Trinh Ky Nam

UI LIBRARY PROJECT SETUP FOR VAIMO GROUP WITH MODERN WEB TECHNOLOGY

UI LIBRARY PROJECT SETUP FOR VAIMO GROUP WITH MODERN WEB TECHNOLOGY

Trinh Ky Nam
Bachelor’s Thesis
Spring 2019
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author: Trinh Ky Nam
Title of the bachelor’s thesis: UI Library Project Setup for Vaimo Group with Modern Web Technology.
Supervisor: Lasse Haverinen
Term and year of completion: Number of pages: 48

The library, which is the Vaimo UI library, set PWA (Progressive Web App) technology as a base, along with some modern Web tools/frameworks/libraries such as ReactJs, GraphQL, etc… and built on top of a usual Magento 2 (version higher than 2.3) framework. The library consisted of some general and reusable components for other Progressive Web App to start with in the future.

Throughout the thesis document, project Meka, one of Vaimo’s current ecommerce store project would be served as the showcase of the Vaimo UI library usage. The Meka project aimed to simplify the user experience and increase conversion rates with a new design and performance functionalities that are taken from the benefit of using the general Vaimo UI library. Expected end-results and methods of execution for building the UI library would be collected and described as detailed as possible. To be able to setup the Vaimo UI library, a solid understanding of Magento 2 architecture and ReactJs best practices were required, followed with hundreds hours of researching, documenting, and discussion with participating developers on how the library structured and functioned to become an elegant, small, simple and reusable library that focus on the Progressive Web App technology and its benefits.

At the end of the project, a base setup of the Vaimo PWA UI library was ready, components could be composed in the Meka project, and best practices in Magento 2 implementation towards PWA were concluded and documented on Vaimo’s Confluence Page.

Keywords: Progressive Web App, Magento 2, ReactJs, reusable UI library, Magento 2 PWA best practices

PREFACE

This thesis topic, Vaimo UI library with Progressive Web App technology, was commissioned by Vaimo Product Team, and it has currently been developed and maintained mostly from some developers in Vaimo Finland office. The start of the project began on March, and the implementation phase has been activated since the beginning of April.

Many thanks should be given to the hard work of engineers in Vaimo Finland offices, for spending hundred hours of researching, planning, communicating, developing together to get the base setup of the library done. A big thank will also be given to Lasse Haverinen, the instructor of the thesis who has pointed out and corrected many problems of the thesis. This thesis could not be done without all the help and cheer from all people involved in.

Oulu, 07.05.2019
Trinh Ky Nam

CONTENTS

1 INTRODUCTION

This thesis is related to Vaimo’s request for a UI library that could get reused throughout the company. Vaimo, a digital commerce business related company, has a main goal to accelerate sales for their B2B (business to business) and B2C (business to consumer) clients by delivering digital storefronts, solutions and mobile applications. From the early day of Vaimo group, Vaimo’s technical experts have chosen Magento 2 as a platform to develop an ecommerce store to start the business, since at that time, Magento 2, a PHP framework, is the dominant in this specific field and fit Vaimo’s knowledge to certain areas.

Though, as the time went, there were a lot of issues discovered along the road, especially with the development process and performance of client web shop, as they were slowing down in various scenarios. The main reason was that, Magento 2 evolved from Magento 1, which inherited the presence of other old web frameworks and libraries, which are less known, and this led to the lack of knowledge and experiences for developers to solve many technical cases. Furthermore, there are projects where the logic for the presentation on client side is extremely hard to implement, leading to the waste of thousand hours of workload, and generally lowering company sales. Heading toward modern web technologies such as Progressive Web App, ReactJS, GraphQL, is one of the solutions Vaimo is approaching to improve the development and production process, as the benefits gained from those stacks are undeniable, in both theoretical and practical scenarios. The upside and downside of those new web stacks will be debated in later chapters of this thesis.

As the needs of shifting toward new frameworks and libraries, Vaimo Product Team has assigned the responsibility to create a general UI library that exploit modern web stacks, ReactJS and GraphQL, to Vaimo Finland Oulu office. From mid-summer 2018, Magento 2 has been actively developing on the Progressive Web App solution, and Vaimo saw the opportunity to continue their old original idea, and planned to have their own UI library that consists of general UI components, which then served as a base for future projects that would deliver Progressive Web App and modern web stacks to the client’s web store.

The Vaimo UI library is expected to be a consistent, extendable, general and reusable UI library, which takes a huge responsibility for improving the user experience and performance of client web shop. This thesis will discuss and describe the author’s responsibility in the process of setting up the library, as well as his role in writing composable components for the library, and all the objections on the way to create such library of which then experiences are shared, and the conclusion is given on the benefits of this project.

2 GENERAL THEORY

To start setting up the UI library, developers were involved in need to deeply understand the infrastructure of Magento 2 as well as the modern web stacks. In this chapter, the definition and the downside of old libraries that Magento 2 is currently using, and the advantage of adopting the modern web libraries will be explained and discussed.

2.1 Overview of Magento 2

Magento 2, the PHP framework which mostly serve for a web ecommerce store, is one of the leading frameworks to build a web shop. What are offered from Magento 2 are a CMS system (Conten6t management system) for merchants to manage their own web shop, such as adding new products, adding promotions, and production ready components for a complete web shop implementation.

Magento 2 conducted their services and system through the figure 1 below (1)

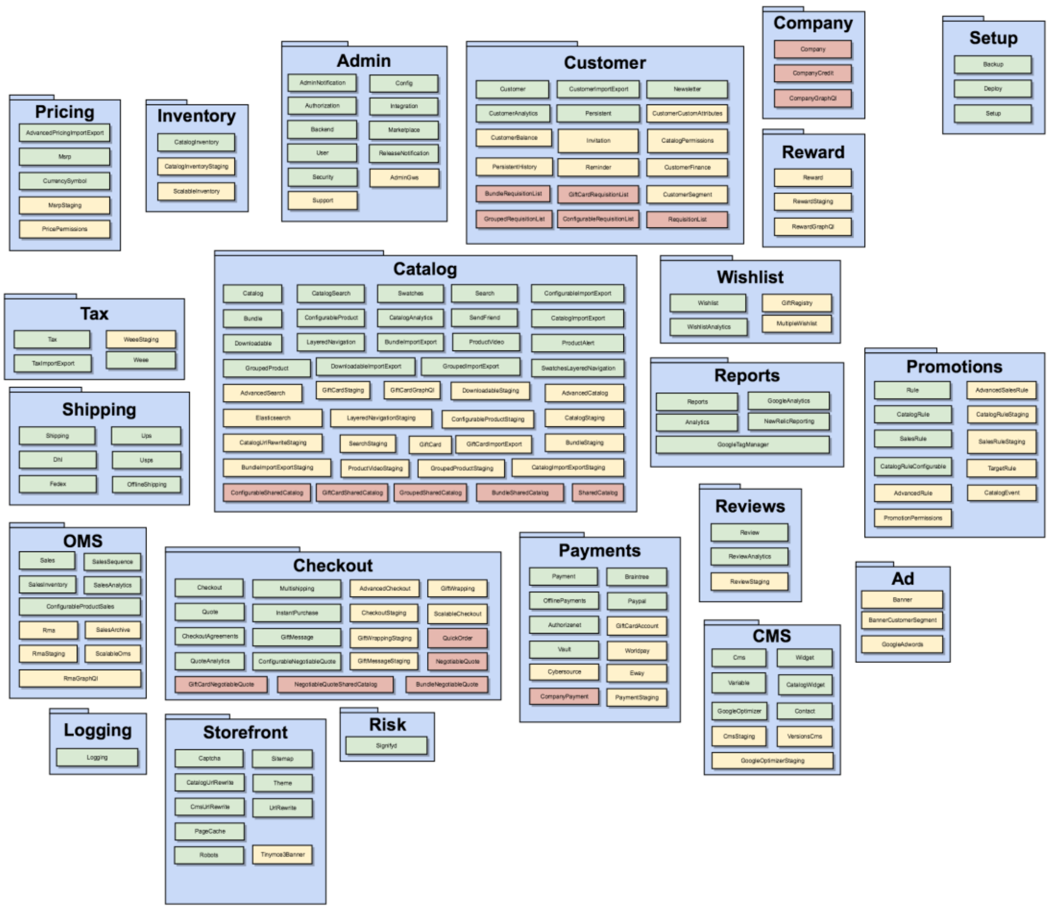


FIGURE 1. Magento 2 services

2.2 The Magento 2 Frontend system

2.2.1 Adaption of MVVM architecture in Magento 2

Magento 2 involved from its brother, Magento 1, even from its model structure. It is used to be an MVC (Model-View-Controller) system in Magento 1, but in Magento 2 instead it is stacked with a Model-View-View-Model system.

