Tampere University

Lasse Linkola

DESIGN AND IMPLEMENTATION OF MODULAR FRONTEND ARCHITECTURE ON EXISTING APPLICATION

Masters Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Terhi Kilamo
Kari Systä
May 2021

ABSTRACT

Lasse Linkola: Design and Implementation of Modular Frontend Architecture on existing application
Masters Thesis
Tampere University
Information Technology
May 2021

Modular architecture in frontend applications is still a fairly fresh concept, however micro services are widely used in backend software design. In UI development micro services are called micro frontends. Micro frontends are adopted by big companies more and more. In this thesis, we will go through the concepts and the implementation of a web application using micro frontends. The thesis tries to find out whether a modular architecture is a good fit for large-scale web applications.

The client had a problem where they could not easily upgrade their project dependencies. After taking a closer look at the existing architecture, it was determined that the existing architecture had some serious flaws that would cause issues even if the critical dependencies would be updated. It was decided to make a comprehensive refactor for the entire web application. Due to the modular structure of the application, micro frontends were chosen as the way to go.

After designing and implementing these changes, it was determined that micro frontends are a suitable contender for enterprise-level applications. No major obstacles were found that would prevent the usage of micro frontends. We found that micro frontends can bring many benefits such as independent module deployment, team-based module development, and simplified codebases.

Keywords: micro frontend, modular architecture, single-spa, UI, React, single-page application, SPA

TIIVISTELMÄ

Lasse Linkola: Design and Implementation of Modular Frontend Architecture on an existing application
Diplomityö
Tampereen yliopisto
Degree Programme
Toukokuu 2021

Modulaarinen arkkitehtuuri on toistaiseksi ollut vähän käytetty Web applikaatioiden kehityksessä, mutta on ollut käytössä palvelinpuolella jo melko pitkään. Palvelinpuolen kehityksessä modulaarisesta arkkitehtuurista käytetään nimeä mikropalvelut ja selainpuolen kehityksessä mikro frontends. Alan suuret yritykset ovat vähitellen aloittaneet hyödyntämään mikro frontend arkkitehtuuria, ratkaisuna moninaisiin ongelmiin. Tässä diplomityössä käsitellään mikro frontend arkkitehtuurin konsepteja ja toteutusta käytännössä. Työ tutkii modulaarisen arkkitehtuurin sopivuutta laajamittaisiin ja pitkäikäisiin sovelluksiin.

Asiakkaalla oli ongelma käyttöliittymäsovelluksen riippuvuuksien päivittämisessä. Tarkemman tarkastellun jälkeen paljastui, että olemassa olevassa arkkitehtuurissa oli rakenteellisia ongelmia. Todettiin että ongelmat aiheuttaisivat ongelmia jatkossakin, vaikka kriittiset riippuvuudet päivitettäisiin. Tultiin siihen tulokseen, että järkevin tapa ratkaista ongelmat olisi tehdä perustavanlaatuisia muutoksia sovelluksen rakenteeseen. Koska sovellus on olemukseltaan modulaarinen, todettiin että mikro frontendit ovat hyvä tapa toteuttaa sovellus.

Kun sovellus oli refaktoroitu mikro frontend arkkitehtuurin mukaiseksi, todettiin että mikro frontendit soveltuvat hyvin suuriin yritystason sovelluksiin. Muutosten toteutuksessa ei havaittu suurempia ongelmia. Työn tuloksena voitiin todeta että mikro frontendit tuovat monia hyötyjä, kuten itsenäinen moduulien käyttöönotto, tiimipohjainen moduulien kehitys ja yksinkertaisempi koodikanta.

Avainsanat: micro frontend, modular architecture, single-spa, UI, React, single-page application, SPA

PREFACE

Modular architecture in web applications is still a fairly new concept. In this thesis I will introduce the concepts of a modular micro frontend architecture and the implementation. The project was implemented in a bit less than a year. The thesis writing process took approximately 9 months.

I decided to use this project as my masters thesis topic because it was a project that I did as a part of my job. I found the topic interesting as micro frontends were not familiar to me before. The project was done based on the customers need. I had the freedom to design the architecture as I saw what would be the best. I thank my co-workers for the help I received during the implementation.

I would like to thank my employer Enersoft Oy for the opportunity to write my masters thesis for them. I would also like to thank the client, who allowed me to use their project as a subject for my thesis. Finally I thank my thesis supervisor Terhi Kilamo for giving me valuable feedback during the writing process.

Tampere, 7th May 2021

Lasse Linkola

CONTENTS

LIST OF FIGURES

LIST OF PROGRAMS AND ALGORITHMS

1 INTRODUCTION

In the ever-changing environment of web application technologies, keeping up with the latest technology is important. Especially when the application needs long-term support and continuous development for the years to come. For larger applications, the architecture must be designed in such a way that it makes the development of new features easy while updating dependencies regularly.

The context of this thesis is a large-scale custom ERP (Enterprise Resource Planning) system, which handles large amounts of data traffic between different parts of the organization. The motivation of the project is to make the existing front-end system more developer-friendly and save money in maintenance costs in the long run. The purpose of this thesis is to design a new functional and modular architecture and map out the process of implementation of said architecture to the existing application. The goal is to use the latest industry-standard technologies while keeping in mind long-term maintainability.

The project at focus in this thesis has run into a problem of very outdated third-party library versions and a messy codebase. The project is approximately 5-years-old semi-modular UI Web application, and has seen minimal refactoring and updates on dependencies over the years it has been in production and active development. It is finally starting to hit the wall of too extensive technological debt and developing new features is becoming increasingly difficult. The dependencies to third-party libraries and frameworks need to be brought up to date, but the current architecture and implementation is preventing the operation to be done in a sensible and sustainable way. The application has numerous other issues such as a lack of reusable UI components and very inconvenient management of configuration and other global variables

The goal of the thesis is to create an improved modular architecture design for the existing application. The design will be implemented, and finally assessed, whether it achieved the expected results. The research question of the thesis is "Are modular web applications a viable option for a large-scale projects with high rate of developer turnover?". The key points of the assessment are maintainability, developer experience, and continuous integration.

The thesis is divided into two parts. The first part focuses on designing an optimal architecture for a large modular front-end application and discussing the advantages achieved by it. The second part determines the process of how the architecture can be applied to the existing laboratory information system in such a way that it is manageable with

minimal changes to the logic of the current modules.

Chapter 2 has general background information about web-applications and their architecture. Chapter 3 will go through the existing architecture and implementation, discuss its flaws, and assess what needs to be changed. In Chapter 4 discusses the modular design in front-end applications and the key technologies to achieve it. The new architecture is described in detail. Chapter 5 maps out and goes through the steps needed to implement the modular design in the project. The chapter focuses on the technical aspects of implementing the architecture changes on this specific project. Chapter 7 reviews the changes that were made and assesses the end product after the redesign.

2 FRONT-END APPLICATION TECHNOLOGIES

A frontend is the part of an application that implements the user interface. Usually, web applications have both the frontend and the backend, where the backend is responsible for managing the data and frontend for displaying it. There are many ways to produce a web frontend application, but they are all built for the same platform, browser. In this chapter, we will look at different technologies and concepts created to help to build web applications.

2.1 JavaScript

JavaScript is a multi-paradigm programming language that implements ECMAScript specification [3]. It is best known as the language embedded in web browsers, but with the help of NodeJS run-time, it can also be run on servers and embedded systems. Implementations of JavaScript sometimes vary slightly between different web browsers. JavaScript is a scripting language and does not require compiling however in the web environment it is often compiled to such a format that it is minimized and compatible with different browsers.

When using the terms ECMAScript and JavaScript they essentially mean the same thing [4]. In this thesis the term JavaScript will be used. According to a Stack Overflow survey, JavaScript is the most popular programming language in the world with 69,7% of professional developers using it [5].

Since 2015, The Ecma TC39 committee has published a new ECMAScript version each year. The versions are named by adding the year on the name e.g. ECMAScript 2015. Currently newest ECMAScript standard is ECMAScript 2020 or in short ES2020. ES. Next is a dynamic name that refers to whatever is the next version of ECMAScript at the time of writing. Sometimes versions are referenced by their edition number e.g. ES2015 is the same as ES6. ECMAScript development follows a pattern of 5 different stages from stage 0 to stage 4. Each stage represents the state of the process of adding new features. Stage 4 means that the feature is finished but not yet included in the standard revision.[6]

A web browser provides a client-side environment for JavaScript which defines objects such as windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies, and input/output. In the browser environment, JavaScript is loaded from HTML and is reactive to user interaction, thus no main program is needed [3]. Most commonly

JavaScript is used to make the web page content dynamic and add functional logic which wouldn't otherwise be possible.

2.1.1 TypeScript

JavaScript is not a typed language, while it brings some benefits and freedoms compared to statically typed languages, it can lead to error-prone code. To add typing features into JavaScript, TypeScript comes into the picture. TypeScript is a superset of JavaScript, so all the features in JavaScript are available. TypeScript can also be optional in a project, meaning parts of the project can be done without typing and some parts with typing. This means that TypeScript can be gradually adopted into an existing JavaScript project.[7]

Browsers do not understand TypeScript so in the context of web applications the TypeScript code needs to be transpiled into normal JavaScript. This can be done for example with the TypeScript CLI tool or TypeScript loader for Babel. TypeScript features are configured with a `tsconfig.json` file.

2.1.2 JavaScript Package Managers

JavaScript package managers are used to maintain the dependencies of a JavaScript project. The dependency information such as versions is usually stored in a single file called `package.json`. This way all the dependencies can be installed automatically on different environments. Most third-party packages are publicly available to be downloaded from different registries. Generally, few alternative package managers are widely used. All the tools use the same registry or a registry that's contents are almost identical. Organizations may have their private package registries so they don't need to make the code publicly available.

`npm` (short for Node package manager) is a package manager for managing JavaScript project dependencies [8]. NPM can also be used to refer to the npm registry, which is a centralized package feed where most of the open-source JavaScript packages are distributed through. `npm` is the piece of software that is used to maintain project dependencies and npm registry is the feed where packages are located. Another popular alternative is yarn. The core functionality is the same as npm, but it has some different approaches on package installation and version management [9].

2.2 Single Page Applications (SPA)

Single-page applications consist of one bare HTML file which loads the JavaScript and CSS on the page. This way the page is loaded from the server only once. JavaScript is used to dynamically generate and manipulate the DOM (Document Object Model). After the page is initially loaded, the application does not need to talk to the server anymore [10]. However in most web applications there is some kind of server API to fetch data

that is used in the application. SPA achieves faster and smoother transitions between the views as opposed to loading an entire page every time something changes.

There is no shortage of different SPA frameworks. Most modern web applications are SPAs in some form. The most common frameworks are React, Angular, and Vue. In this thesis, we will mostly focus on React as it is the relevant framework in the context of the thesis.

React is a declarative and component-based single-page app user interface framework/library. React application consists of encapsulated components that manage their state [11]. A component is a combination of presentation, logic, and state. Components can pass around data through component props. props change, it triggers a re-render on that component.

Virtual DOM

The virtual DOM is a concept where a virtual presentation of the UI state is kept in memory and synced with the real DOM. This process is called reconciliation which allows React to know when something has changed and a new render is needed. React uses the `render()` function to determine the new state of the UI. React then compares the previous and new state of the UI and calculates what needs to change using the virtual DOM. To calculate the needed updates efficiently React implements a heuristic O(n) algorithm with two assumptions:[12]

1. Two elements of different types will produce different trees.[12]
2. The developer can hint at which child elements may be stable across different renders with a key prop.[12]

So for example for large lists keys should always be given to the list items to ensure fast updates on the list. In most cases, this results in very fast update times.

