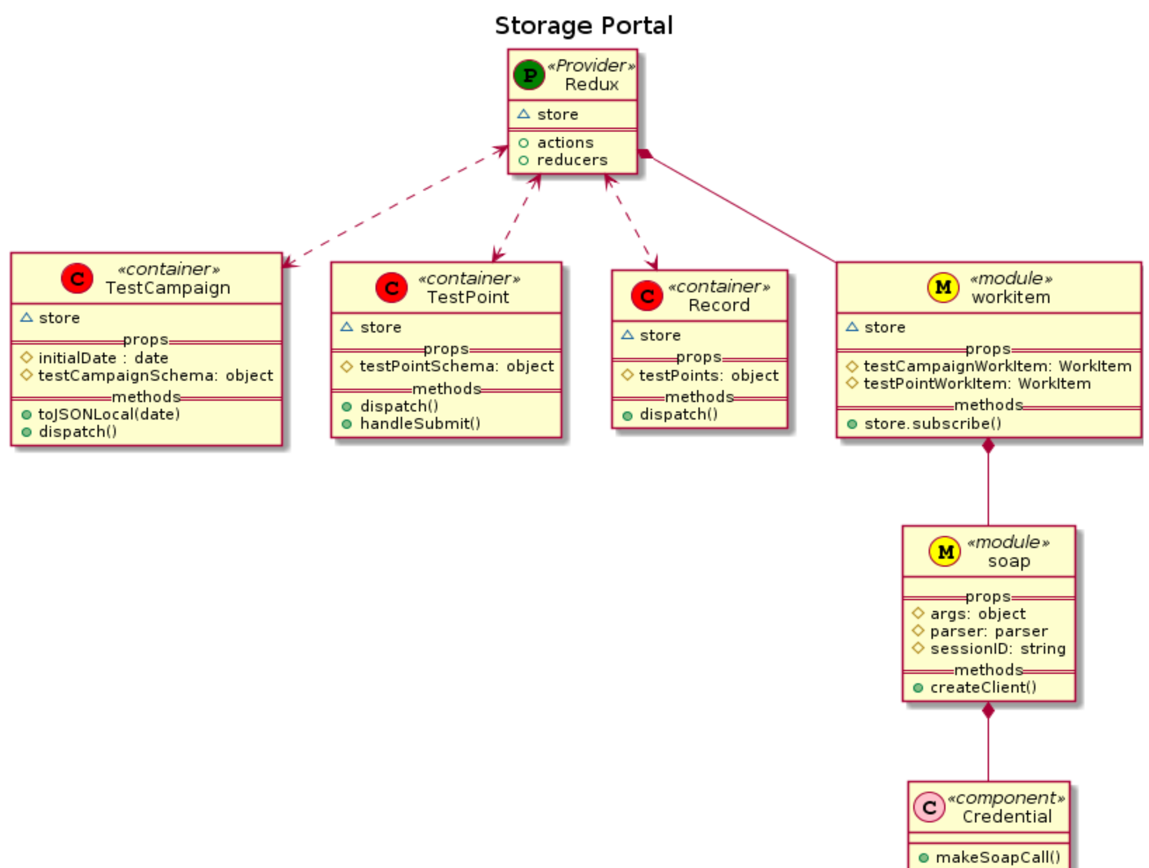

Figure 12. Class Diagram

5 SOLUTION IMPLEMENTATION

This section describes how the features presented in Chapter 3 were implemented. The implementation was based on the model from the previous chapter.

5.1 Development

This subsection describes the development process of the application to meet the requirements specifications laid earlier in the document.

5.1.1 Storage Portal

The application was bootstrapped from Facebook's Node module called create-react-app, which is a boilerplate of a React application. To install this module, Node.js which comes with *npm* was first installed into the development machine. Using *npm install* command, other dependencies necessary for building the application were added on top of create-react-app.

At its first stages, the application was run locally using the *npm start* command. However, after containerizing with *Docker*, a *docker image* was built using the *Dockerfile* and run using the Docker desktop application.