At the beginning of Magento 1, developers usually referred to the MVC architecture as a system to be considered when starting a new project, since this system is extremely popular and get implemented by many frameworks, such as Rails, Django. The term itself has three clear and separated sections: Model, View and Controller. The Model will focus on data processing with a database, transferring data fetched and connecting to the Controller, which is the middleware of the Model and View. The View is the presenter, the layout to present data to the client. The architecture became popular among the community since the code base could be easier to structure and maintain, thus leading to a great chance of reusability. Also, the logic is separated into each individual component leading to coordinate those logics easily without worrying much about the architecture. However, in Magento, the View component would directly get the system models in Backend to get the data it needs to present, instead of going through a controller. The View in Magento is also a bit more complicated than the traditional MVC system, as Magento also separated the logic into blocks, layouts and containers to present. The layout, using an xml file to describe the structure of the webpage, defines the container that provides which content belongs to where, and the block is the UI components renderer to present the view. This approach helps the Magento View logic to be extendable, easy to modify and has brought Magento a huge success since then.

Though, Magento 2 has shifted toward the MVVM system. The main difference is that, the block acts as the View Model instead of simply being the middleware between Model and View. “Unlike the MVC method, the ViewModel isn’t a controller. It instead acts as a binder that binds data between the view and model. Whereas the MVC format is specifically designed to create a separation of concerns between the model and view, the MVVM format with data-binding is designed specifically to allow the view and model to communicate directly with each other” (2). When a request is sent to a route, the MVC system will send the request to Controller to process data, to View, Model, or both View and Model, while the MVVM system sends it to the ViewModel instead. The ViewModel will take care of selecting the right layout, managing data from the request and sending it to the Backend to decide what to response, or even redirecting user to another route. It means that the ViewModel works with both the data and the layout that present those data, instead of passing logic around the system like Controller in MVC. This helps web application to have a quick way to save data into a database and an action could be forwarded quickly like a Single Page Application (a page that loads a HTML mark-up and updates the web application dynamically on the user action). This use case could be found in many places in Magento 2, adding product to cart functionality for instance, as data is bound and passed between multiple sections, such as a cart, a mini cart, and a product checkout page.

2.2.2 Replacement of the old stacks in Magento 2.3.1

The modern web stacks come into Magento 2.3 to replace and improve implementation of the current Magento 1 and Magento 2 application libraries, frameworks and model: Frontend theme, Jquery, KnockoutJs and REST API.

The Frontend theme, or Magento theme, is just a component that describes how the layout of the store looks. The normal Magento 2 web application comes with 2 ready themes, Blank and Luma, which would be a base theme to inherit from when creating a new custom theme. All frontend code that is related to the theme, such as creating a new block, a new template, or modifying the Magento 2 parent theme, will sit under the theme folder.

KnockoutJs is a MVVM Javascript library which is lightweight, fast, and convenient in data binding for Frontend developers to work with when building a Magento 2 web application (4). At the time of developing Magento, it was chosen because the MVVM system matches with Magento design, and the library is fast and lightweight. However, the reality is quite different when working with it in Magento 2. The main use case of the library normally sticks to Magento checkout, as there are many forms, such as billing form and address form, and many inputs that need data binding. Otherwise, the library is rarely used on any other pages such as category, product page, search page. In addition, finding the source of truth where the data comes from while passing among the Model, View and ViewModel in KnockoutJs is always a pain to Frontend developers, even with experienced Magento 2 ones. One more issue Knockout would get “knocked out” in the newest Magento 2 version, 2.3, is that the library could not control the state of data easily. For a simple web store, this might not be an issue, but in a bigger project that has many customised modules and requires many complex logics under the hood, the codebase would massively become a mess when implementing with Knockout, thus reducing and decreasing the code quality, maintainability and reusability.

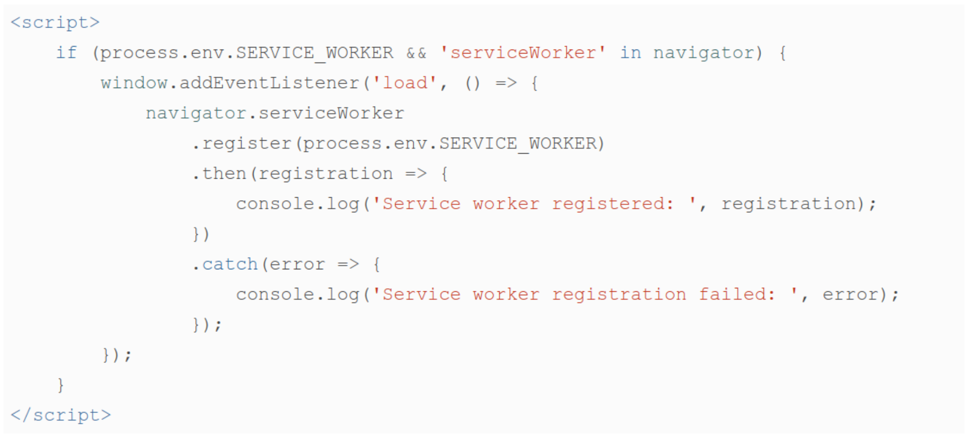
REST API is quite opposite compared to the above frontend tool, Knockout, as the technology is not yet outdated and could still be exploited in some manners. REST (which stands for Representational state transfer, introduced and defined in 2000 by Roy Fielding in his doctoral dissertation) (5), has been a supreme king when referring to the method of connecting with an API, Application Programming Interface. Every time a user requests to a Web URL, for instance vaimo.com, the user is talking with an API, located somewhere in Vaimo server, that could understand the user message and send a response back. In this case, REST acts a layer between the user and the Vaimo server, understanding the user request and sending it to Vaimo server. With REST, all the users that reach the same API endpoint will communicate with the server in the same way and get the same response every time. This also means that REST is stateless, as it does not save or record any message/request client send to the server. The same logic applies to any Magento 2 storefront, REST API main responsibility is to communicate with the client and the web store server, for instance, fetching product data when navigating to a product page.

2.3 Progressive Web App with modern Web stacks in Magento 2.3

Even though Magento 2 has been improved significantly since the release day, the Magento team decided to take a leap toward new technologies and libraries, a Progressive Web App, along with ReactJs and GraphQL, as these tools are strongly believed to boost the performance of Magento 2 web store, following the best technologies in the current development process for any Web application.

The Progressive Web App itself is a new technology which gets numerous of praises from Web developers back from the day it was introduced by Google as a new future of Web development. Fathers of the technology, designer Frances Berriman and Google Chrome engineer Alex Russell, state that the progressive web app steps on new features of the modern browser, consisting of web application manifest and service worker, which help the user to interact with their app in an extremely convenient and user-friendly way (6). The word “Progressive” has already stated that the app will work for every user, who is using the modern browser and OS (7), while the service worker is described as a tool that make the app work independently in any network condition, meaning that the application now could work even if the user has no Internet connection.

In addition, the mobile user can even add the application icon on their mobile home screen and serving as a loader of those static files and caching them. The service worker is a script registered in the browser background, separated from application logic and open to a network request to synchronize the application state. A typical service worker implementation could be added into the application shell, registered as a piece of Javascript code, shown in figure 2 below**.**



```
<script>
    if (process.env.SERVICE_WORKER && 'serviceWorker' in navigator) {
        window.addEventListener('load', () => {
            navigator.serviceWorker
                .register(process.env.SERVICE_WORKER)
                .then(registration => {
                    console.log('Service worker registered: ', registration);
                })
                .catch(error => {
                    console.log('Service worker registration failed: ', error);
                });
        });
    }
</script>
```

FIGURE 2. Basic implementation of service worker in the application shell.

For getting registered by a small piece of code, though, the service worker lifecycle is completely separated from the web application. After having registered, service worker is installed on the browser, and the cache process would be started. Any static file that developers want to cache will be downloaded and cache, and those files will be controlled by the service worker. The service worker will then be terminated to save memory, or it will manage data fetching and message events when there are client requests. The lifecycle of a service worker could be seen from this figure 3 below (8)