JSX

JSX is a syntax extension to JavaScript that is used to describe the UI. It looks very similar to HTML but it is important not to mix them up. The reason for using JSX instead of HTML is the fact that rendering logic is inherently coupled with other UI logic, for example, event handling, state changes, and data display preparation. Any valid JavaScript can be embedded into JSX by using curly braces as can be seen in Program 2.1.

```
1 import React from "react";
2 import ReactDOM from "react-dom";
3 import "./index.css";
4 import logo from "./logo.svg";
5 import "./App.css";
6
7 function App() {
```

```
 8    return (
 9      <div className="App">
10        <header className="App-header">
11          <img src={logo} className="App-logo" alt="logo" />
12          <p>
13            Edit <code>src/App.js</code> and save to reload.
14          </p>
15          <a
16            className="App-link"
17            href="https://reactjs.org"
18            target="_blank"
19            rel="noopener noreferrer"
20          >
21            Learn React
22          </a>
23        </header>
24      </div>
25    );
26  }
27
28  ReactDOM.render(
29    <React.StrictMode>
30      <App />
31    </React.StrictMode>,
32    document.getElementById("root")
33  );
```

Program 2.1. *An example of react application. Generated by* `create-react-app` *[1]*

Hooks

Originally React components were mostly written as JavaScript classes that had certain lifecycle functions such as `componentDidMount` and `componentDidUpdate`. Another type of React component is a functional component which is a normal function returning some JSX. Before introduction of hooks in React version 16.8, there was no way to have a state or control the lifecycle of a functional components. [13]

The most important hooks React provides are `useEffect` and `useState`. The useState hook allows a functional component to have a state. The function returns the current value of the state and a function to change the value. The useEffect hook allows the user to tap into the lifecycle of the component. It takes a function and a dependency array as parameters. If the dependency array is empty, the function will be run only on the component mount. Otherwise, the function will be run when any value in the dependency array is changed.

2.3 Compilers and Bundlers

JavaScript does not require compiling, however not all browsers and environments support the newest JavaScript features. To get code that works everywhere, a compilation is needed. The most common tasks that are done by a JavaScript compiler are syntax transformation, implementing features that are missing in the target environment, and source code transformations.[14] A feature that is missing from the target environment and implemented separately is called a polyfill.

Bundlers are tools that take in all of the source code and its dependencies and output it in a single file. The underlying need for bundlers is dependency handling. Traditionally JavaScript files are imported and exported through global variables and `<script>` tags. In this case, the order of the `<script>` tags affects when and where the variables are available. This becomes very messy very quickly and bundlers solve this problem. Other important features bundlers offer are for example code splitting and asset management. Code splitting allows the output to be split in smaller bundles, which can then be loaded on-demand or in parallel reducing loading times. Asset management allows importing files other than just JavaScript such as CSS and images.

2.3.1 Babel

Babel is the most common compiler for front-end JavaScript projects. It leverages plugins and presets to provide a broad amount of functionality and customizability. Babel can be configured using a file called `babel.config.json`. After installing NPM packages for desired plugins and presets, they can be configured for use in the configuration file. For example to compile React JSX syntax, `@babel/preset-react` preset is needed [14].

```
1  {
2      "presets": [
3          [
4              "@babel/env",
5              {
6                  "targets": {
7                      "edge": "17",
8                      "firefox": "60",
9                      "chrome": "67",
10                     "safari": "11.1",
11                     "ie": "11"
12                 },
13                 "useBuiltIns": "entry",
14                 "corejs": "3.6.4"
15             }
16         ],
17         "@babel/preset-react",
```

```
const path = require("path");

module.exports = {
  entry: "./path/to/my/entry/file.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "my-first-webpack.bundle.js"
  },
  mode: "production",
  module: {
    rules: [{ test: /\.txt$/, use: "raw-loader" }]
  },
  plugins: [new HtmlWebpackPlugin({ template: "./src/index.html" })]
};
```

Program 2.3. *Example of a simple webpack config*

```
        "@babel/preset-typescript"
    ],
    "plugins": [
        [
            "module-resolver",
            {
                "root": ["./src"]
            }
        ],
        "transform-class-properties"
    ]
}
```

Program 2.2. *An example of simple babel configuration*

2.3.2 Webpack

Webpack is a static JavaScript module bundler. It does not require a configuration file to run but it can be configured through `webpack.config.js`. The configuration has 6 core concepts: entry, output, loaders, plugins, and mode. Entry is the file where the program starts and tells Webpack to start building its internal dependency graph from there. Output defines where the final output bundle will be created and how it is named. Loaders are used to configure tools to process other file types than JavaScript or to use a custom JavaScript loader. Loader configuration requires a pattern of how to pick files on which it will be applied and the name of the loader to be used. Plugins can provide extra functionality that loaders will not provide, such as bundle optimization, asset management, and injection of environment variables. [15] Setting `mode` parameter to `production`, `development`, `none` enables webpack's built-in optimizations.

2.4 JavaScript Module Systems

In the beginning, JavaScript programs started very small, and they only used to provide some interactivity here and there. They were usually small snippets embedded in the HTML whenever needed. Currently, complete applications such as single-page apps are mostly made with JavaScript, not to mention in other contexts like NodeJS, where there is only JavaScript. Due to this, programs need a way to split up into separate modules which can be imported when needed [16]. There are multiple different types of modules that are currently commonly used. The most notable ones are AMD (Asynchronous Module Definition), UMD (Universal Module Definition), CommonJS, and ES6 Modules.

2.5 Design System

A Design system is the programmatic representation of a website's visual language. It provides guidelines on how to communicate with the user in the user interface. It is a collection of colors, fonts, buttons, image styles, typography, and UI patterns used to convey mood, meaning, and intent [17]. Design system can be a specification of the UI components, or it can include the implementation of said components in one or many technologies. Some popular existing design systems are for example Material-UI by Google and Bootstrap. When creating a web application, one has to decide whether to use an existing design system, create one, or if one is needed in the first place.

A component library is a collection of UI components implemented in a specific framework and language. It is not the same as a design system but usually, a component library is a part of a design system. A design system defines the patterns of component usage in specific contexts. For example, one pattern could be a page layout. Component library implements components `Title`, `Content`, `Footer` and design system defines how to use those components to create a page layout.

There are many ways to create and develop design systems. Many tools have been created to make implementing reusable UI components easy, as well as documenting and demonstrating them. In the context of React, some of these tools are Storybook, Atellier, and React Cosmos. More about how Storybook works in later parts of this thesis.

2.6 CSS in SPA

Traditionally styling web pages has been done by creating style sheets to determine the outlook of the page. The same method can be used in single-page applications however there are a lot of tools to make it more convenient.

There are different ways to approach styling in single-page applications. One of the most common approaches is CSS-in-JS which is a collection of ideas where the styling problems are solved by using APIs instead of convention leveraging JavaScript [18]. Some of the benefits CSS-in-JS provides are modularity, scoping, and state-based styling. There

are many different implementations of CSS-in-JS and the most notable ones are libraries called `jss` and `styled-components`.

2.7 Modular Design and Microservices

Modularity is a useful concept in design cases where complex systems are required. A modular system consists of several modules that are independent of each other but work together. Modular design is based on the way humans solve complex problems, which is by "breaking it down" to smaller separate issues. This way the brain does not have to grasp large complicated concepts and can focus on one thing at a time. [19]

Keywords in modular design are abstraction, information hiding, and interfaces [19]. In computer science the objective is to abstract parts of the software through information hiding [20]. Interfaces are one tool that can be used to hide information between modules. Interfaces define a set of functions that the underlying system implements. Through interfaces, different modules can communicate with each other, without having any of the implementation details exposed.

The benefits of modules can be significant if the modules and their interfaces are designed carefully. For example, modules can be developed simultaneously by different people. The developers only have to communicate about the changes in the interfaces, instead of the implementation specifics. Modular design makes it also easier to design large and complicated systems. It provides a way to isolate smaller concepts into their context and thus making it significantly easier to design good patterns on the top level of the application. [21]

Microservice is a part of a microservice architecture that is implemented using modular design rules. Typically the term is used in the context of server-side applications. Microservices communicate between each other through messages and have their own independent memory persistence. [22]

2.8 Monorepo

Monorepo (Monolithic Repositories or Multi-Package Repository) is a single repository containing more than one project whereas the more traditional model is a single project per repository. The monorepo model is in use in several large software companies like Microsoft, Facebook, and Google. It has also been adopted by some open-source projects. Monorepos can be either Monstrous monorepos or Project monorepos. Monstrous monorepos can be extremely large and usually contain all the organization's code. Project monorepos are smaller and contain a single project's source code. Examples of Monstrous monorepos are large organizations such as Google and Facebook and examples of project monorepos are open-source projects Babel and Ember. [23]

There are many benefits and challenges in using a monorepo. Examples of benefits brought by monorepos are simplified dependencies, cross-project changes, and easy

refactoring. The dependencies are simplified as library versioning is de-emphasized and libraries are expected to have a stable API. When APIs are changed their callers are updated simultaneously. In a well-maintained monorepo, refactoring is easier because everything is in one place. [23]

Some of the more notable challenges that monorepos poses are code health and codebase complexity. Monorepos do not encourage the developers to make stable and well-defined APIs as it is easy to add dependencies between the projects. It is also more difficult to keep track of the dependency graph, which can lead to issues in build binary size and extra work. [23]

Monorepos have been a source of debate for a long time in many different forums. However, no clear consensus has been found on the validity of the monorepo model. Choosing between a monorepo and a traditional project per repository model is difficult as both models have their pros and cons. The decision of using monorepo should be made on a case by case basis and is not necessarily a better approach. [23]

2.9 Continuous Integration, Continuous Delivery and Continuous Deployment (CI/CDE/CD)

Continuous Integration is a practice in software development where the process of integrating team members' work is automated through frequent automatic builds and testing. The goal of CI is to reduce integration problems and detect errors as fast as possible. CI allows companies to have a shorter and more frequent release cycle increasing the productivity of the development team [24]. CI pipeline is a sequence of steps to perform on the source code to make it a package, that is ready to be released into production.

Continuous Delivery (CDE) is a process that ensures that the application is always in a production-ready state by the means of automated testing and quality checks. CDE leverages a set of practices like CI, deployment automation, and more to deliver the software to a production-like environment. [25]

Continuous Deployment (CD) is CDE but taken a bit further. CDE automates the creation of a production-ready application, but the release needs to be done manually. CD also automates the build process but also automates the release to the production environment. In practice, this means that when the developer pushes a code changes to the production branch in the version control, it will be automatically processed by the CD pipeline and released into production. Therefore there are no manual steps in CD. [25]

There are many tools available to implement a robust CI which usually implements the same base features. CI tool is usually a piece of software running on a server, that offers an isolated build and testing environment. Some of these tools are for example Jenkins and TeamCity. The processes can be configured step by step to use whatever tooling is needed by the build. There are separate tools to help to automate release processes. Even when fully automated releasing is not desired, these tools can be very helpful in

managing releases. Some of the CI tools can also be used to manage deployments, but there is also specialized software for it like Octopus Deploy.