Storage Portal uses Bootstrap to style its user interface. It implements the CSS classes for the buttons, icons, forms and tables put forward by the library. It has been built to be responsive for two views, i.e. XL and LG, because the application is intended to be used by either a laptop or a desktop.

Test Campaign, Test Point and Credential components were built using the *Formik* JavaScript library along with Yup, which makes it efficient to handle forms. The fields T&V Activity request, SIP Activity Request and Path to File were developed using *Field Array* Component from Formik to allow multiple dynamic input.

The application was developed in VS Code and using Prettier as a code linter. Moreover, as it can be seen below, the application was built to be intuitive for the target users.

Figure 13. GUI of Storage Portal

5.1.2 Performance Data Management

As discussed earlier in the document, the application creates work items in Polarion in a specific project created for this purpose. This Polarion project has two work items called Test Campaign and Test Point in accordance with the requirements. The custom fields configuration for Test Point *workitem* is shown in Figure 14.

In Figure 15, a view of a list of dummy test work items inside Performance Measurement Data inside Polarion is shown. The search box is where the Performance Engineer can type in the filters to find a work item with its full information which has been sent from Storage Portal.

5.2 Data Management

Data entered to Storage Portal is validated using the Yup schema for each component containing a form. Yup is a JavaScript schema builder which works very well with Formik. In the Formik element the property *validationSchema* is fed with the Yup schema defined. As an example, we can look at LogInSchema object schema in Figure 16, which validates the input in Credentials section.

test_point-custom-fields.xml

ID	Name	Type
TEST_ID	Test ID	String (sing
TEST_SOFTWARE	Testing Software	String (sing
TEST_FILE_PATH	Path To Test File	String (sing
TEST_POINT_INFC	Test Point Info	String (sing

Figure 14. Test Point Custom Fields

Figure 15. Work Items in Polarion

```
const LoginSchema = Yup.object().shape({
  username: Yup.string()
    .min(6, "Invalid ACCDOM ID")
    .max(6, "Invalid ACCDOM ID")
    .required("ACCDOM ID is required"),
  password: Yup.string()
    .min(3, "Password too short")
    .required("Password is required")
});
```

Figure 16. Login Form Schema

As it can be seen, the schema is made so the minimum and maximum amount of characters are set to be 3 and 6 respectively. The schema also requires the user to input both fields. If these requirements are not met, then the form shows an error message such as the following.

Figure 17. Required Fields Error Message

After the data is validated using the schema the data gets stored in the Redux *store* using dispatch actions. The Record component dynamically fetches and displays these data, which come from the *store* in a table format with the edit and delete buttons for each row. It fetches the data using a React state hook called *useSelector*.

When the user opts to edit a row, the UI changes from Figure 18 to Figure 19 as shown below by fluctuating the *td*'s from *span* elements to *input* elements. Upon clicking the check button on that row, the *EDIT_TESTPOINT* action is dispatched with the *payload* from the table row that made the change. In Figure 20, it can be observed how the *EDIT_TESTPOINT* updates the state in reducers. Other action types used in similar way include *DELETE_TESTPOINTS, SUBMIT_TESTCAMPAIGN* and *ADD_TESTPOINT*.

However, when the user clicks on submit after entering their credentials, *makeSoapCall* is called from within the Credentials component with *username* and *password* as arguments. These arguments are in turn used to create a client and get a *sessionID* from Polarion. This *sessionID* can then be used as a *soap header* to allow further communication with the API.