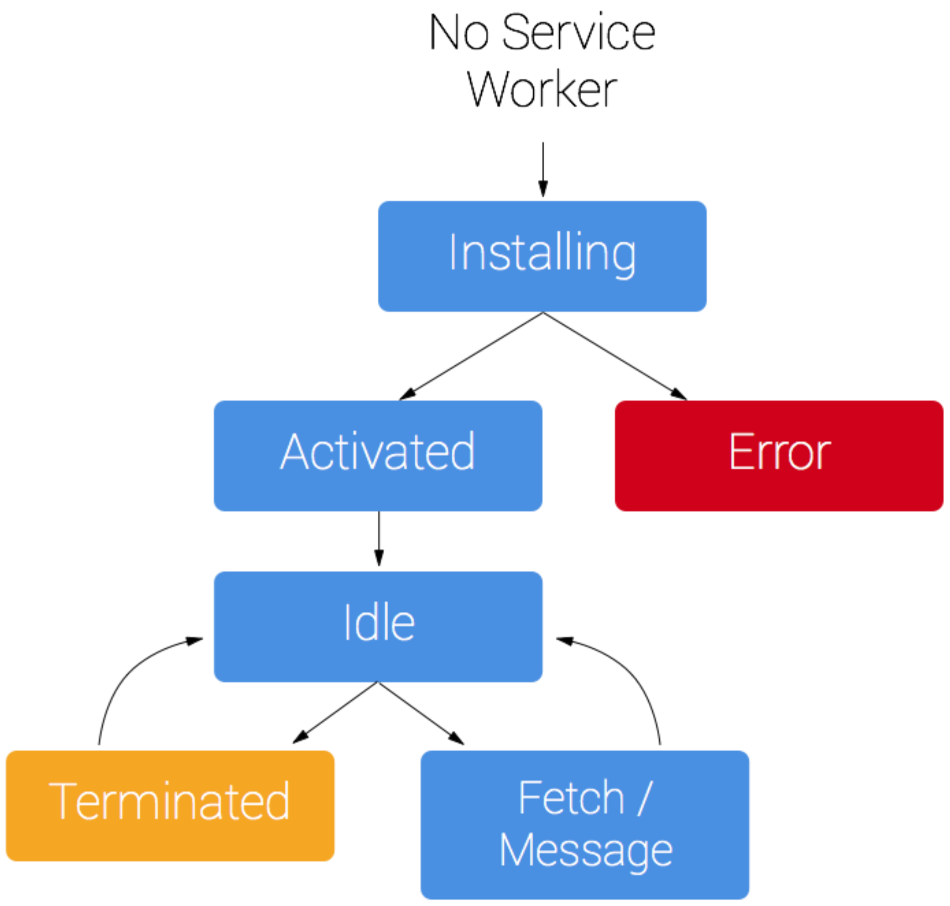
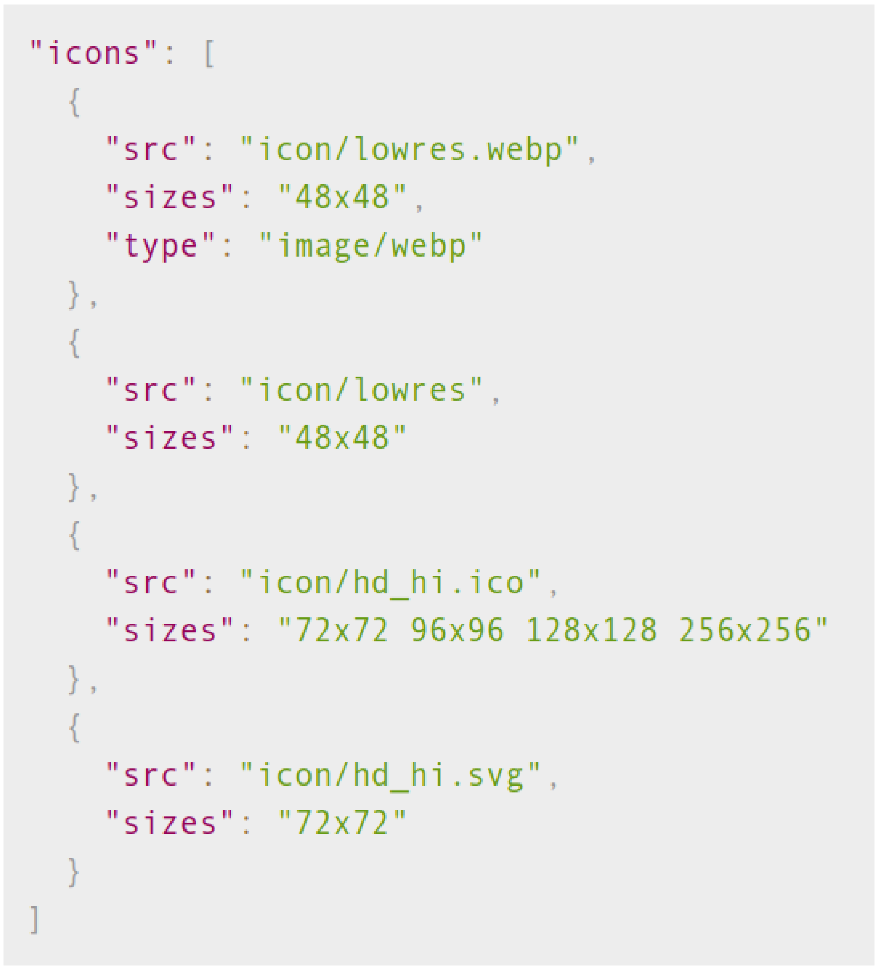


FIGURE 3. Service worker lifecycle.

Progressive Web App collection also contain web app manifest. The web app manifest provides basic application information, such as its name, author, and icon, in a JSON text file. With this information stored inside the web app manifest file, the web application can then be added on the user’s mobile home screen. Figure 4 (9) provides some basic information that a web app manifest file contains:



```
"icons": [
  {
    "src": "icon/lowres.webp",
    "sizes": "48x48",
    "type": "image/webp"
  },
  {
    "src": "icon/lowres",
    "sizes": "48x48"
  },
  {
    "src": "icon/hd_hi.ico",
    "sizes": "72x72 96x96 128x128 256x256"
  },
  {
    "src": "icon/hd_hi.svg",
    "sizes": "72x72"
  }
]
```

FIGURE 4. Icons that used as a home screen icon on a PWA

As of those benefits brought from using the Progressive web app, it predictably becomes the future of Web application. Huge giants in web app development such as Pinterest, Twitter, Netflix, etc… have already switched to the Progressive Web App and are gaining undeniable success in both the development process and sale records.

There are many case studies that show how Progressive Web App improves user experience, thus increasing the time the user interacts with the application and increasing in the sales. A case study research about Pinterest's implementation on PWA illustrates those improvements, through figure 5. (10)

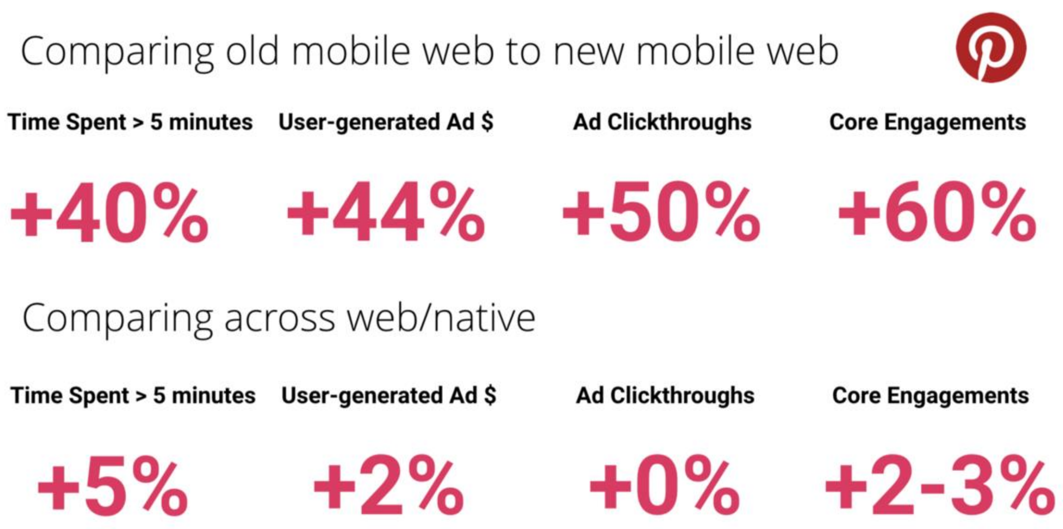


FIGURE 5. Pinterest improvements after switching to Progressive Web App

ReactJs is backed by a technology giant, Facebook, having a huge community that actively commits and submits pull requests every day to the open source project, having one of the highest stars in Github, the world leading development platform. Having been already praised from the day of introduced by Facebook in 2015 as the best library to use for presenting a web application, ReactJs brings many great architectures with it and also defines the standard of writing a web application, such as exploiting the new technique to control the application state, data binding, writing maintainable, testable and reusable web components. As ReactJS is a component-based library, which helps developers to separate the concern of the web app into small pieces that have own functionalities, thus making those components can be abstracted, reusable and easier to maintain. The biggest issue that old libraries and vanilla JS have is the state change management, which is solved by this component-based technology. Alberto Gimeno, Mar 27, 2018, stated that with the old way of implementing the state for the application, the code is fragile and not ready to receive many changes in the application state (11). Another feature that ReactJS brings in with is the use of React Virtual DOM (12). In the old way of doing DOMs (Document Object Model)(13) manipulation and managing the old DOMs state, the code base will be extremely messy as there needs to be a reference to the specific DOM, and after the DOM element needs to change its state, the whole DOM element will be refreshed or recreated. This would affect significantly to the performance of the web page. As the React Virtual DOM will create a copy of the original DOM element, which has all the immutable properties of its origin, React Virtual DOM does not get painted on the browser page, thus making the improve on the performance compared to the old technique. It also has its own debugger extension, so developers can debug their code inside the same working environment and browser, thus improving significantly the development process. This library will be a complete replace to the old friend, KnockoutJs, in the use cases mentioned.