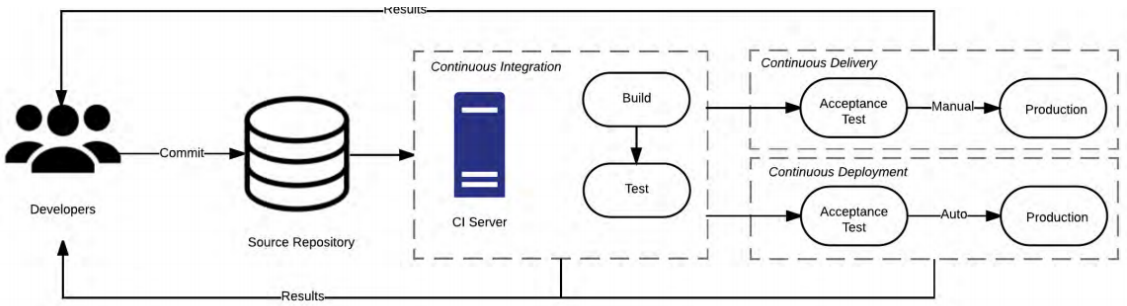

Figure 2.1. *"The relationship between continuous integration, delivery and deployment"[25]*

2.10 Static Code Analysis

There are many ways to find bugs and other issues in the code such as writing tests and doing a thorough code review. Both of these take up a lot of resources. Writing a bunch of tests is of course a good idea but cannot catch some issues such as bad coding conventions. Code review can point out these, but a thorough review can be very expensive because it uses human resources. We also want to find the issues as fast as possible and sometimes code review by humans can take time. [26]

Very often the issues fall into known categories and follow repeated patterns. To pick out most of the common pitfalls, static checkers are employed. A static checker is a tool that is run on the whole code base, searching for certain problematic patterns. The tool checks for errors in the code without executing it. This process is called Static Code Analysis. The checker is usually run after compilation and before testing. [26]

3 EXISTING DESIGN

The architecture is something that could be called semi-modular. It has some ostensible modularity through having the application frame and some components in a separate project, which can be dynamically imported. Inside the projects, the code is structured in a way that promotes their modular usage. However, this is not fully leveraged in the current design because the way it is implemented, does not bring all the benefits of modular design. These benefits could be for example independent development and production delivery.

The project is divided into two applications, MainApp and Portal. MainApp contains all the logic and presentation for all the modules. Portal is running on a separate instance and has the common functionality between the modules, such as the application frame, authentication, password change, etc. Portal can be accessed independently, or the UI components can be imported from the published bundle file over the network. The main application imports the navigation menu and authentication components from Portal which makes it seem like there is only one instance of the navigation menu/frame, but it is separately rendered when accessing MainApp modules.

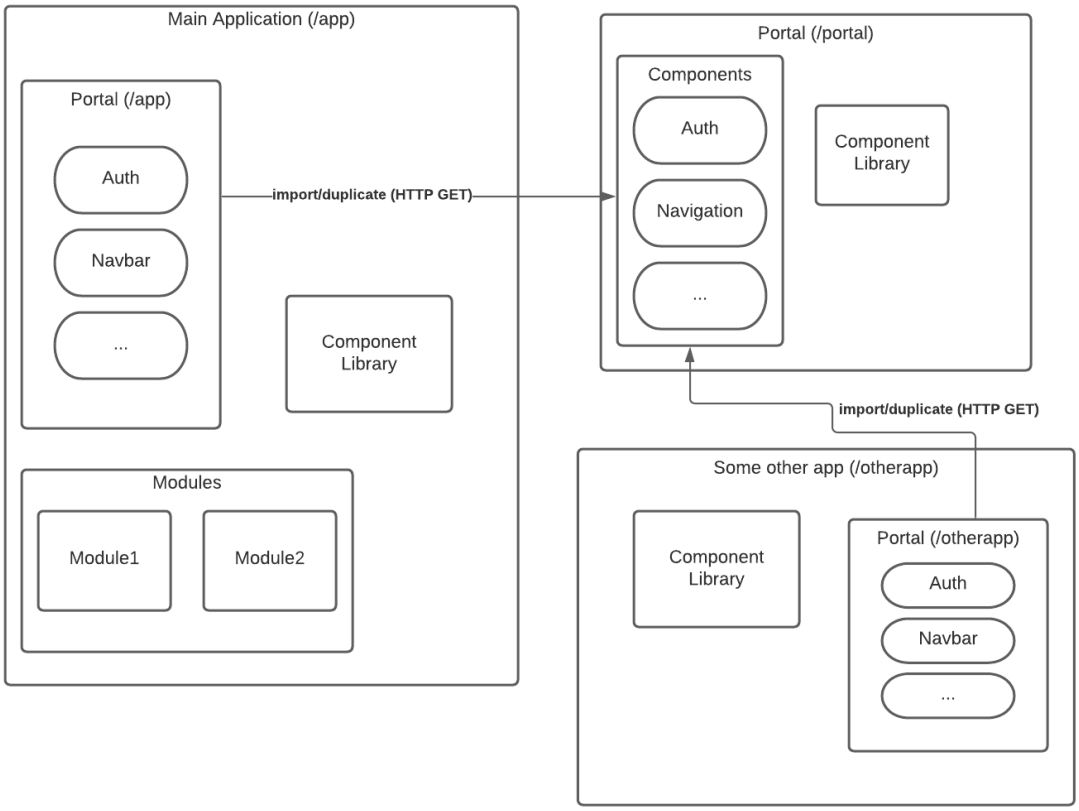

Figure 3.1. *The existing application architecture*

The modules are independent entities of the application. They can also be viewed as sub-applications that have their own set of functionality and purpose. Usually, only one module is active at a time, and navigation to these modules is from the portal.

The modules are React components, implemented in such a way that they are not dependent on any outside data, and are responsible for fetching all the data they need. Module props are used to define some behavior that is dependent on the context they are run. Such functionality could be for example getting some properties stored in the query string. Each module stores and manages its state independently.

The state management is done using a library which was made inside the company some years ago. The library provides an interface to create module-level state management. This means that each module has its state object which can be accessed and modified throughout the module. While the basic functionality is working it is not maintained and can be difficult to use in some cases. It also has some compatibility issues with newer React versions.

DevExpress is a component suite that provides UI components for different frameworks [27]. This project uses the collection of UI components for JavaScript. At the current date, DevExpress has its library for React components, however, when the project was started it did not exist. To be able to use the JavaScript components in React, the project implements a React wrapper that wraps all the JavaScript components and the interfaces

into React components.

The folder structure in both projects is similar. Following the traditional JavaScript project structure, the root path has folders `src` and `public`. The `src` folder includes all the source code and assets and the `public` folder includes the index.html file and some static assets. Inside the `src` the `modules` folder contains all the modules which can be used independently. `app` folder has the top-level application which does the routing and module composition. `services` implements the methods for accessing the REST data API. `lib` and `components` folders contain reusable UI components, theme, and some utility functions.

The configuration for the applications is stored in a JavaScript file that is located in the public directory of the app. It means that it won't be bundled with the rest of the app. Instead, the configuration is injected and accessed through the `window` variable. The configuration variable values are populated with Octopus variable substitution. Both Portal and MainApp have their configuration files, which causes some duplication of the values.

Having the configuration stored in the `window` variable makes it a global variable. Global variables are generally not recommended in programming due to the issues they tend to cause. They can make the code hard to read as the global variable can be changed anywhere without a restriction.

3.1 External modules

Portal is an external module that is published in an independent process and path on the server. When the user enters the application, the first view is the portal which provides navigation buttons to then navigate to the modules. When the user navigates to a module, the user is no longer in the portal applications and is now accessing the MainApp application. However, for the user, it still looks like the same navigation and application frame is present. In reality, the MainApp project just imports the navigation components and renders them themselves. This causes a difficult React version dependency.

To be able to use components exported from the Portal, MainApp needs to have the same version of React. In the scope of these projects, the version should have been updated on both projects at the same time. Normally this would not be a major task, but what made it difficult was that the React version upgrade for just one project was fairly big. The update from version 15.x to 16.x introduced breaking changes that took a while to fix. The company wanted to remove the possibility of severe version difference between modules happening again.

Having Portal as an external module and a runtime dependency did not provide any significant benefits, and instead made the architecture more complicated and caused more problems.

3.2 Component library

The component library is the part that causes the most issues in development. There are two different old component collections, that are not centralized or versioned and have started to live their own lives in multiple different projects. The code is also aged and some components still use some deprecated React API methods. They also have dependencies to some old versions of certain libraries such as bootstrap.

Ideally, a component library would be an independent project with strict versioning. This is not the case in the current system. The current system is difficult to maintain and causes more issues than it solves.

4 MODULAR DESIGN IN FRONT-END DEVELOPMENT

Microservices have gained a significant amount of popularity in recent years in back-end architecture, but in frontend architecture, it is still a relatively new design pattern. Many companies struggle with the limitations of monolithic frontend codebases and micro frontends should be considered as a solution for the problem. [21]

In Web applications that are using the micro frontend architecture, the application is divided into smaller and simpler chunks. These micro frontends can be maintained, developed, and delivered independently. A micro frontend could be defined as "An architectural style where independently deliverable frontend applications are composed into a greater whole". [21]

In a practical case, a micro frontend is a section of the UI, consisting of dozens of components that use a UI framework such as React, Vue, or Angular for rendering the UI. The codebase of these parts is often separated in different repositories but composed in runtime into a single browser tab and single operating system process. One application can have micro-frontends that use different frameworks. For example, the navigation bar could be implemented with Vue and the page content with React.

4.1 Benefits

4.1.1 Incremental upgrades

When the application has a long life and will be developed for many years, possibly tens of years, at some point there will be a time when developing the old codebase is no longer feasible. Maybe the framework has aged and developers want to use the latest technologies. What if hiring developers for the old tech becomes difficult? Normally if using a monolithic application, a complete rewrite would be the most likely option. [21]

Micro frontends make incremental upgrades possible and new technologies can be used in some parts of the application, without having to do it all at once. The old parts can still be maintained as long as necessary. From a business perspective, this is also beneficial, because the developing costs are divided in a much broader time frame, instead of one massive rewriting project. [21]

4.1.2 Simple, decoupled codebases

Each micro frontend is smaller than one monolithic application, and usually smaller applications are easier to develop and design. Micro-frontends discourage unintentional coupling of the components. It is harder to cause unnecessary coupling because you would have to intentionally make it happen between micro frontends. [21]

Micro-frontends should not be an excuse to dismiss other clean code principles. However, it can enforce better practices, because it causes thicker boundaries between contexts of the application. Micro frontends enforce deliberate data and event flow between the parts of the application.

4.1.3 Independent deployment

The micro frontends can be deployed independently. This reduces the risk involved when pushing new features to production. If something goes wrong with the deployment, only that part of the application is affected, and other parts will run just like before. [21]

Micro frontends usually have their independent continuous integration pipeline which builds, tests, and deploys it to the staging environment and production [21]. Since the deployable entities are small, there should be fewer things to think about when making a new release. It is also easier to make smaller releases instead of waiting for other features to finish before some other feature can be released. When the released features are encapsulated in smaller autonomous modules, only the released modules need to be tested. This can reduce the time it takes to execute tests in the release pipeline. However, it is likely a good idea to still run all the integration tests to find unintended coupling with other parts of the application.