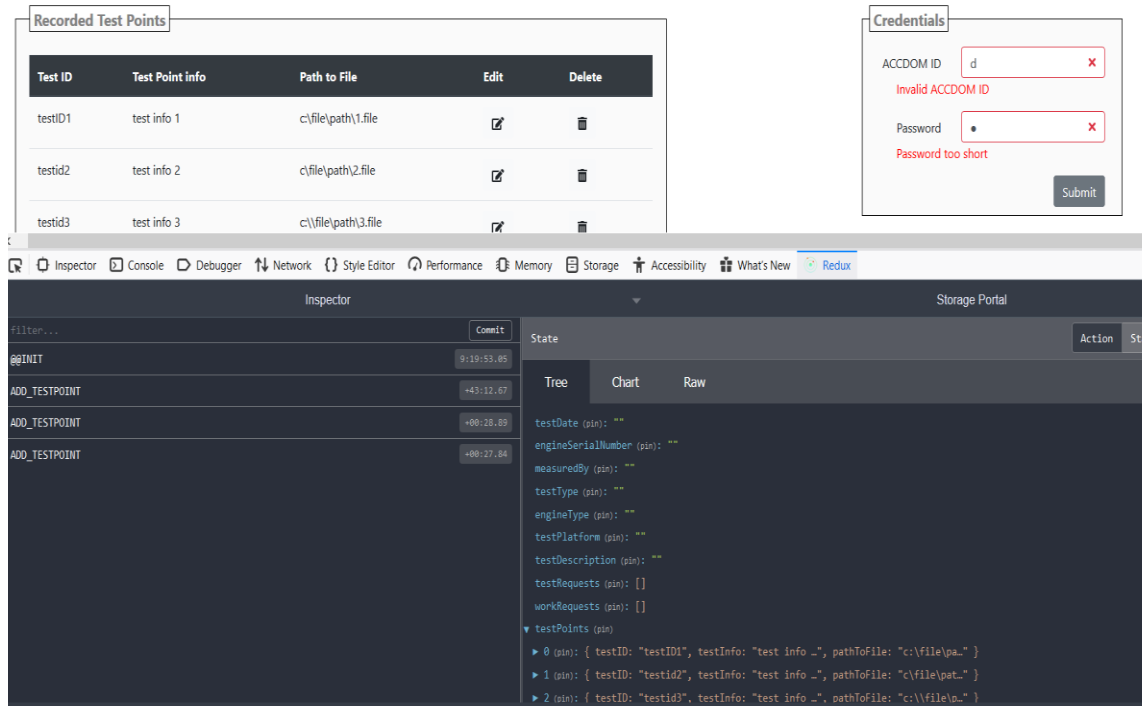

Figure 18. Displaying Test Points from store

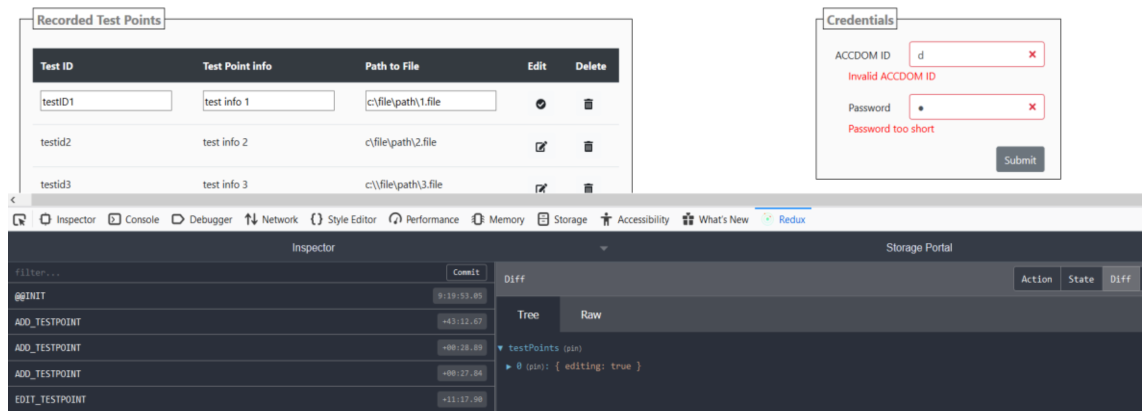

Figure 19. Edit Mode of Record Component

```
case ACTION_TYPES.EDIT_TESTPOINT:
  const editedArray = state.testPoints.map((testPoint, testID) => {
    if (testID !== action.payload.testID) {
      return testPoint;
    }
    return action.payload.testPoint;
  });
  return {
    ...state,
    testPoints: editedArray
  };
```

Figure 20. Editing Test Points in store

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

logIn

```
void logIn(java.lang.String userName, java.lang.String password) throws java.rmi.RemoteException
```

Logs a user in for the current session.