GraphQL, on other hand, is defined as "*a data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data*". (14) GraphQL had been developed internally by Facebook from 2012 before being publicly released in 2015. It has already been recognised as a new technique to be used to get data in the most efficient and effective way and it would become the best replacement for REST API in many use cases in the manner of data fetching. However, currently, Magento 2.3 has not been able to fully support GraphQL, as this would take plenty of time and effort. There are some ready GraphQL based module, such as module-catalog-graphql, module-category-graphql, where a GraphQL schema is defined and resolvers are ready to return data to the client on the GraphQL query request. Taking the product graphql module for example, a resolver's main role is to construct the query (the matching type of the query to be a Product type), fetch data from the Product type table or any other sources, transform that data into a GraphQL array format, and return the result in a callable function in a resolver class. server side should have a schema that define that data query (like a way to understand the query in user request). A schema, on the other hand, will include the query structure, whose attribute is available for GraphQL queries, as well as the connection between the Schema and its resolvers that verify and process the query data and response. However, it is crucial to note that, Magento 2 would still use REST in some modules (e.g the Catalog module), and exploit GraphQL (like CustomerGraphQl module) in some manners that could reduce the size of response. The figure 6 (13) here shows the technical vision of Magento 2 in adoption to the Progressive Web App.

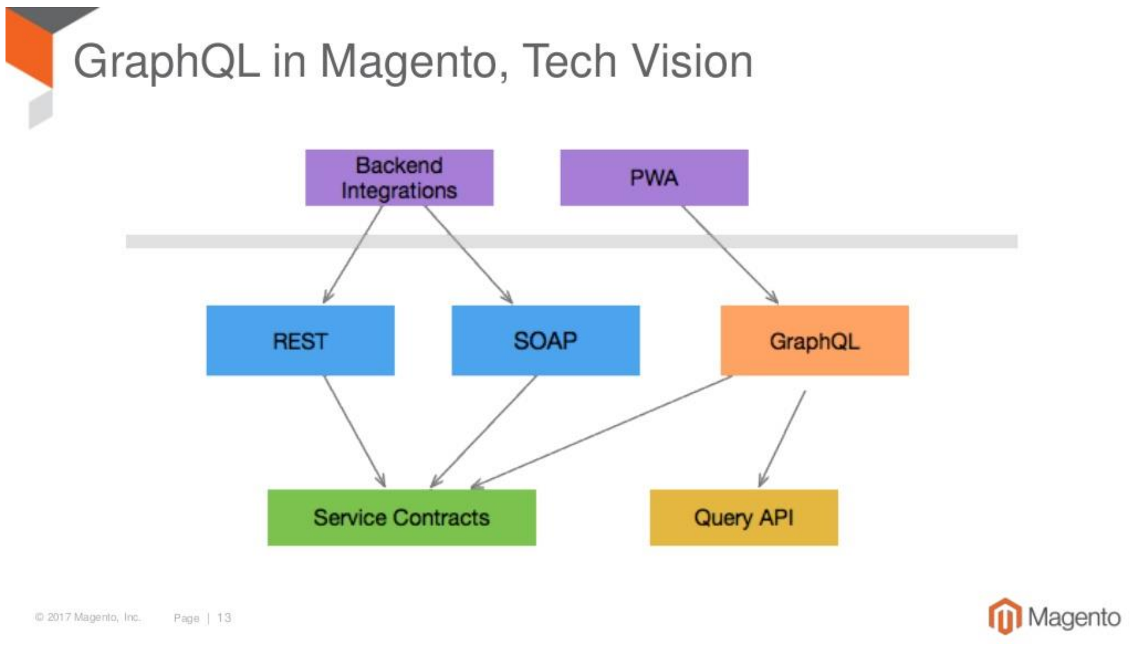


FIGURE 6. GraphQL in Magento 2

A clear ambition from Magento 2 on how the system architecture would look like with the use of those modern tools could be shown below in figure 7 (15), standing for the old implementation and figure 8 (16), standing for the ideal system that Magento 2 desires to have. There are plenty of huge differences between these 2 systems. However, it would be out of scope for this topic to cover all the changes, as this thesis will mainly focus on the Frontend part. How these tools are replaced to the old tools, their usage, advantages and drawback will be considered in a separated chapter later in a detail perspective.