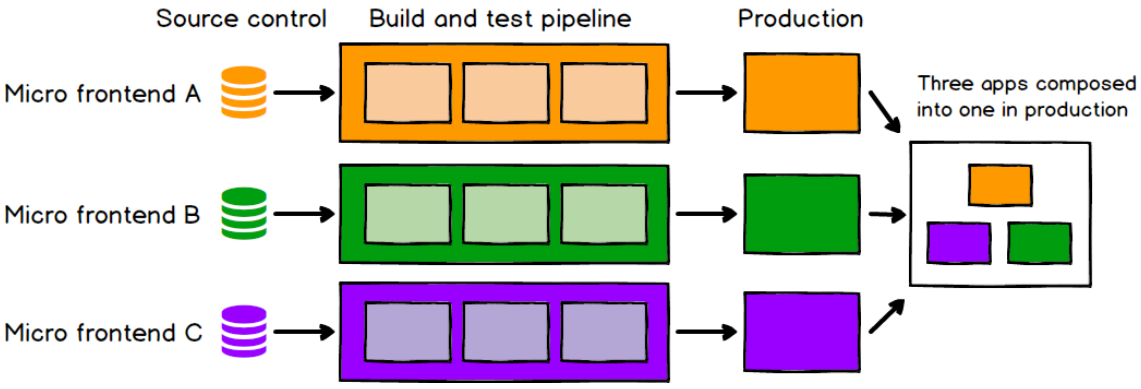

Figure 4.1. *Micro frontend CI pipeline [21]*

4.1.4 Autonomous teams

As micro-frontends allow us to decouple both codebases and CI processes, it leads us towards the ability to have autonomous development teams. One team can have full

ownership of a section of a product from the start to finish. This can make development faster and more effective [21]. To function properly autonomous teams need to have people with a variety of skills that cater to every stage of the development [28]. These teams are called vertical teams, which means the team is product-oriented instead of focusing on a single area of development. An example in frontend development would be to have a graphic designer, programmer, test engineer, and DevOps engineer whereas a horizontal team would have just a group of programmers focusing on the programming of several different micro frontends.

4.2 Single-SPA

The options of tools for implementing a micro frontend application are limited. You can either use `single-spa` or implement the orchestration yourself. In most cases, single-spa is the most sensible option. Some other tools exist, but they are mostly based on `single-spa` anyway. `single-spa` is a framework to help orchestrate and bring together micro frontends. It helps in getting the full benefits of micro frontend architecture without having to write the functionality yourself. These features, among others, are the ability to use multiple frontend frameworks such as React, Vue, and Angular, independent module deployment, and lazy loading of the modules. [2]

The framework is created, developed, and maintained by Joel Denning. The development of single-spa was originally started at Canopy Inc. out of the desire to use React and react-router instead of being stuck at Angular framework [2]. At the moment single-spa is the most mature and established framework to enable micro frontend architecture.

The single-spa application architecture consists of two key parts: single-spa config and micro frontends. Microfronteds can be then divided into 3 different types: single-spa applications, single-spa parcels, and utility modules. Together these concepts are used to build a complete web application. The micro frontends and the root config typically have independent CI processes, versioning, and version control.

Single-SPA config

`single-spa` config often referred to as root config, is the javascript module that orchestrates all other modules. It contains the index.html file of the application and a configuration file. The index.html is the file where the whole application will be rendered to and the configuration file is used to register the micro frontend applications.

Importmap is JSON file that tells where to find each of the micro frontends or shared dependencies. `single-spa` will read this file and fetch the application code from the configured addresses. The paths can be relative or absolute.

```
1 {
2   "imports": {
3     "@org-ui/root": "#{OrgUiRootUrl}",
```

```
    "@org-ui/portal": "#{OrgUiPortalUrl}",
    "@org-ui/app": "#{OrgUiAppUrl}",
    "@org-ui/config": "#{OrgUiConfigUrl}",
    "@org-ui/common": "#{OrgUiCommonUrl}"
  }
}
```

Importmap 4.1. *importmap.json*

`single-spa-layout` is an add-on library to single-spa that helps to set up the top-level routing for the application. The most important tasks that `single-spa-layout` makes easier, are DOM placement of applications, defining application top-level routes, default routes for Not Found pages, and loading UIs/indicators when applications are being loaded [2]. The layout can be configured in 3 different ways: placing a `<template>` element to the index.html `<head>`, providing the library a js configuration object or a JSON configuration file. The template in HTML is the clearest way to define the layout but has some downsides. For example, if the routes are configured externally, injecting the routes to the index.html file is more difficult. Here is where the js configuration object is easiest. The external route configuration can be imported in JavaScript and placed into the configuration object very easily. Unfortunately, the js configuration object does not allow the injection of HTML elements into the layout, unlike the template.

Single-SPA microfrontends

An application is otherwise a normal single page app but it does not have an HTML file. Applications take care of their internal routing and only the root path is configured in the root config. Once the application is mounted on the DOM, it can manipulate its HTML content freely.

Application on a single-spa page has a lifecycle. On each phase of the lifecycle, single-spa runs a lifecycle function. The most important lifecycle functions are bootstrap, mount, unmount, and unload. These functions excluding unload, need to be implemented by the application. Each of these functions must return a promise which means they are asynchronous. For most frameworks, there is a helper library to make registering an application to single-spa easy. The lifecycle functions are implemented by the library and minimal configuration is needed to get a functioning application. For React application the library is called `single-spa-react`.

The following is an example of a single-spa react application index file. The `rootComponent` refers to a React application root component, starting from where everything is like in any other React application. The function `singleSpaReact` returns the lifecycle functions which are then exported so that `single-spa` root config can register the application.

```
import React from 'react';
import ReactDOM from 'react-dom';
```

```
3  import rootComponent from './path-to-root-component.js';
4
5  import singleSpaReact from 'single-spa-react';
6
7  const reactLifecycles = singleSpaReact({
8    React,
9    ReactDOM,
10   rootComponent,
11   errorBoundary(err, info, props) {
12     return (
13       <div>This renders when a catastrophic error occurs</div>
14     );
15   },
16 });
17
18 export const bootstrap = reactLifecycles.bootstrap;
19 export const mount = reactLifecycles.mount;
20 export const unmount = reactLifecycles.unmount;
```

Program 4.2. *Example of react application in single-spa [2]*

single-spa parcel is a framework-agnostic component with a set of functionality, that can be used inside any single-spa application regardless of the framework used to implement the application. They are mounted manually by a function call unlike applications, which are configured in the root config. It is recommended to not use parcels if you use only a single framework in your application. If you only have one framework, you can just import/export components from other applications. If you need a component that works with any framework, you can export a component wrapped in a parcel and then mount it in a DOM node inside the target application.

Micro frontends utility module (MFE) is a type of micro frontend that is not rendered anywhere but can export UI components or other functionality to the applications. Shared logic between applications can be extracted to an MFE to increase code re-usage. MFEs can be also used to share some state between the applications. For example, authentication can be implemented in a way that it has to only be initialized once in the MFE, and other applications can just check the authentication state provided by the MFE.

Topic	Application	Parcel	Utility
Routing	has multiple routes	has no routes	has no routes
API	declarative API	imperative API	exports a public interface
Renders UI	renders UI	renders UI	may or may not render UI
Lifecycles	single-spa managed lifecycles	custom managed lifecycles	external module: no direct single-spa lifecycles
When to use	core building block	only needed with multiple frameworks	useful to share common logic, or create a service

Figure 4.2. *Comparison between different types of single-spa microfrontends[2]*

4.3 Goals for the new design

The goals of the design were discussed with the client. They are mostly based on the problems the existing implementation had, but also include general goals to improve the development process in the big picture. Some of the goals were automatically achieved by using a micro frontends architecture, but some of them required a little bit of thinking. Before starting with the new design, a list of goals was formed and they were following:

1. Independent module development
2. Centralized configuration
3. Update shared dependencies from a single location
4. Modules with independent versioning and CI pipeline
5. Improved development cycle/process
6. No absolute paths in configuration
7. New reusable component library

The product is released in different environments with different configurations so we want the configuration process to be easy and coherent. The goal is to centralize the configuration as much as possible. This was achieved in the old design too, but as the design changes significantly, it is important to keep this in mind.

Dependencies in JavaScript projects can easily go out of date, as there is a large number of external dependencies and libraries that update fairly often. To keep the dependencies up to date, it is important to make the updating process easy. In the new design, we want to have a single location to update shared dependencies for all the modules.

Previously all the modules were behind a single version number even though the modules are largely independent. In the future, we want to track the changes better by having the

modules separated in both versioning and releases. We want to have a separate CI process for each module, so we know which modules are affected when releasing new features. This will also improve the development process by making pull requests, code review, and releasing to production easier.

Configuring URLs with absolute paths has been causing some issues when accessing the application from different environments, for example through proxies. One of the goals for the new architecture is to get rid of these problems and make it possible to use relative URLs as much as possible.

The old reusable components are out of date and need to be replaced with versions implemented using more modern React methods. The components are not versioned and cannot be updated from a single file as the implementation has been duplicated in multiple projects. The goal is to create a centralized and well-maintained component library and use it as much as possible in the micro frontends.

4.4 New Architecture

Due to the nature of the application, it can very naturally be designed in a modular way. The application consists of navigation where you navigate to different sub-applications or views, and they are independent of each other, and no communication between them is needed. For example, one view could be for browsing laboratory test results and another to track the transportation of laboratory samples. If for some reason information was needed to transfer between modules, it would happen through URL query parameters. We wanted to make each independent part of the application have independent versioning and CI process, which fits right into the idea of micro frontends and modular architecture.

It was decided it would be best to separate the application into several repositories, each having its specific purpose. The separation was made mostly based on what would be logically best, and the size of each repository was a secondary thought. Therefore some of the repositories ended up being minimal and only have a few lines of code. The primary goal of the new design was to separate the application frame from the primary content of the app and centralize the configuration into one place. Another goal was to make the modules more independent.

The central piece of the new architecture is the root config. It has a minimal amount of code and contains only the configuration for `single-spa`. It routes and loads the other modules and initializes needed polyfills on the index.html file. This part of the application should not be touched frequently, only when new modules are added.

Originally all of the UI was in one code repository and one React project. For the moment, we wanted to keep them in one place and not touch them too much, so the original UI application was kept inside one `single-spa` module. This is the Module1 shown in the figure 4.3. Module2 represents the future micro frontends which each contains one UI

view instead of many.

One of the goals was to separate the application frame and navigation from the main content of the app. In the figure 4.3 `org-ui-portal` contains the application layout and navigation. The layout consists of a header bar on top of the page and navigation is a menu of links to other modules. By separating the layout and navigation, we can make sure that the UI page remains intact even if an error occurs with another module.

In the figure 4.3 `org-ui-common` is a utility module. As described in 4.2, it does not have a route, but exports functionality for other modules to use. In this case, the utility module contains Auth React component, which can be used as a wrapper to enforce authentication for viewing a part of the application. `org-ui-common` also exports an API service interface, which can be used to access backend API endpoints.

In the figure 4.3 `org-ui-importmap` and `org-ui-shared-dependencies` are not single-spa modules, but contain important files for `single-spa` configuration. Importmaps are separated in a different repository because I wanted them to be versioned separately to reflect the composition of the modules in production. In the production environment, we do not have access to the internet, so the shared dependency libraries and polyfills need to be served inside the internal network. For this purpose, we have `org-ui-shared-dependencies`, where we can also determine the versions of the shared dependencies.

The goal of centralizing the configuration into a single repository was achieved with the module `org-ui-config`. Like `org-ui-common` it is an utility module. The repository includes UI configuration such as API endpoints and module base routes. These can then be accessed from other modules when needed.

The component library is a normal NPM package that implements reusable UI components and it is not a `single-spa` module. In this case, the main difference between a module and a normal library is that a module is imported in run time. Therefore if using a module, there can be only one version in use at a time. A normal library package can be bundled into each sub-application separately in the desired version. The downside is that if all the micro frontends are using the same version of the library, they will be duplicated in each bundle.