Parameters:

`userName` - the name of the user to log-in.

`password` - the password of the user to log-in.

Throws:

`java.rmi.RemoteException`

Figure 21. Polarion Documentation for *logIn* API function

Using the *sessionID,* we can now create the Test Campaign and Test Point work items inside the Performance Management project. *createWorkItem* is the API function we call to create these work items. The function creates a new Work Item with the given content. This is the content of a *workitem* object in Polarion.

```
WorkItem(Approval[] approvals, User[] assignee, Attachment[] attachments, User author, Category[] categories, Comment[] comments, java.util.Calendar created,
Text description, java.util.Date dueDate, ExternallyLinkedWorkItem[] externallyLinkedWorkItems, Hyperlink[] hyperlinks, java.lang.String id,
java.lang.String initialEstimate, LinkedOslcResource[] linkedOslcResources, Revision[] linkedRevisions, Revision[] linkedRevisionsDerived,
LinkedWorkItem[] linkedWorkItems, LinkedWorkItem[] linkedWorkItemsDerived, java.lang.String location, java.lang.String moduleURI, java.lang.String outlineNumber,
java.util.Calendar plannedEnd, Plan[] plannedIn, java.util.Calendar plannedStart, PlanningConstraint[] planningConstraints, EnumOptionId previousStatus,
PriorityOptionId priority, Project project, java.lang.String remainingEstimate, EnumOptionId resolution, java.util.Calendar resolvedOn, EnumOptionId severity,
EnumOptionId status, TimePoint timePoint, java.lang.String timeSpent, java.lang.String title, EnumOptionId type, java.util.Calendar updated,
WorkRecord[] workRecords, Custom[] customFields, java.lang.String uri, boolean unresolvable)
```

Figure 22. *workitem* Constructor

A *workitem* object can be created with only *Project* and the *Type* set. However, without the custom fields the *workitem* is simply an empty work item created in the specified project. The content of the *Custom* array of all the *workitems* created by this application comes from the *store*. The *workitems* objects are created inside the *workitem.js* module and exported to *soap.js* module.

Creation of Test Campaign and Test Points from the client into Polarion is implemented using jQuery soap library. As an example, we can look at the SOAP call to Polarion's Session Webservice.

```
$.soap({
  url: data.soap.endPointURL.sessionWS,
  method: "logIn",
  appendMethodToURL: false,
  soap12: true,
  data: args,
  success: function(soapResponse) {
    // do stuff with soapResponse
    // if you want to have the response as JSON use soapResponse.toJSON();
    // or soapResponse.toString() to get XML string
    // or soapResponse.toXML() to get XML DOM
    const sessions = soapResponse
      .toXML()
      .getElementsByTagNameNS([illegible], "sessionID");
    sessionID += sessions[0].innerHTML;
    console.log("Congrats! you got a session ID:" + sessionID);
```

Figure 23. SOAP call using jQuery *soap*

5.3 Testing

Tests were performed to make sure that the written code did not break and works as intended. There are three major test named End to End, Integration and Unit Tests. From these majors, however, apart from Manual tests and Static tests – using Prettier, only Unit tests were successfully performed in 8.2 seconds.

These Unit tests were written inside *.test.js* files corresponding to the Component to be tested. The tests were run using *npm test* command. As mentioned in the Theory section of this thesis document, Jest along with Enzyme were used in the testing process. Jest provides an interactive environment with descriptive error messages and error locations printed for the developer in the command line. This can be seen in the example screenshot Figure 24.