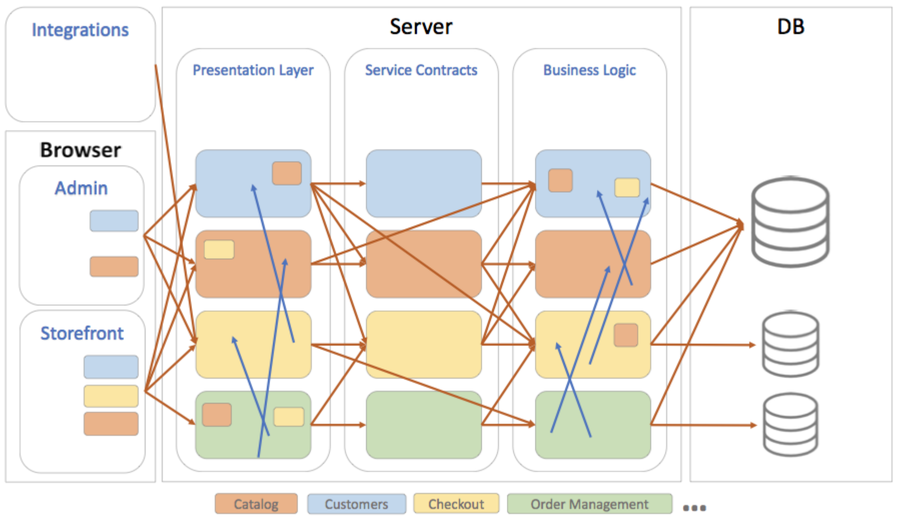


FIGURE 7. Current implementation of Magento 2.2

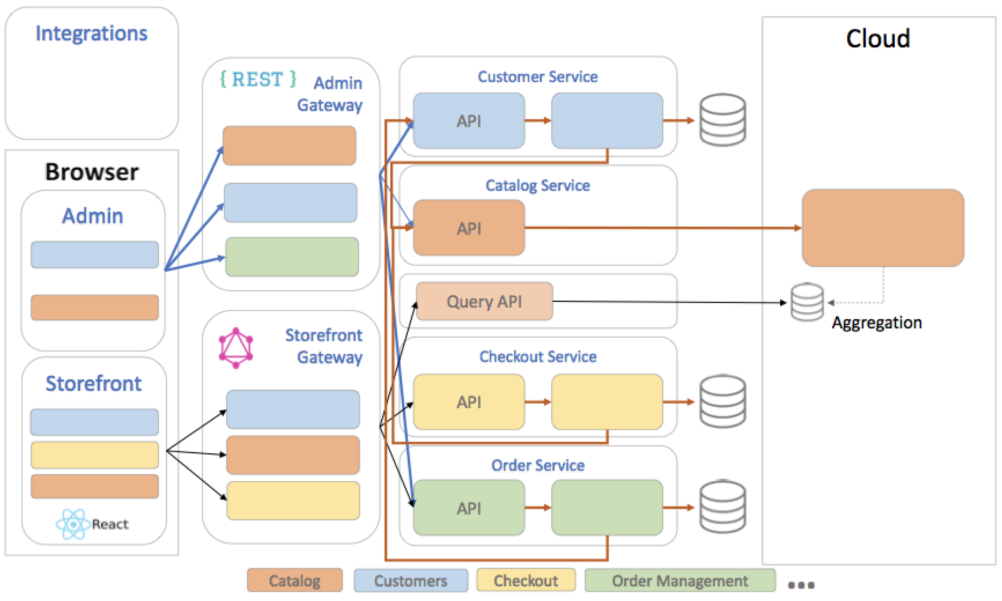


FIGURE 8. Implementation in the future of Magento 2.3

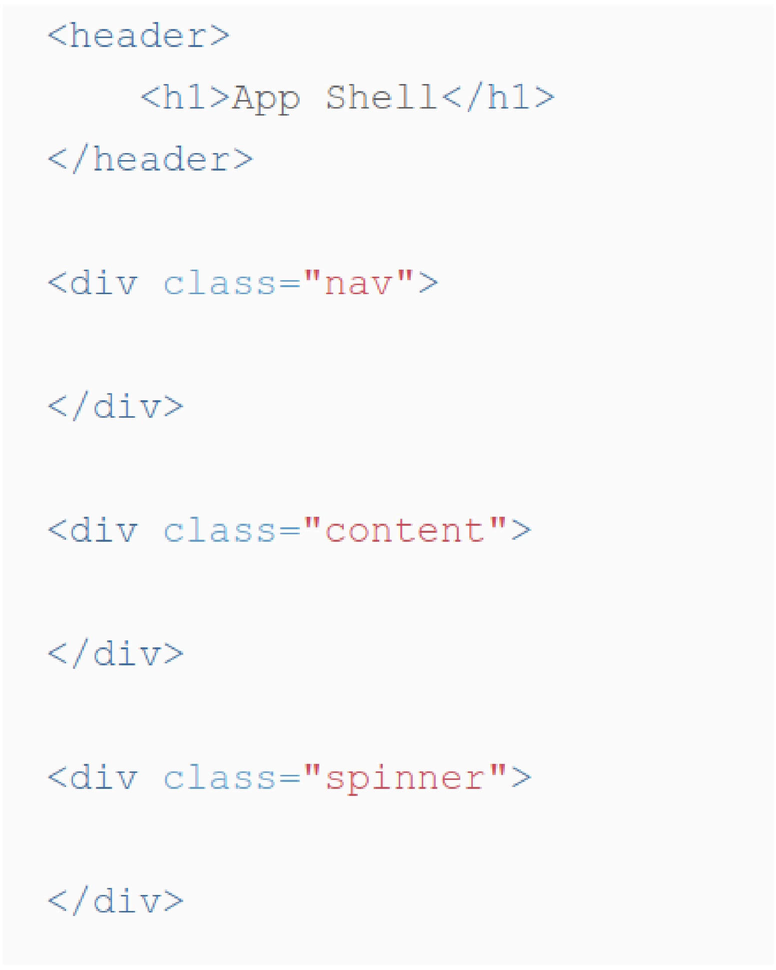
Due to Magento 2 design, for the current state and design state, there are many differences to be spotted. All communication between the server and the storefront will be done by the GraphQL query request in the frontend, getting populated and presented by ReactJS, and a REST API request for admin merchant, instead of just using a REST API request to retrieve data in current implementation. Server implementation in the desired state also indicates that there will be no more presentational layer, as data will be fetched from the server by GraphQL or the REST API can be rendered in the client side (client-side rendering is much more concentrated than just pure server-side rendering). There are also some services like checkout and order that will exploit the use of GraphQL to speed up the request on data, and Query API will be sent to the cloud service (Magento has its own cloud).

2.4 Magento 2 PWA Studio System Overview

Magento 2.3 introduces the Magento 2 PWA studio, as an open source project with a collection of tools that gives developers the power to build complex Progressive Web Applications on top of Magento 2 stores. The aim of PWA Studio is to provide developers and agencies complete control over the Magento Frontend workflow, from the theme's structure, modules setup, and project infrastructure from start to completion.

The approach of Magento 2 PWA studio is a Progressive Web App, which has the main components: application shell, service worker, ReactJs , GraphQL, Css modules, Redux and a web app manifest.

The PWA studio application shell contains logic that helps with caching HTML, CSS and Javascript files, so the web page will always have a minimal content and a UI client for the user to interact with while the page is loading, thus shortening the time loading and improving the User Experience (UX) of the web application. The rest of the content that are missing will be fetched from the API or some other ways.



```
<header>
    <h1>App Shell</h1>
</header>

<div class="nav">

</div>

<div class="content">

</div>

<div class="spinner">

</div>
```

FIGURE 9. Application shell

Though, the content shown in figure *9* (17) above is just a plain HTML document that serves as a fallback while the data is fetching and does not have any logic that helps caching static files mentioned above. There comes the service worker, which is one of the best features PWA provide, explained above. By registering the service worker, the Magento 2 application can now use in offline mode (without the Internet connection).

ReactJs and GraphQL were discussed before, though it is also important to note the main job of these 2 technologies in PWA studio. Their main role is to bind and dispatch data in components, between the client and the server, with the help of RootComponents, Components, and React Router. Root components is just DOM nodes under the hood, where the basic component of a web store is implemented, such as Product and Categories, while

Components are the React Component that will render data and logic in the UI, with the help of React Router to specify the routing of the application.

Along with them, Redux serves as a state management tool that contains all the logic for dispatcher, reducers and actions. Redux is implemented based on the architecture of creating a global store to manage the state object for any Javascript application. Whenever a user interacts with the UI that require the state management, such as adding a new product to cart, the **action** will be sent to **reducers** and then updated in the **global Store** that has all the application states and updated the UI asap. Figure *10* (18) indicates how Redux works as a store with state management.

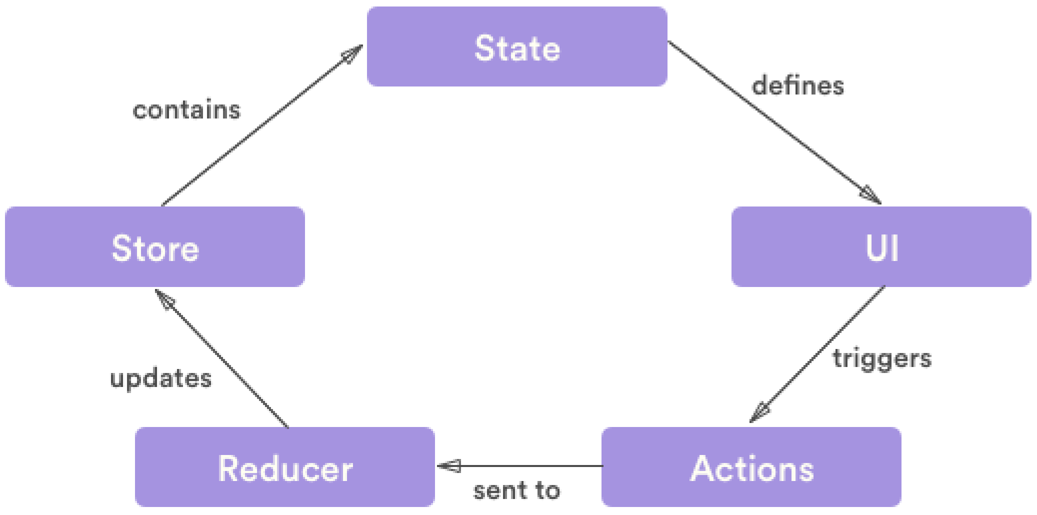


FIGURE 10. Redux logic involves around state management

CSS modules is a new technology to write clean and maintainable CSS code, by adding individual scope to CSS classes in a React component. By doing so, not only ReactJs class component separates the concern on the UI, but also the styling aspect of that respective React component is managed in a more granular way, by spreading hash to the end of a class name inside a component. The normal CSS class name would become a unique format, that would not get repeated anywhere else in the application. Thus, the technique allows developers to build the layers of styling in a modular way and negate the bad practices from using normal CSS that exploit global styling, means that all class names would be available in the whole web application. Though, this technique requires a tool to transform its code into the browser understandable language, and Webpack is usually a tool that pairs with it. CSS modules currently has been added as a dependency in create-react-app, one of the most famous tools to create a new scratch React application backed by Facebook.

Finally, the PWA studio also contains **Webpack**, which acts as a static module bundler for a modern Javascript application. In brief, Webpack scanning through the application dependencies and packages, creates a dependency graph that contains all various packages that the application asks for, then act as a build tool to wrap all the static files, bundling and chunking them into files which have the browser understandable language. It is most known for its wide range of configuration options to initialize tool loaders, such as **Babel** (modern Javascript code and features compilation tool), **CSS modules. Webpack** also takes care of creating a development server with custom options from **pwa-buildpack dev server** (explained in the next chapter)**.** Configuration options for **Webpack** will be discussed later when the UI library is in development.

2.5 Magento PWA studio packages

Magento 2.3 introduces the Magento 2 PWA studio, as an open source project with a collection of tools that gives developers the power to build complex Progressive Web Applications on top of Magento 2 stores. The aim of PWA Studio is to provide developers and agencies a complete control over Magento Frontend workflow, from the theme's structure, modules setup, and project infrastructure from start to completion.

The PWA studio comes with a set of tools that help developers to easily setup a running demo PWA project and can be considered as the complete and great structure code base for a new project. There are 4 main packages in the current PWA studio repository: **pwa-buildpack**, **peregrine**, **upward-js**, and **venia-concept**.