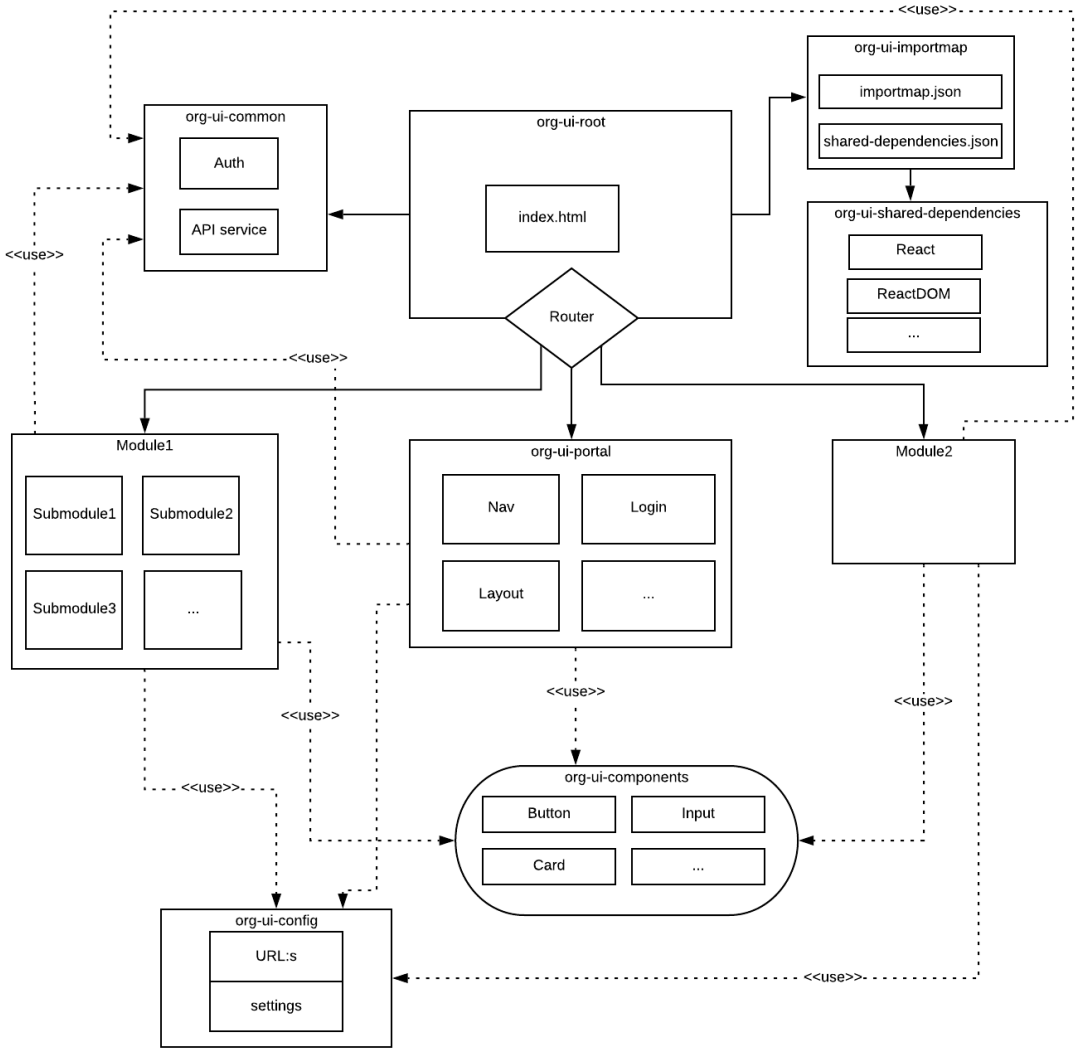

Figure 4.3. *The new application architecture*

4.5 Future improvements

Some changes in the architecture were decided to be postponed due to the need to make large changes in the original code. The original project was divided into modules inside the repository. The modular structure was only on the structure of the code and it did not have any modular features like what `single-spa` provides. It would make sense to split these code modules into micro frontends.

The new architecture at the moment consists of many different repositories. This might make making some otherwise small changes fairly tedious to do. For example, adding a new micro frontend would require changes in `org-ui-root`, `org-ui-importmap`, `org-ui-config` and possibly `org-ui-portal`. This would require creating a pull request for each repository change and would have to be reviewed by someone else. It would be nice to be able to make said changes in a single pull request and repository. This problem could be solved by a mono repository. Repositories `org-ui-common`, `org-ui-config`, `org-ui-root`,

`org-ui-importmap` could be combined under a single mono repository while still preserving the ability to have individual versioning and CI process.

5 IMPLEMENTATION

Even though the old system shares some of the design ideas, the differences between the old and the new implementation are quite significant. Such a large refactor requires time and careful planning. In this chapter, we will define a plan to get the new version out as efficiently as possible. We will also look at each step of the refactoring process and see what needs to be done and how.

The process was divided into two segments: Tasks that needed to be done before the first release of the new system and the improvements that can be done at a later time. It is important to make minimum requirements for the first release version to manage the size of the project.

5.1 Tasks and Road map

To get the project going a road map was created for all the steps that needed to be done to get the product into testing. The road map was good for providing structure to the development as it is sometimes easy to drift into spending too much time on unnecessary tinkering. A structure was also helpful in dividing the project into smaller tasks, which then could be assigned to different people. However, in this case, most of the work was done by a single developer.

1. Upgrading dependencies, most importantly React
2. Fix errors from said upgrades
3. Set up single-spa project
4. Design the microfrontend division
5. Design the configuration management
6. Plan which components and functionality is shared
7. Set up a component library
8. Implement new navigation menu
9. Create microfrontend for shared components
10. Convert the existing modules into `single-spa` microfrontend
11. Configure CI processes
12. Release a version for testing

The easiest point for starting was upgrading the React version as it was one of the things that could be done before the final design for the new architecture was created. Upgrading React did not depend on other changes, but it was a dependency for many other changes. For example, creating a new component library and using the new components required them to be close to the same version of React.

Next up was setting up the `single-spa` project. At this point, We did not have a clear understanding of how exactly `single-spa` works in practice. This was one of the biggest tasks as it required some studying and initial configuration to get the project up and running. After the base setup of the project was done, it was time to start plan out the architecture of the big picture. The plan went through some iterations but the result was the architecture described in the figure 4.3.

After the design was completed it was time to set up a new component library. This is good to do as early as possible so that among other changes required by the refactoring, the old code could be refactored to use the new components. One of the goals in long term is to get rid of the old component library described in section 3.2 and this is a big part of achieving that goal.

After all the foundation for the new application was built, it was time to re-implement the navigation, which was previously done in the `portalweb` project described in the section 3.1. This means a complete rewrite of the navigation menu as a micro frontend. At the same time, micro frontend utility module `org-ui-common` was be created to host shared UI components such as Auth and data API interfaces.

Now it was time to finally convert the existing application modules into a micro frontend. Each of the modules needs to be isolated to work independently. Luckily this was not a big task since a modular code structure was already present on the old implementation. The task includes mostly fitting the routing to work in isolation also some styling on the app container needs to be refactored because the new app container styling is slightly different.

Once the single-spa project is running smoothly with the old modules, it is time to publish a new version for the development server environment. The company already has a functioning CI system using Jenkins for automated builds and Octopus for managing releases. Jenkins and Octopus processes need to be configured for each micro frontend. The goal is to get as much reusable configuration as possible. For this, a Jenkinsfile template that would need minimal changes for each micro frontend will be created.

When the single-spa project has been running for a while and all the detected issues are fixed, it will be released to the test environment. In the test environment, the whole application will be thoroughly tested by some of the testing staff and end-users. At this point, we will repeat a cycle of testing and fixing found issues until it's ready for production.

5.1.1 Future tasks

Some changes were a bit too much effort for the initial release, but some of them are still essential to do at some point. These tasks are not required in the minimum viable product and can be implemented at a later time once the project is successfully adopted in production. Some of them also make more sense to do only when necessary.

1. Do more refactoring as old modules are developed
2. Develop new reusable components as they are needed
3. Refactor old modules to use the new component library
4. Set up an automatic testing system
5. Write tests for the micro frontends

Some of the code in the old project is somewhat outdated and partly done by inexperienced people. However as it still works well enough to be used in production, it is not a critical thing to fix. On these parts of the application, refactoring can be done, when new features related to said part are needed. It will add some extra development time for otherwise small feature updates, but this way the cost of refactoring can be spread out to a longer period.

One major thing the whole application is missing is automated testing. The company has not had any thought or resources put into automatic testing on the frontend development. However, this has changed recently and one of the future goals is to implement wide automated UI testing to help with the reliability of new features and reduce the amount of manual testing. Implementing testing is not on the scope of this project so it will be done at a later point in time. However, it still needs to be taken to account when designing the architecture, since it can affect the testing ability of the code.

5.1.2 New technologies used

When making big changes to a project it is also possible to bring in some new technologies. The following list contains the new libraries and technologies that were adopted during this project.

- styled-components - CSS-in-JSS solution for styling components
- single-spa - Micro frontend framework for single-page applications
- nswag - Utility tool for generating JS classes for backend API access
- Typescript - Typing features for JavaScript
- storybook - Documentation/playground for the component library development

5.2 Upgrading Dependencies

Sometimes upgrading dependencies in old projects can be more work than it might seem, and this was not an exception. It made the most sense to start the upgrading from the most important dependencies and in this case, it was React. To get a newest version of React, the following packages needed to be upgraded: `react` and `react-dom`. These packages need to have the same version to work together properly.

5.2.1 Upgrading React

Upgrading React packages expectedly caused the project not to compile. The errors were examined and fixed one by one. Some of the errors were easy to fix and some of them required more refactoring. The first issue that was a simple fix but a lot of tedious work was to refactor the es5 style import/export statements to es6 style. In the old implementation, the exports were done from a single file, using a small script that scans the individual files/modules. Each of the modules had an ES6 export statement, but the files that re-exported the modules, used es5 `module.exports` statement.

```
1  const modules = (function() {
2    const reqContext =
3          require
4            .context('./modules/', true, /\.module.jsx$/);
5
6    return reqContext.keys().reduce((retVal, filePath) => {
7      const exportName =
8        filePath
9          .slice(0, filePath.lastIndexOf('.module.jsx'))
10         .slice(filePath.lastIndexOf('/') + 1);
11
12         return Object.assign({}, retVal, {
13           [exportName]: reqContext(filePath).default
14         });
15   }, {});
16 }());
17
18 module.exports = modules;
```

Program 5.1. *Old module export file*

This scanning implementation could not be re-implemented with the es6 syntax due to the limitations of es6, and therefore the modules needed to be re-exported explicitly one by one. It was determined that the cleanest way to do that was to use the `export { default as ModuleName } from "path-to-module"` syntax. The statement can be used to re-export default export as a named export.

Next up was refactoring the components that used some deprecated React features. One of these was a modal component that used `ReactDOM.unstable_renderSubtreeIntoContainer()` method to render outside the DOM hierarchy. This particular interface has been completely removed from React so it needed to be replaced with `ReactDOM.createPortal()`, which essentially achieves the same result.

Since React developers published version 17, the developers removed the following old react component life cycle methods:

- `componentWillMount`
- `componentWillReceiveProps`
- `componentWillUpdate`

In the earlier versions the functions caused a warning message, but since React 17, they renamed these functions to

- `UNSAFE_componentWillMount`
- `UNSAFE_componentWillReceiveProps`
- `UNSAFE_componentWillUpdate`

These methods were widely used in the existing code base and needed to be renamed. Luckily there was a tool provided to do it easily. I decided not to start replacing these life cycle methods with valid ones, because it would have been too time-consuming and it was not required for the release. It would have likely caused new bugs and testing would have had to be even more extensive.