The main tests performed were smoke tests. These tests made sure that each component is fully rendered to the DOM successfully. *Shallow and mount* rendering were also performed to the same components for tests involving only that component without the involvement of other components.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\Hmo040\storage-portal> npm test

> storage-portal@0.1.0 test C:\Users\Hmo040\storage-portal
> react-scripts test
 PASS  src/store/reducers/index.test.js
 PASS  src/store/actions/index.test.js
 FAIL  src/containers/TestPoint/TestPoint.test.js
  ● submits correct values

    TypeError: testID.simulate is not a function

      33 |     const pathToFile = container.find('input[name="pathToFile"]');
      34 |     const submit = container.find('button[type="submit"]');
    > 35 |     testID.simulate("change", { target: { value: "mockID" } });
         |            ^
      36 |     testInfo.simulate("change", { target: { value: "mock info" } });
      37 |     pathToFile.simulate("change", { target: { value: "mock path" } });
      38 |     container.find("button").simulate("click");

      at Object.<anonymous> (src/containers/TestPoint/TestPoint.test.js:35:10)

 PASS  src/App.test.js (5.506s)

Test Suites: 1 failed, 3 passed, 4 total
Tests:       1 failed, 7 passed, 8 total
Snapshots:   0 total
Time:        8.19s
Ran all test suites related to changed files.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

Figure 24. Jest test log

As shown on Figure 25, on the Redux side, the *actions* and *reducers* were tested ensuring that the proper action is being dispatched for the specific action type and if the state updates according to the dispatched action type.

5.4 Deployment

While developing and testing the application in the development environment, the version of the code base was being controlled using *git*'s commands, mainly *git add, git commit* and *git push.* When committing a change, the prefixes, such as *feature, bugfix, refactor, style and test* were used. Figure 26 shows the commit history of cors-support branch as an example.

When a branch is ready to be merged to *master* branch, a Pull Request was created. After the code had been checked, commented and approved by the operations team,

the merges took place. After *merge*, the code is *built* as the first stage on the *pipeline* then testing continues. The tests written for each Component will be *run* on the deployment environment. On a successful *build*, the *docker image* will be built following the *Dockerfile* included in the project's repository. This is quickly followed by *docker push,* which pushes it to the registry with its own ID. The code quality and security are inspected by SonarQube. After this stage, the *app* is deployed.

```
src > store > reducers > JS index.test.js > describe("reducers") callback > it("should handle DELETE_TEST_POINT") callback
161          testPoints: [],
162          pathToFile: [],
163          testPointFilled: false,
164        });
165      });
166      it("should handle ADD_PATH_TO_FILE", () => {
167        const payload = {
168          pathToFile: ["path1,path2"],
169        };
170        expect(
171          reducer(undefined, {
172            type: types.ADD_PATH_TO_FILE,
173            payload,
174          })
175        ).toEqual({
176          testDate: "",
177          engineSerialNumber: "",
178          measuredBy: "",
179          testType: "",
180          engineType: "",
181          testPlatform: "",
182          testDescription: "",
183          testRequests: [],
184          workRequests: [],
185          testCampaignFilled: false,
186          testPoints: [],
187          pathToFile: ["path1,path2"],
188          testPointFilled: false,
189        });
190      });
191      it("should handle ADD_WORK_REQUEST", () => {
192        const payload = {
193          workRequest: ["WRQ1,WRQ2"],

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

Test Suites: 8 passed, 8 total
Tests:       26 passed, 26 total
Snapshots:   0 total
Time:        8.364s, estimated 9s
Ran all test suites.
```

Figure 25. Testing reducer for ADD_PATH_TO_FILE action type

CI/CD is implemented using Bamboo server, which performs the tasks specified on the *yaml* file found inside the project. The *Nginx* server serving the *app* is configured to add CORS header to the SOAP calls to Polarion's webservice.