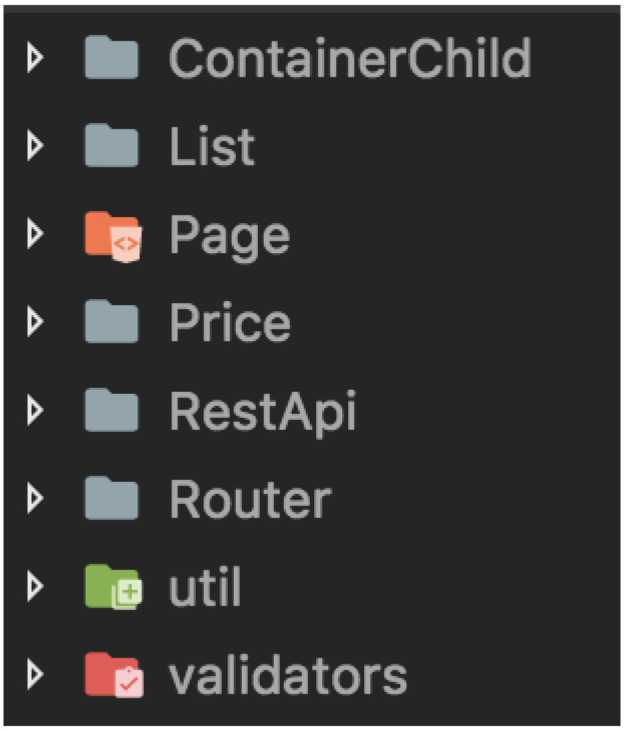
2.5.1 pwa-buildpack

This package takes the advantage of Webpack to serve as a building, deploying tool for the progressive web app in the development and production environment, “pwa-buildpack”, contains:

- **PWADevServer**: which creates a development server for the development process, by creating an SSL enabled server in the local host environment, with service workers wrapped inside.
- **MagentoResolver**: An adapter that configures Webpack to resolve assets.
- **MagentoRootComponentsPlugin**: This plugin helps to split **Root Components** into unique chunks with **Webpack**, mostly for optimization when bundling the code base. **Webpack CommonsChunkPlugin** moves commonly used libraries across routes up to a single bundle file, thus significantly improving the load speed of some components, and the whole web application in general.
- **magento-layout-loader**: A Webpack loader that compile JSX syntax (React Javascript syntax extension) to a normal Javascript syntax so that browser could then read and understand.
- **ServiceWorkerPlugin**: A wrapper around the **Google Workbox Webpack Plugin** (19). In short, it helps disabling and debugging service workers in development mode

2.5.2 Peregrine

Peregrine, on the other hand, is a set of React components which were created to handle Magento-concentrated functionalities such as routing, layout handler. It also acts as a bride between root components and other components, and a tool to communicate with the Rest API. The content of Peregrine is demonstrated below in figure 11.



```
ContainerChild
List
Page
Price
RestApi
Router
util
validators
```

FIGURE 11. System overview of Peregrine

2.5.3 upward-js

Upward-js, or the **upward server**, is the bridge to connect client-side Frontend implementation with Backend implementation by **GraphQL** or **REST API**. As shown from the figure *12* (20) in Magento research design, any request comes from the client Progressive Web App will go through the Upward Server, then data will be handled in Upward before going further to the Magento Server.

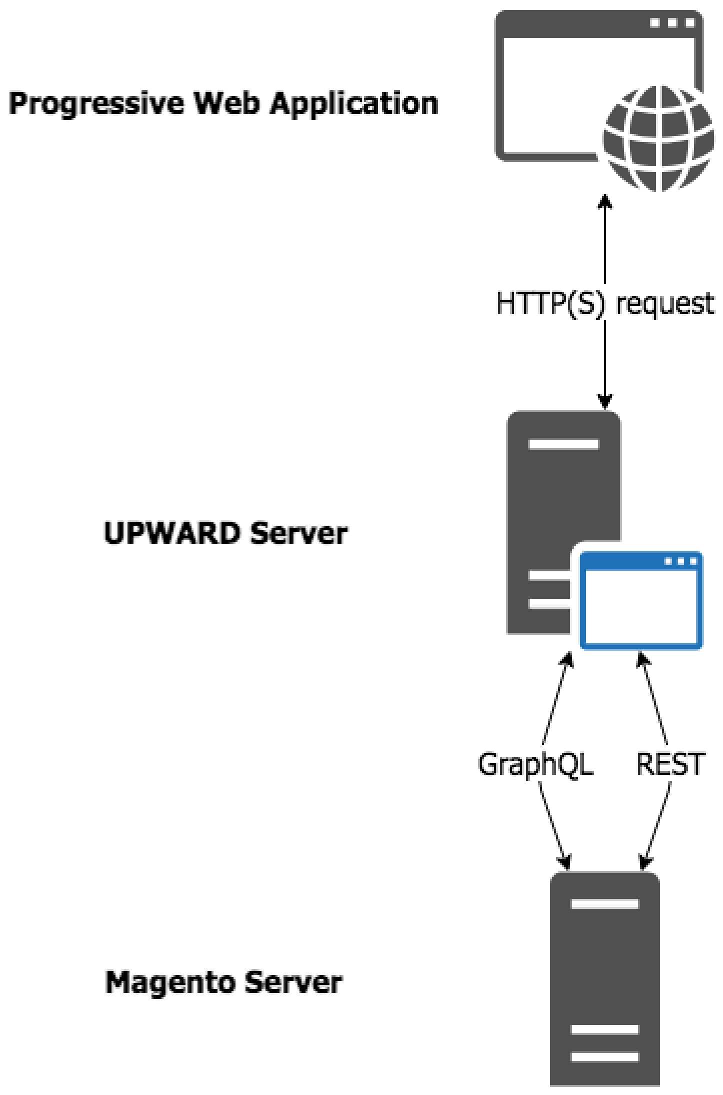


FIGURE 12. Design of data transport between Magento 2 Server and Upward Server

At the first glance, this might look unnecessary as data has to go through Upward before actually reaches Magento Server, thus wasting bandwidth, resources and time for data to be resolved. Though, the idea of creating a connection server between the application and the Magento Server is that, it reduces the usages of “backed services”. Those backed services could be found on almost all web applications for them to function properly, for example, **Mysql** service, **SMTP (**Simple Mail Transfer Protocol) service, as indicated in figure *13* (21)

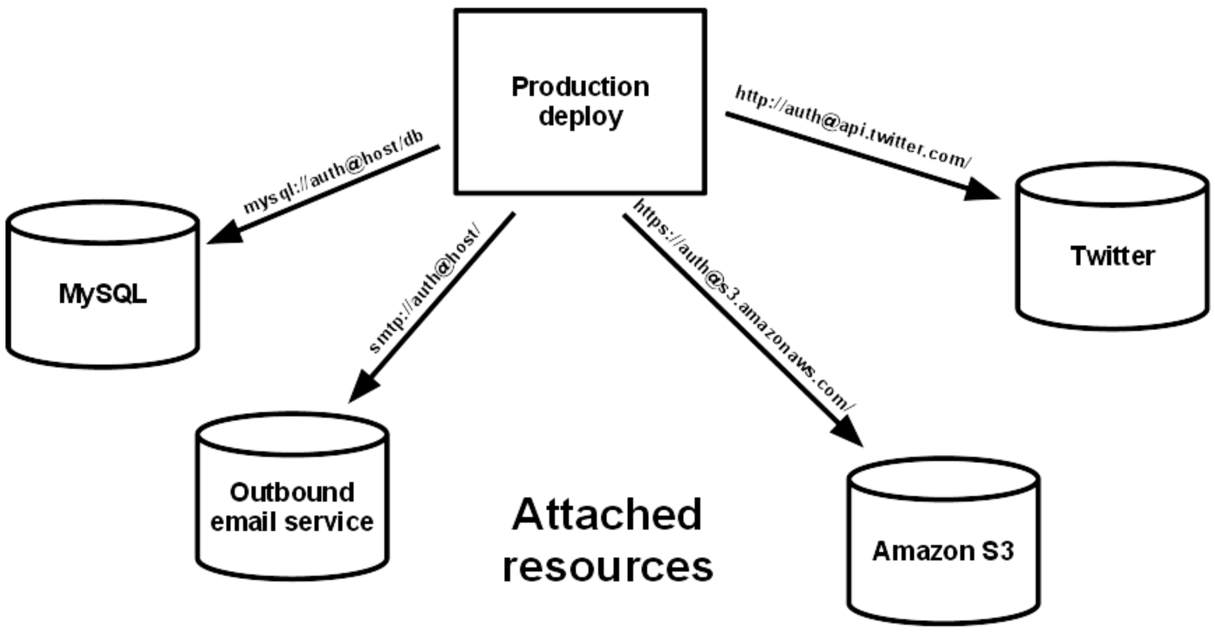


FIGURE 13. Example of backed services for a web application

The Progressive Web App must be loosely coupled from any types of backed services, as individual servers can be swapped in and out. Therefore, a utility is needed to combine all those backed services into a single layer that will deliver and synchronize with the Progressive Web App. Magento development team came up with the idea of creating such tool to unify those backed service, called “Upward Server”. The architecture now should be defined as in figure 14 (22). The server is a presentation tier, serving the app assets and reverse proxying to services and combining those backed services that the web application needs.