5.2.2 Upgrading third party components

There were some third-party libraries used in the project that were stuck on old versions because the new versions did not support the old version of `react`. Once the React version was brought up to date, components such as `react-datepicker` could be updated. The old code had some hacks to circumvent bugs in the third-party libraries, and I was able to remove them once all the libraries were brought up to date.

Some third-party libraries had breaking changes on the API. Luckily most of these changes were not extensive, and they were easily fixed without much trouble. However, in the case of Devexpress, I decided not to upgrade the version as it would require too much refactoring. After consideration, I decided that it would be best to just make sure the old version was working as it was before. These Devexpress components will be removed from the code base as new features are added to the existing application.

5.3 Setting up single-spa project

To create a single-spa project, at minimum the following applications need to be created: root config and a micro frontend. There is a certain amount of configuration needed to

make the JavaScript modules consumable by `single-spa`. The JavaScript code needs to be traspiled and bundled into the correct format. It is highly recommended to use a bundler such as Webpack, Rollup, or ParcelJS. In this project, Webpack was chosen due to it being the industry standard for compiling JavaScript files into bundles. To get Typescript features and wide browser support, Babel was chosen for transpiling.

5.3.1 Webpack and Babel Configuration

Single Spa documentation provides steps to configuring both Webpack and Babel, but to make it even easier, they provide a utility tool called `create-single-spa`. This helper program will create a project with a good default configuration to save the effort of doing it yourself. `create-single-spa` is a command-line tool that takes arguments that define what kind of configuration it will create. Our configuration for the root config was the following.

- framework - React, configures webpack and babel to use React
- moduleType - root-config, creates a template for root config module
- layout, uses `single-spa-layout` to organize routing.

The tool also has a prompt asking whether Typescript will be used or not. If chosen, `create-single-spa` will generate Babel config for Typescript usage. In this case, Typescript was chosen to be used. After the root config was up and running, it was time to transform the old project into a micro frontend. The configuration was generated with the same tool. The only difference was choosing `microfrontend` as a `moduleType`. The tool allows us to run it in an existing project, and it will transform the configuration to compatible with single-spa. However, the new configuration did not work straight away as expected. Some compilation errors needed to be fixed but the fixes were relatively simple and minor.

To allow Webpack configuration customization, we can use a library called `webpack-merge`, which allows us to merge the default configuration provided by `create-single-spa` and our customizations to the config. We needed to add some non-default shared dependencies so we provided our custom externals array. We also wanted to inject configuration variables to our index.html template based on the build environment. This could be done by modifying the HtmlWebpackPlugin options. Finally, we needed to copy the `Web.config` file to our build output. The `Web.config` file is used to configure the server environment on a Windows IIS server. The result for our customized/override configuration was the following:

```
1 mergeWithCustomize({
2     // Prevent duplicate UnusedFilesWebpackPlugin plugin
3     customizeArray: unique(
4         "plugins",
5         ["UnusedFilesWebpackPlugin"],
```

```
6           (plugin) => plugin.constructor && plugin.constructor.name
7       ),
8   })(defaultConfig, {
9       devServer: {
10          port: 3000,
11          historyApiFallback: true,
12      },
13      externals: ["i18next", "react-i18next"],
14      plugins: [
15          new HtmlWebpackPlugin({
16              inject: false,
17              template: "src/index.ejs",
18              templateParameters: {
19                  isLocal:
20                      webpackConfigEnv && webpackConfigEnv.isLocal === "true
21                  orgName,
22                  env:
23                      webpackConfigEnv && webpackConfigEnv.isLocal
24                          ? localConfig
25                          : argv.mode === "production"
26                          ? prodConfig
27                          : devConfig,
28              },
29          }),
30          new CopyWebpackPlugin({
31              patterns: [{ from: "src/Web.config", to: "Web.config" }],
32          }),
33          new UnusedFilesWebpackPlugin({
34              globOptions: {
35                  cwd: path.resolve(process.cwd(), "src"),
36                  ignore: [
37                      "**/*.test.*",
38                      "**/*.spec.*",
39                      "**/*.*.snap",
40                      "**/test-setup.*",
41                      "**/*.stories.*",
42                      "static/*",
43                  ],
44              },
45          }),
46      ],
```

```
47 });
```

Program 5.2. *Root config Webpack configuration*

We use function `mergeWithCustomize` to add our own Webpack configuration, and override the `UnusedFilesWebpackPlugin`. We needed to do this so we could customize the default ignored files list. In addition we added `HtmlWebpackPlugin` so we could use `.ejs` format for the index.html file. This allows us to inject variables to the `index.html` easily. We also define some external modules with the `externals` property. This tells Webpack to use these certain libraries from our shared dependencies instead of bundling them.

5.3.2 Shared Dependencies

Shared dependencies are libraries that are used by all or some of the micro frontends. Typically in a single-spa setup, these dependencies are loaded from CDN through importmap in the root-config. In this case, accessing the public internet was not a possibility since the software is running in a closed network. To go around this problem we created the `org-ui-shared-dependencies` project. Its only function is to serve the static bundle files of the shared dependencies. The following simple script is used to copy the desired packages from the `node_modules` folder to a `public` folder, which then is served from the server:

```
1  const fs = require("fs");
2  const path = require("path");
3  const sharedDependencies = require("./sharedDependencies");
4
5  // Copy shared dependencies to public folder
6  fs.mkdir("public", (err) => {
7    if (err) {
8      console.log("can't create dir");
9    }
10 });
11 sharedDependencies.forEach((value) => {
12   const filename = value.from.split("/").slice(-1)[0];
13   fs.copyFile(
14     value.from,
15     "public/" + (value.to ? value.to : filename),
16     (err) => {
17       if (err) throw err;
18       console.log("copied", filename);
19     }
20   );
21 });
```

Program 5.3. *Shared dependencies copy script*

The script imports a JSON file that contains all the shared dependencies. The file has the filename and the file path of the shared dependency bundle. In most cases, the path points to a folder inside `node_modules`. We can control the version of the shared dependency by editing the `org-ui-shared-dependencies` package.json dependencies.

5.3.3 Shared Logic

Functionality that is very often used by almost every module was separated into a utility module to avoid code repetition. Such functionality includes authentication, web API calls, and configuration. The shared logic is stored in utility modules and can be imported from other modules.

Handling authorization

Auth React component is one of the components that implemented a shared logic. The component's job is to check if the user is authenticated and if not, redirect to the login page. It also manages the user information in `localStorage`, so that it can be accessed and updated across different browser tabs. The component is a wrapper component and inside the wrapped application, user information can be accessed by `useAuth` React hook.

To fill the need for handling data in `localStorage` a React hook `useLocalStorage` was created to abstract the functionality away from the Auth component. Because this is a useful feature in general, it is also exported from the shared logic utility module. The hook takes three parameters: `key: string`, `initialValue: string` and `listenToLocalStorage: boolean`, where `key` is the `localStorage` key, `intialValue` is the initial value of the stored item and `listenToLocalStorage` is a boolean flag that determines whether the outside changes to the `localStorage` value are reflected into the hook value. The hook returns up to date value of the stored item `value`, `setValue` and `clearFromLocalStorage` to make changes to the stored item. Following is an example of the hook usage:

```
const [ value, setValue, clearFromLocalStorage ] = useLocalStorage('KEY', 'Intial
string value', true)
```

API service and nswag

Keeping web applications up to date with the backend APIs is often a headache. You never know what the interface returns and you need to constantly check the API documentation to remind yourself. To solve this problem, a client-side API client was created to access the backend. The client's backend APIs are documented with a tool called swagger, which automatically generates the API documentation from the code. There is a tool called `nswag`, that can generate a JavaScript API client with methods to access the backend. It also generates Typescript typings for the request parameters and return values, which can be used in the front-end JavaScript code. We did not want to create

this separately for every micro frontend as many of them use the same backend API's, therefore it was shared through the shared logic utility module.

To generate the API client nswag needs a configuration file.

Configuration

The goal number one mentioned in section 4.3 was to implement a centralized configuration. In short, when the product is released in different environments, we do not want to have to configure every single micro frontend separately. To avoid this, we created a utility module to store all the configurations needed to make the application work in different environments. These settings include for example API access point URLs and theme settings. The configuration module is also used to store the base routes of each micro frontend.

5.4 Component Library Implementation

There are many tools available to help with creating a component library for React components. For this project storybook was the one that got selected due to the features it provides. It works both as documentation and a development sandbox for the components. It is easy to set up and maintain and it ticked all the required boxes including Typescript support.

One thing that was not done out of the box was building the component library in a form that could be published with npm. As we wanted the library to work with non Typescript projects, we needed to compile the components into regular JavaScript. On the other, we did not want to lose the typing information for the projects that use Typescript. Luckily this could be solved by first extracting the typings from the code using the `tsc` command-line tool, and then building the project using `webpack` and `babel` combination. The type information could be extracted with the following parameters: `tsc -declaration -emitDeclaration -declarationMap`. The type extraction and building were then combined into one NPM script so all the above steps could be run with a single `yarn build` command. On the Webpack configuration we chose UMD as the `libraryTarget` to ensure good compatibility with different projects including `single-spa`.

In Storybook, you can save component use cases as stories. A story is a sandbox that the programmer can use in development. A story is defined in a JavaScript file and in this case each component has at least one story. Stories can also be more complicated and describe a whole view. This is useful when demonstrating and developing the components in cohesion.