Commits

cors-support ···

Author	Commit	Message
Mohammed, Hindia	514dfd4eafe	refactor: remove redundant location block
Mohammed, Hindia	7596a2d96ca	refactor: override SOAP service endpoint address
Mohammed, Hindia	8272fa07119	bugfix: remove declaration of a variable twice
Mohammed, Hindia	5b866662fda	bugfix: change proxy address to relative path
Mohammed, Hindia	88c137fddbc M	bugfix: adjust adal config
Mohammed, Hindia	8740aa81ebc	bugfix: add cors headers and proxy_pass address
Mohammed, Hindia	64f4d63e617	bugfix: add proxy address
Mohammed, Hindia	a412274447b	bugfix: adjust test date format to match Polarion's format
Mohammed, Hindia	70b380c8bba	bugfix: add proxy and headers to allow cors
Mohammed, Hindia	1852d1f641c	feature: add FieldArray for dynamic array within Formik
Mohammed, Hindia	03a5da4e97b	bufix: add display prop to containerize table
Mohammed, Hindia	3a2b94eb32b	refactor: change variable type for safety
Mohammed, Hindia	928ac86bcb0	feature: add actions for adding paths, work reqs and test reqs
Mohammed, Hindia	748e7e64a18	bugfix: add a checker in state to ensure fields aren't empty
Mohammed, Hindia	e813a0b7d48	bugfix: change date format to match polarion's date format
Mohammed, Hindia	30d785678c5	bugfix: adjust adal config

Figure 26. Commit history

6 CONCLUSION

This thesis project was developed for Wärtsilä to solve a specific problem – acquiring *boundary values* from the performance team – and as part of a large Machine Learning project.

In order to achieve the benefits this project has to offer, specific challenges were overcome, both conceptual and technical. However, the main issue was very poor and low documentation on Polarion and SOAP. In addition to the lack of enough resources on these technologies from the internet and community of developers.

The other main issue is that authentication is carried out in two modes within Wärtsilä. The ACCDOM ID cannot be used to authenticate in Azure AD. Polarion uses ACCDOM for authentication; however, this project is deployed in a location which authenticates users via Azure AD. This meant signing in twice to land on the page or the alternative, which was to let the user land on the page using Azure AD UPN but then signing in using ACCDOM ID when the user is ready create *workitems.* The later alternative was chosen by sacrificing fetching of Test Platforms, Work Requests and Test Requests from Polarion upon landing on Storage Portal.

This project is the first steppingstone and proof-of-concept in the benefit of centralizing engine test data.

7 FUTURE WORK

This project can be continued to centralize the archived test data found in different network drives within Wärtsilä. This can be achieved using Robotic Process Automation along with Storage Portal to save these data in Polarion, i.e. Performance Data Management. The Machine Learning algorithm can then crawl through these data to learn patterns that would immensely support in the study and development of new Wärtsilä engines.

Fetching *workitems* from Polarion upon landing on the page can be implemented by prompting the user to use ACCDOM ID to login before showing the page. After that, the application can update the session by communicating with Polarion before the session expires until the user is ready to submit the data to Polarion.

Currently, the list of Engine Types and Test Types is not found the way the customer wanted it. It has not yet been defined in the combination necessary for the project. Therefore, for the moment, an input field has been opted instead of a dropdown/Autofill list. In the future, this can also be easily solved by dedicating a small server for housing the list as required by the customer, which can be edited by the performance team that will be fetched and displayed in Storage Portal.

REFERENCES

/1/ Wärtsilä Landing Page. Accessed 10.09.2019. Wärtsilä Internal Network

/2/ About Wärtsilä. Accessed 21.10.2019. https://www.wartsila.com/about

/3/ React GitHub repository. Accessed 05.02.2020. https://github.com/facebook/react

/4/ ASAM ODS. Accessed 26.08.2019. https://www.asam.net/standards/detail/ods/

/5/ Google Trends. Accessed 03.02.2020. https://trends.google.com/trends/

/6/ Redux Learning Card. Accessed 21.10.2019. https://www.udemy.com/course/react-the-complete-guide-incl-redux/learn/lecture/8211836#overview

/7/ POLARION ALM unified solution. Accessed 12.09.2019. https://polarion.plm.automation.siemens.com/hubfs/Docs/Fact-sheets/Siemens-PLM-Polarion-ALM-unified-solution-fs-56100-A9.pdf

/8/ Kubernetes. Accessed 21.02.2020. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

/9/ Pant, K. & Juric, M. B. 2008. Business Process Driven SOA using BPMN and BPEL. 1st ed. Lincoln Road Olton Birmingham, B27 6PA, UK. Packt Publishing Ltd.

/10/ Ashley Kitson UML React design. Accessed 29.02.2020. http://zf4.biz/blog/visualizing-react