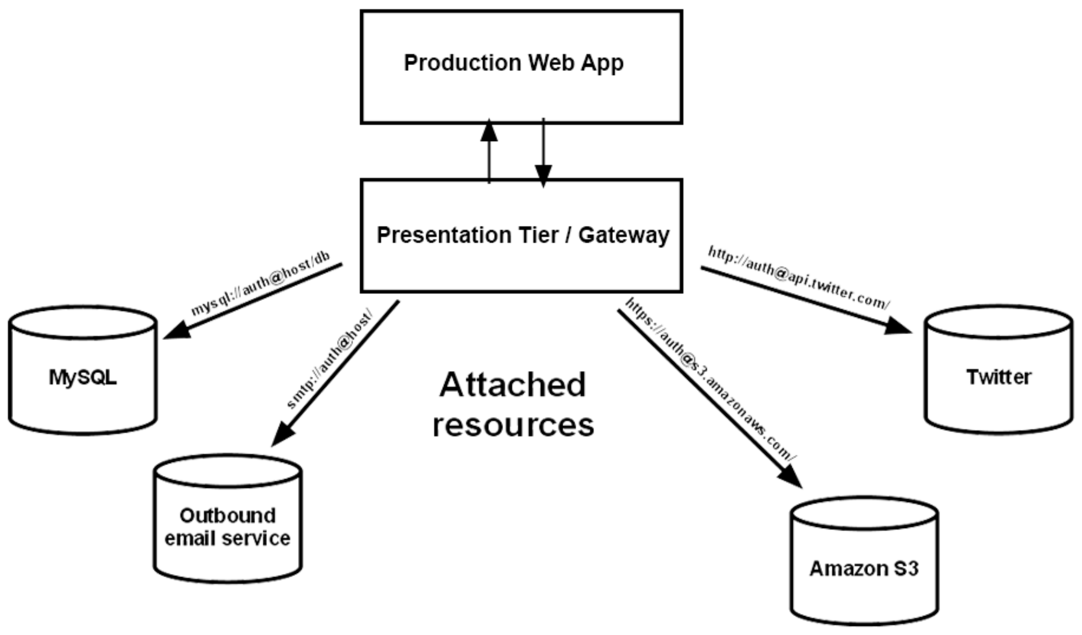
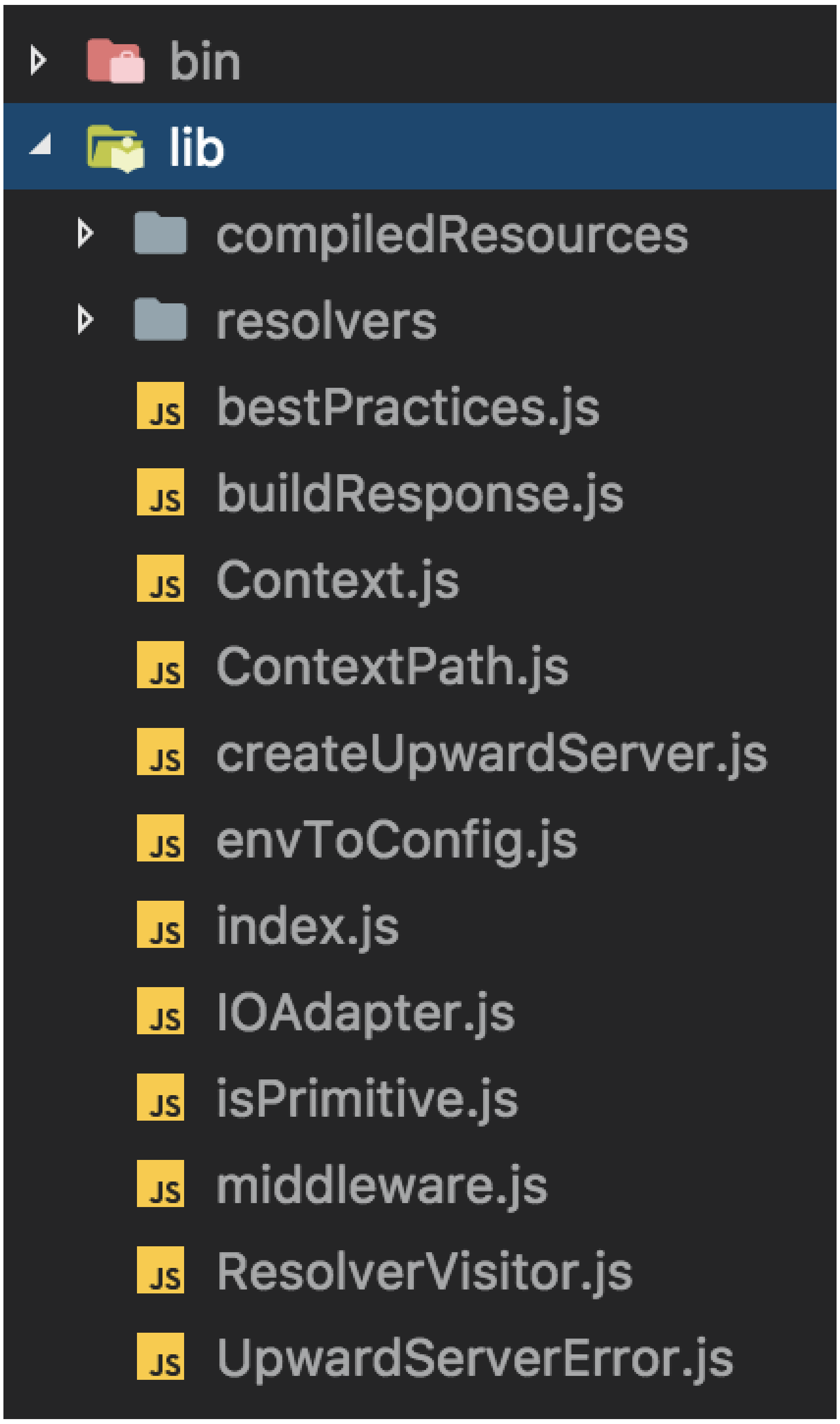


FIGURE 14. Tired architecture that has a presentation tier

Figure *15* demonstrates its project structure and file names show its core functionality, to act as a server to resolve data between the client and the Magento 2 server. As listed there, the main functionalities of the Upward server contain a server file (createUpwardServer.js), a resolver folder and an IO adapter file. Other files inside Upward are utilities files that act as method helpers for creating the Upward Server.



```
▷ bin
◢ lib
    ▷ compiledResources
    ▷ resolvers
      JS bestPractices.js
      JS buildResponse.js
      JS Context.js
      JS ContextPath.js
      JS createUpwardServer.js
      JS envToConfig.js
      JS index.js
      JS IOAdapter.js
      JS isPrimitive.js
      JS middleware.js
      JS ResolverVisitor.js
      JS UpwardServerError.js
```

FIGURE 15. Structure of upward server

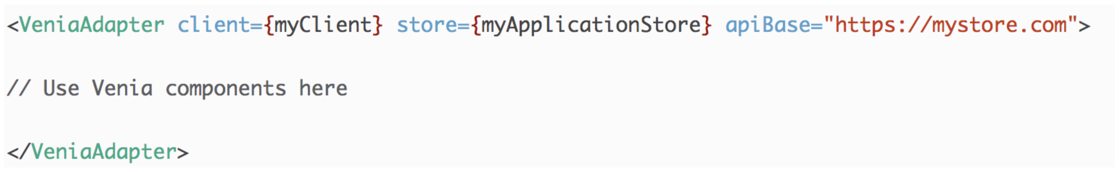
The application shell will then be defined in an upward definition file that act as a bridge between the web app and the Upward server, demonstrated in figure 16. (23)

```yaml
appShell:
  inline:
    status:
      when:
        - matches: resource.model
          pattern: '.'
          use: 200
      default: 404
    headers:
      inline:
        content-type: 'text/html'
    body:
      engine: mustache
      template: resource.template
      provide:
        model: resource.model
        bundles: assetManifest.bundles
        urlResolver: urlResolver
        env: env
```

FIGURE 16. upward.yml file contains appshell information to connect to Upawrd server

2.5.4 Venia-concept

Venia-concept is a ready theme that is built on top of the Magento PWA studio. It contains implementation for the Product Page and the Category Page by using Peregrine and pwa-buildpack. Wrapped around the Venia concept is Venia drivers, which is an adapter between the storefront theme and logic under the hood like GraphQL or a Redux store. Basic implementation of the adapter is closely similar to implementation on figure *17*



```
<VeniaAdapter client={myClient} store={myApplicationStore} apiBase="https://mystore.com">

// Use Venia components here

</VeniaAdapter>
```

FIGURE 17. Basic implementation of an adapter

More detailed information about the Venia concept will be discussed in later chapter, where the Vaimo UI library exploits most of the implementation from the Venia concept.

3 HIGH LEVEL OF COMPOSING A NEW UI LIBRARY

Before starting to develop the UI library, it is fair to compare tools that have already been implemented on top of the Progressive Web App that could integrate into the Magento 2 system. Vaimo Product Team has been working on investigating, documenting and comparing various tools. There are 3 options that are used as the base of the library: Vue Storefront, the Magento PWA studio, and Deity. According to an interview with PWA studio contributor by Inchoo, Adrian Bece has a clear view on differences between these tools. (24)

PWA Studio is currently developed and maintained by a dedicated PWA team at Magento. It is updated alongside with the Magento 2 update. It contains some built-in packages that help speed up the development process, as well as support building packages in production mode.