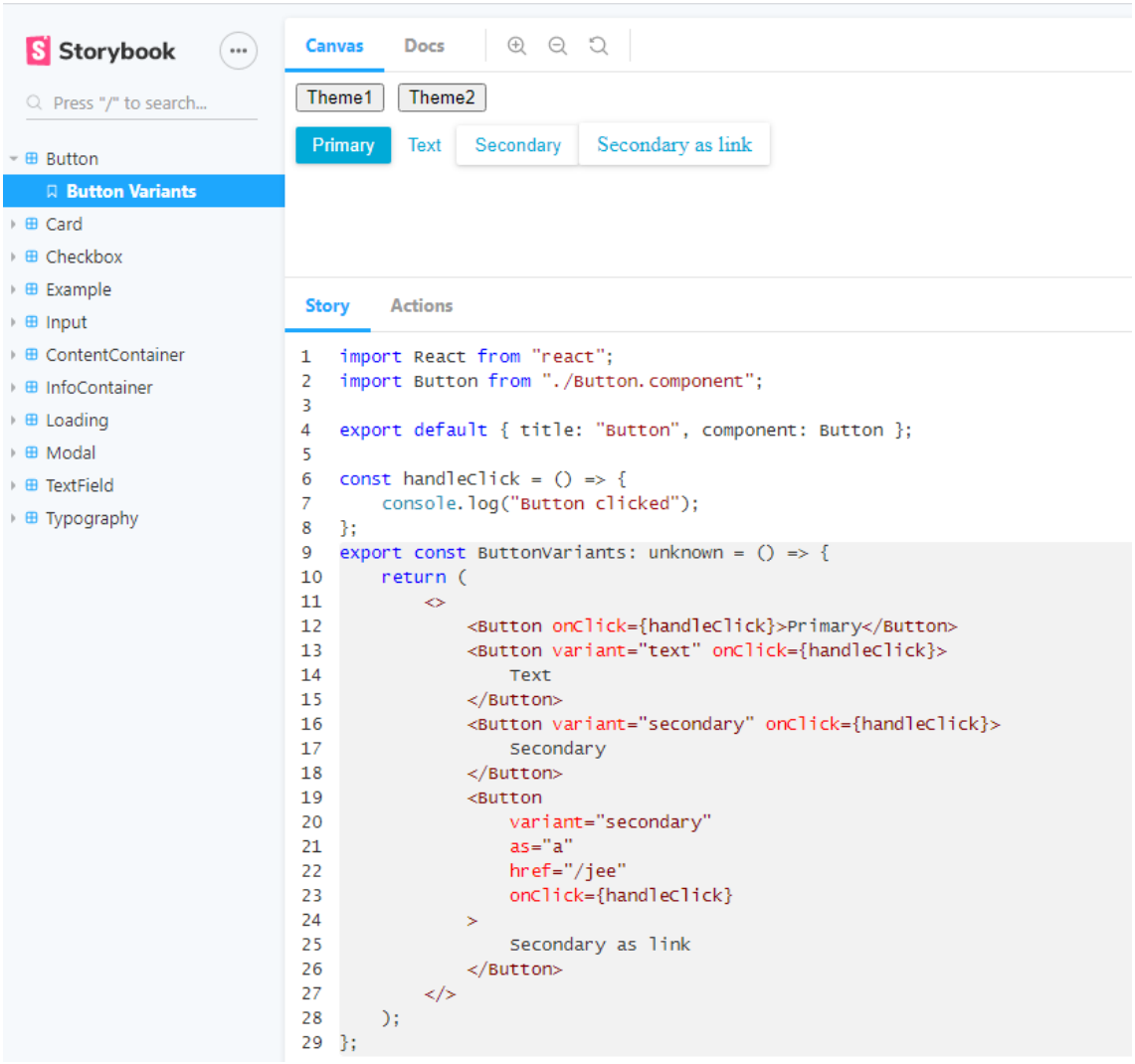

Figure 5.1. *Storybook stories*

The component library needed to support theming. In `styled-components` a theme is applied by wrapping the application with `ThemeProvider`. The theme consists mostly of different colors. To make the theme development easier, we created a storybook story wrapper for the theme. Now every story is automatically wrapped with the `ThemeProvider` and there's no need to do it in the story. The wrapper also includes buttons to switch between themes in the story sandbox. This makes the component development and testing for both themes easier.

```
1 import React, { useState } from "react";
2 import { ThemeProvider } from "styled-components";
3 import theme1 from "../src/themes/theme1";
4 import theme2 from "../src/themes/theme2";
5
6 import { addDecorator, configure } from "@storybook/react";
7
8 // automatically import all files ending in *.stories.js
```

```
 9 configure(require.context("../src", true, /\.stories\.js$/), module);
10
11 const GlobalWrapper = (storyFn) => {
12     const [theme, setTheme] = useState(theme1);
13     const handleUseTheme = (t) => {
14         setTheme(t);
15     };
16     return (
17         <ThemeProvider theme={theme}>
18             <div style={{ marginBottom: 10 }}>
19                 <button
20                     style={{ marginRight: 10 }}
21                     onClick={() => handleUseTheme(theme1)}
22                 >
23                     Theme1
24                 </button>
25                 <button onClick={() => handleUseTheme(theme2)}>
26                     Theme2
27                 </button>
28             </div>
29             {storyFn()}
30         </ThemeProvider>
31     );
32 };
33
34 addDecorator(GlobalWrapper);
```

Program 5.4. *Storybook theme warpper*

5.5 Continuous Integration

The client had a CI pipeline setup that the project needed to be adapted to. Luckily the existing pipeline was flexible enough to accommodate a micro frontend application. The CI pipeline had already automated building, static code analysis, and automated release management tools configured. We only needed to configure the process for each micro frontend.

5.5.1 Jenkins

The client had a Jenkins environment set up for automated building, however, the version that was currently in use was very outdated and hard to upgrade. During the thesis project, there was another project ongoing with the intent to create a new Jenkins environment from scratch. The timing was perfect as this project could be configured in the

new environment.

Previously the Jenkins jobs were configured as normal Jenkins jobs, but the new system leveraged the Jenkinsfile configuration. This means that every job is configured in a file called `Jenkinsfile` which resides in the root of the code repository. The previous system did not allow the configuration to be managed in version control, which made tracking changes more difficult. The Jenkinsfile uses `Groovy` syntax. The CI process in Jenkinsfile consists of stages. In our Jenkinsfile there are stages build, scan, quality gate, pack, publish and release. Each of the stages has steps which are a sequence of shell or Jenkins commands. For example, in the build stage commands `yarn install` and `yarn build` are run. The stages can be run conditionally. For example, in this case, we only wanted to run the release stage on the `develop` branch.

The goal was to create a Jenkinsfile that could be used with any micro frontend without changes. The individual configuration for the job was done with environment variables. In this particular case, customized environment variables were not needed. All the variables are extracted from the build process environment. For example, the project name is parsed from the branch name and the version number of the build is extracted from the `package.json` file.

Jenkins can run multiple build processes concurrently, however, in our case it causes some issues. Yarn is a tool that cannot have multiple instances running at the same time on the same server. We ran into this issue when we noticed that sometimes builds fail due to a cache issue. One solution for the problem would be to containerize the build process, however, at this time the environment was not ready for it. Instead, it was decided that the best solution for the moment was to disable concurrent builds in the Jenkinsfile.

5.5.2 SonarQube

SonarQube is a tool for static code analysis. It allows automated code checking for potentially problematic patterns in the entire code base. SonarQube analysis is started from the Jenkins pipeline. It will produce a report for every build done by the pipeline it will either pass the quality gate or not. If the code does not meet the criteria based on the SonarQube configuration, the build will fail.

Setting up this project for Sonarqube analysis was straightforward. For the most part, no extra configuration was needed. Some projects had programmatically generated code, which we did not want to be analyzed. Certain files could be excluded from the analysis by creating a file called `sonar-project.properties` in the root of the project and configure a key `sonar.exclusions` to point in the file or files.

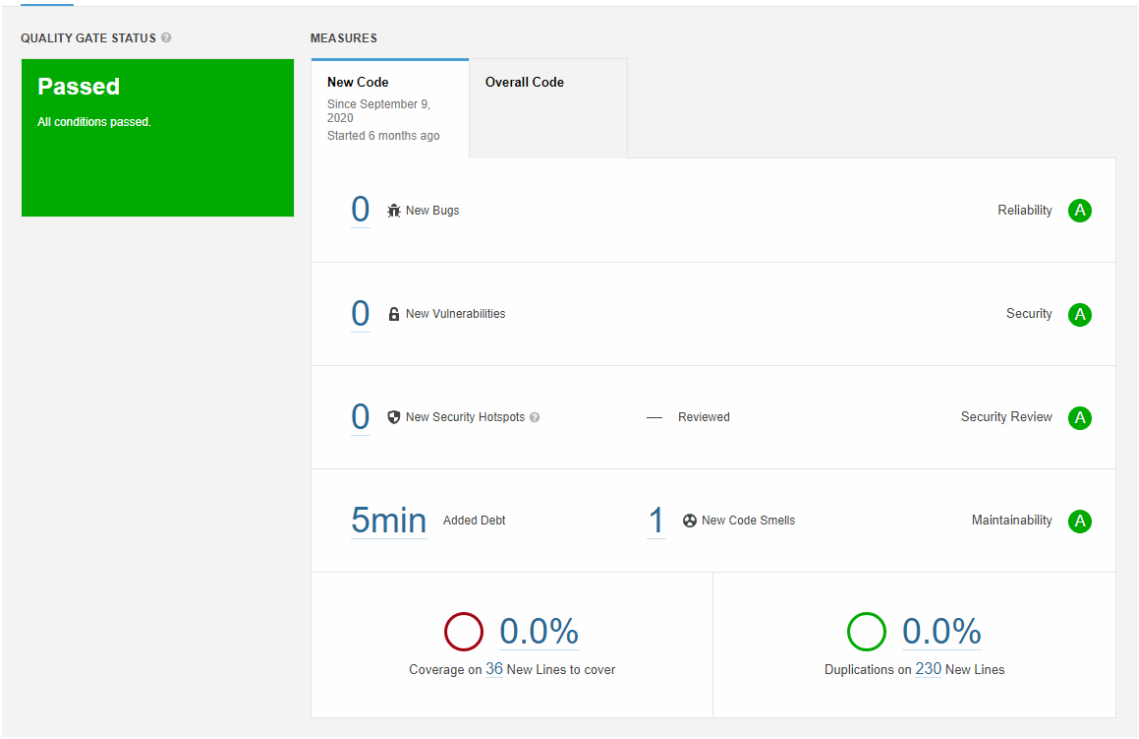

Figure 5.2. *An example of a SonarQube analysis report.*

5.5.3 Octopus

Octopus is a tool for managing and automating releases and deployments in different environments. An Octopus project consists of the following concepts: deployment process, channels, lifecycles, and releases. The deployment process is a sequence of steps needed to create a release from the application package that was previously created by a Jenkins job. A lifecycle is a set of stages/environments that the application goes through in its lifetime. For example, the application can be first released in a staging environment and later to the production environment. A release is the product of the deployment process. After a release is created, it can be deployed to as many environments as needed. Releases can be assigned to specific channels based on criteria configured in the deployment process. For example, different channels can have their lifecycles. Octopus allows the user to do variable substitution on the package. This is very useful in configuring variables that are dependant on the environment of the release. [29]

The client already had an Octopus environment set up and standardized application lifecycles so none of that needed to be designed in this project. An Octopus project was created for each micro frontend. The variable substitution was configured for the configuration micro frontend and the root project. This was done by using syntax `#{Variable}` in the code, which then would be found by Octopus and replaced with the corresponding variable configured in the octopus project. The variable substitution was primarily used for configuring API endpoints and other URLs.

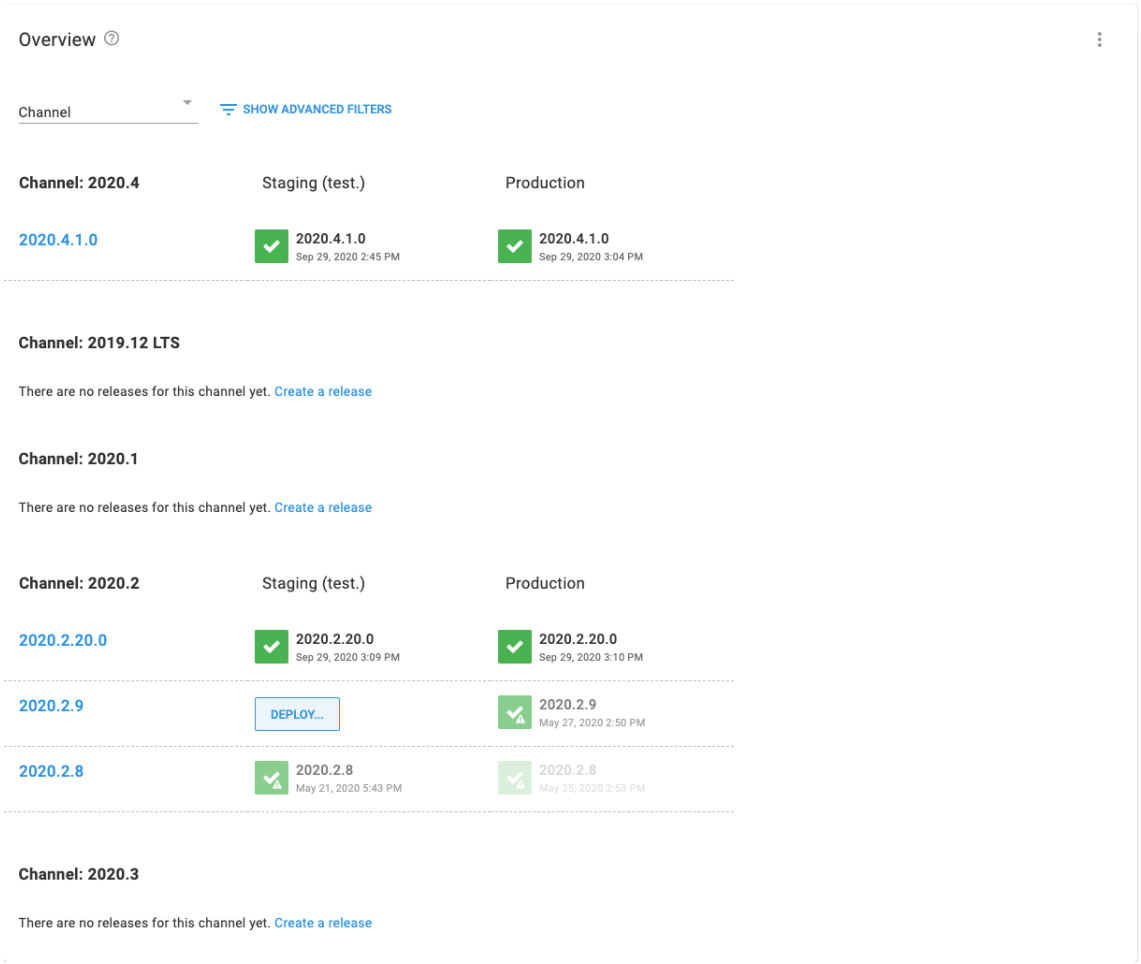

Figure 5.3. *An example of octopus project overview. [29]*

5.6 Package Distribution

Npm and Yarn allow installing packages from git repositories however it has some caveats. We wanted to make package installation as simple as possible, so we published each of the packages into a private ProGet registry. The publishing step can be easily included in the Jenkins pipeline. All that was needed was to configure a NodeJS configuration in Jenkins to use the private ProGet registry and then use said configuration in the `Jenkinsfile`.

```
1 stage('publish') {
2     when { branch 'master' }
3     steps {
4         nodejs(configId: 'private-proget',
5             nodeJSInstallationName: 'Node'){
6                 sh "npm publish"
7         }
8     }
9 }
```

6 EVALUATION

The design and implementation of modular architecture in web applications were completed successfully. During the project, plans and requirements were adjusted on multiple occasions accordingly when issues emerged. The project did not face any major obstacles even though it lasted longer than expected. This chapter reviews the original goals and whether they were achieved in the result.

6.1 Modular architecture in web applications

Now that the project is completed, we can look into the research question of the thesis. The question was whether the modular web application architecture would be a good fit for a large-scale web application with a high rate of developer turnover. By looking at the result, we can see that we managed to isolate parts of the application into mostly independent entities. This makes it easier for new developers to get into development as they do not need to get familiar with the entirety of the project. The isolated entities fit well into the ecosystem of enterprise-level software development. The micro frontend architecture can leverage all the other modern tools for code management, analysis, and CI/CD/CDE.

We found no great obstacles that would prevent large-scale applications from adopting the micro frontend architecture. Based on the findings we can determine that it brings more benefits than downsides. Most of the downsides are based on the overhead that the architecture brings on the development so it is recommended to review each case independently to see if the benefits of micro frontends are relevant to the application.

6.2 Goals

In section 4.3 a set of goals was listed for the transformation of the existing web application. We will go through each of these points and see if and how they were solved in the new design and implementation.

Goal #1 - Independent module development

The first goal was that independent teams could work on parts of the application, without needing close coordination with other teams. In the new design, each of the modules is

stored in their repositories, which means that this goal was achieved. Only when making modifications to shared micro frontends is coordination needed. In a real scenario, most of the time modifying shared modules means adding some values to the configuration module. This is a simple change and does not need much of a review so most of the time teams can work on their features independently.

Goal #2 - Independent module versioning

The second goal was independent module versioning. This was achieved by separating modules into different repositories and having independent build processes. It will make release management easier.

Goal #3 - Independent module CI pipelines

The goal of Independent CI pipelines for each module was also achieved. Once the repositories had their build processes and no build dependencies to each other, the solution was trivial. The same Jenkins configuration is mostly reusable without changes for each micro frontend.

Goal #4 - Improved development cycle/process

The fourth goal of the improved development process is a combination of goals 1-3. In conclusion, since all the above goals were met sufficiently, Also the fourth goal was achieved. The development cycle from code changes to deploying the changes to the production environment was improved and the release changes can be more isolated.

Goal #5 - No absolute paths in configuration

Previously absolute paths in configuration caused problems in some cases where the web application was accessed through proxies. The new design allows using only relative paths, which eliminates the issue. In addition, the configuration is more flexible and thorough without adding much more complexity.

Due to the nature of micro frontends and requirements by the singe-spa library, the number of needed configuration variables increased, but we did not find that it was a problem. However, it might require a bit more time for a new developer to get familiar with the architecture and therefore understanding all the configuration. The result is deemed satisfactory, while improvements can and will still be made.

Goal #6 - New reusable component library

A new component library was founded. At this time the library is not yet fully mature and will take time to form into its final form. However, it can already be used for most of the simple UI components such as buttons, inputs, and page layout organizing. The library is at a point where the old corresponding components can be deprecated and the codebase can be refactored to use the new library instead. It became clear at the very start of the project that this will however take time and cannot be done at once. In the scope of this project, the result was as expected.

6.3 Concerns

The project was successful and no major concerns regarding the architecture surfaced. However, there are always things that are possibly inconvenient and cause some headaches.

One of these things is the fact that while the architecture allows using different UI frameworks, using the same framework with two different versions can be problematic. Of course, the goal is not to use different versions of the same framework, but sometimes it might be a nice temporary solution. For example, if a library update comes with a breaking change, all the modules would not be needed to be upgraded at once. It is dependant on the framework whether parallel instances work. For example, React will work as long as the micro frontends running simultaneously do not use cross micro frontend components.

Another inconvenient feature of modular architecture is a dependency management and specifically managing dependencies between different modules. A lot of focus needs to be put on dependency management during development to prevent the dependency graph from exploding into a complicated mess. This issue can be managed by having strict code review practices to make sure no unnecessary dependencies are created.

7 CONCLUSION

The thesis was made in collaboration with Enersoft Oy. A client of Enersoft needed to upgrade their outdated UI project and a new underlying architecture that would be more maintainable and robust. The research question for the thesis was whether a modular architecture in web applications is a good fit for long lifespan projects, where the turnover of developers is high.

In the first part of the thesis, we made an introduction to single-page web applications. We first went through the traditional single-page applications and then moved on to take a closer look at how to utilize a modular architecture in web development. We introduced a concept of micro frontends studied the benefits they may bring to large-scale web applications.

In the second part, we looked into an existing, rather traditional single-page application and made a plan on how to migrate this particular application into a modular architecture. We laid out goals we wanted to achieve in the refactoring process. We carefully planned the steps to convert the existing architecture and implementation into a modular architecture using micro frontends. The `single-spa` framework was chosen as the primary basis for the application as it implements most of the features we needed to reach our goals.

The changes were implemented using the latest robust technologies while keeping long-term maintainability in mind. After completion we determined that the micro frontend architecture is a valid contender when choosing the architecture type of a web application project. We also looked back on our goals and reviewed if the result accomplished those goals. No great great obstacles were faced that would have prevented the successful completion of the project. The goals set for the project were met in a satisfactory manner.

REFERENCES

[1] *React - Create React App*. URL: https://reactjs.org/docs/create-a-new-react-app.html (visited on 09/11/2020).

[2] *single-spa*. URL: https://single-spa.js.org/docs/getting-started-overview (visited on 02/10/2021).

[3] *ECMAScript 2021 Language Specification*. Sept. 10, 2020. URL: https://tc39.es/ecma262/#sec-intro (visited on 09/11/2020).

[4] Wirfs-Brock, A. and Eich, B. JavaScript: The First 20 Years. *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386327. URL: https://doi.org/10.1145/3386327.

[5] *Stack Overflow, Most Popular Technologies Survey*. URL: https://insights.stackoverflow.com/survey/2020/#technology-programming-scripting-and-markup-languages (visited on 09/11/2020).

[6] *The TC39 Process*. URL: https://tc39.es/process-document/ (visited on 09/18/2020).

[7] *TypeScript*. URL: https://www.typescriptlang.org/ (visited on 03/25/2021).

[8] *npm - Node package manager*. URL: https://www.npmjs.com/ (visited on 03/25/2021).

[9] *Yarn package manager*. URL: https://yarnpkg.com/ (visited on 03/25/2021).

[10] Flanagan, D. *JavaScript: the definitive guide*. 2006.

[11] *React - A Javascript library for building user interfaces*. URL: https://reactjs.org/ (visited on 09/11/2020).

[12] *React - What is virtual DOM?* URL: https://reactjs.org/docs/faq-internals.html#what-is-the-virtual-dom (visited on 09/11/2020).

[13] *React Hooks*. URL: https://reactjs.org/docs/hooks-intro.html (visited on 03/04/2021).

[14] *Babel*. URL: https://babeljs.io/docs/en/index.html (visited on 09/18/2020).

[15] *Webpack - Concepts*. URL: https://webpack.js.org/concepts/ (visited on 09/18/2020).

[16] *Developer Mozilla - JavaScript Modules*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (visited on 09/25/2020).

[17] Godbolt, M. *Frontend architecture for design systems: a modern blueprint for scalable and sustainable websites*. " O'Reilly Media, Inc.", 2016.

[18] *What actually is CSS-in-JS*. URL: https://medium.com/dailyjs/what-is-actually-css-in-js-f2f529a2757 (visited on 04/01/2021).

[19] Baldwin, C. Y., Clark, K. B., Clark, K. B. et al. *Design rules: The power of modularity*. Vol. 1. MIT press, 2000.

[20] Colburn, T. and Shute, G. Abstraction in computer science. *Minds and Machines* 17.2 (2007), 169–184.

[21] *Micro Frontends*. URL: https://martinfowler.com/articles/micro-frontends.html (visited on 11/27/2020).

[22] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.

[23] Brito, G., Terra, R. and Valente, M. T. Monorepos: a multivocal literature review. *arXiv preprint arXiv:1810.09477* (2018).

[24] Fowler, M. and Foemmel, M. Continuous integration. *Thought-Works) http://www.thoughtworks. com/Continuous Integration. pdf* 122.14 (2006), 1–7.

[25] Shahin, M., Babar, M. A. and Zhu, L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.

[26] Louridas, P. Static code analysis. *Ieee Software* 23.4 (2006), 58–61.

[27] *DevExpress*. URL: https://www.devexpress.com/ (visited on 03/31/2021).

[28] Stray, V., Moe, N. B. and Hoda, R. Autonomous Agile Teams: Challenges and Future Directions for Research. *Proceedings of the 19th International Conference on Agile Software Development: Companion*. XP '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450364225. DOI: 10.1145/3234152.3234182. URL: https://doi.org/10.1145/3234152.3234182.

[29] *Octopus Deploy*. URL: https://octopus.com (visited on 03/19/2021).