Vue Storefront, using VueJs as a view library, is also packed with a boilerplate theme that can be used as a basis for developing a custom theme. It also offers a complete theme that can be modified and used in the production. Therefore, it also requires certain development and customization in options.

On the other hand, Deity aims for both merchants and developers, as it offers a powerful theme editor that allows users to make fast and easy theme modifications. Though, it is only useful for merchants who just want to have a simple PWA theme without any additional development or cost of customization.

Both Vue Storefront and Deity aim to be complete theme packages with additional customization of development and extension, while the PWA Studio provides a set of tools for developers to create custom themes from scratch with the tools and technologies they are accustomed to use.

Though Vue Storefront is more mature in term of birth, having more contributors and a bigger active community compared to the 2 competitors, Vaimo Product Team preferred to use Magento PWA studio as a base setup for the library project. The main reason is that the Magento PWA studio is backed and updated by Magento core team, and it would fit more to Vaimo's needs and plans on technologies to be used in time ahead, as any update on Magento 2 could potentially affect the UI library implementation. By choosing the original package from Magento, it is a guarantee to have the newest feature from the PWA studio.

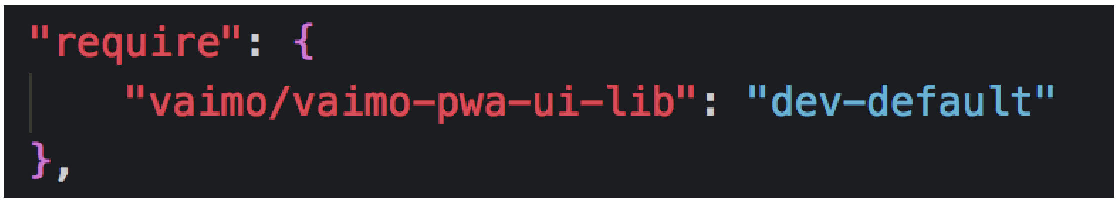
3.1 First thought and project requirements

The Venia concept could become a good base setup for the UI library, since everything include architecture, repository folder structure, and a working code base. The real challenge there is how to wrap and export the library to be used in the Magento 2 store application.

The Venia concept come along with other packages on the pwa-studio, such as Peregrine and pwa-buildpack, so it would get installed automatically inside the pwa-studio module, which then could not apply it to the case for the Vaimo UI library.

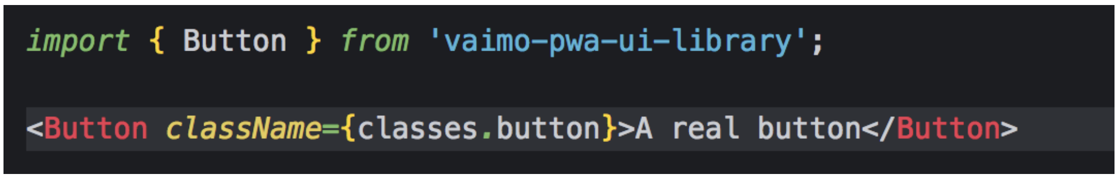
The Vaimo PWA UI library should go under a normal Magento 2 project **vendor** folder like other modules and could then be required by any component by the standard module import. A vendor folder in the Magento 2 application is where packages and modules get installed and stored by **composer**, popular package manager library for PHP and its frameworks. All files inside the vendor folder should not be edited directly, as those modules could get updated in the future, culminating in all changes to that module will be gone. Furthermore, the vendor folder should always be ignored and not uploaded into a repository, as it will rise the size of the project. Instead, project build tools will install those dependencies defined in **composer.json** file and added in the build artifact.

The desired output should look like figure 18, for **composer.json** file in the project root folder which need the PWA UI library and get imported and could be use in any other project level ReactJS component. The property of the object is the name of the package, follow by the version number, or in this case, **dev-default** to point to the default branch code of that package. The component inside the UI library will be imported to any other project later by a basic Javascript import syntax, illustrated in figure 19.



```
"require": {
    "vaimo/vaimo-pwa-ui-lib": "dev-default"
},
```

FIGURE 18. Desired state for exporting the UI library

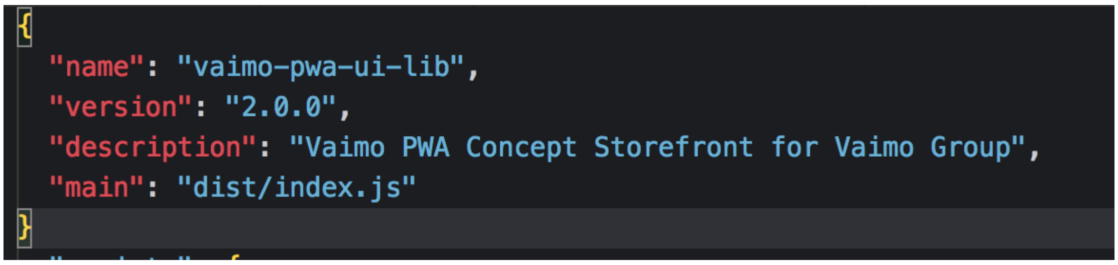


```
import { Button } from 'vaimo-pwa-ui-library';

<Button className={classes.button}>A real button</Button>
```

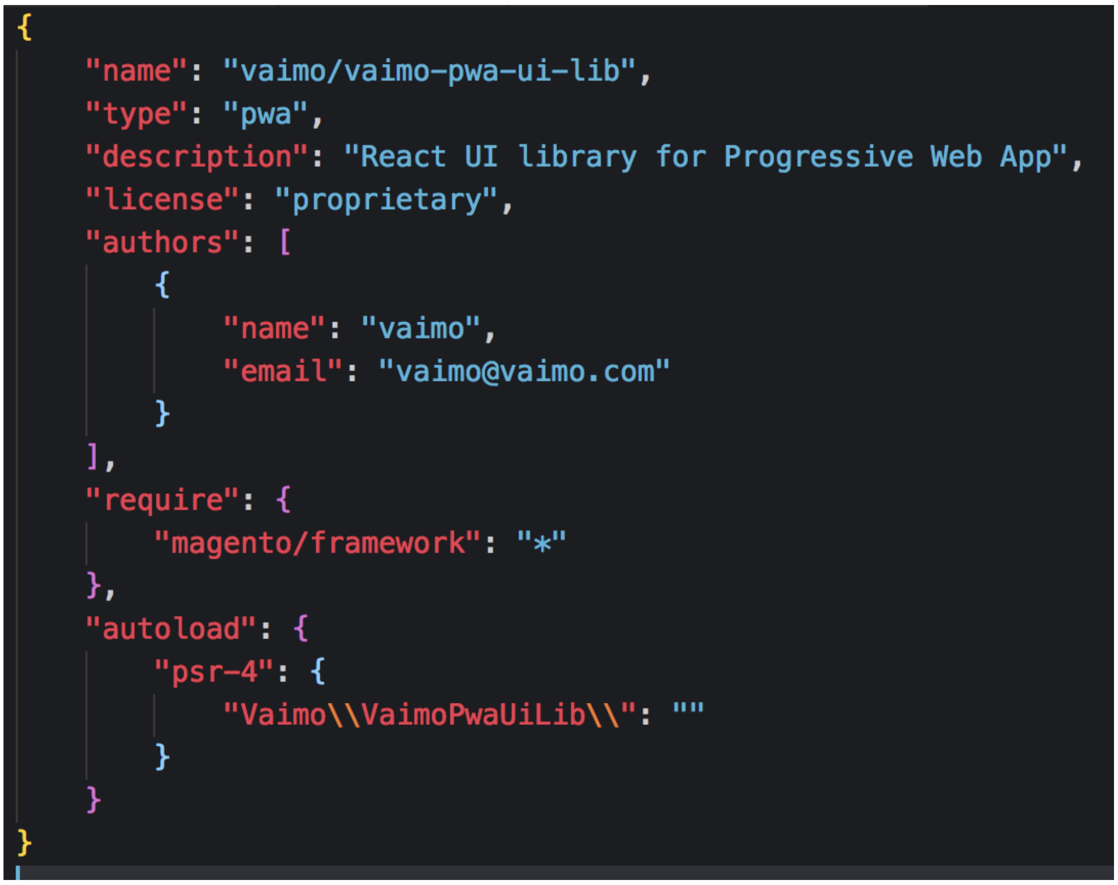
FIGURE 19. Importing the UI library

To achieve that goal, the UI component should export all its UI components and could be installed by the **composer** lirabry. In the library, a **composer.jso**n and a **package.json** file should be created, which contains all the information like name, authors, along with an **index.js** file that will export all the components which are defined in the main keyword on the **package.json** file, as shown in figure 20 and figure 21



```
{
  "name": "vaimo-pwa-ui-lib",
  "version": "2.0.0",
  "description": "Vaimo PWA Concept Storefront for Vaimo Group",
  "main": "dist/index.js"
}
```

FIGURE 20. Information on the UI library in package.json file

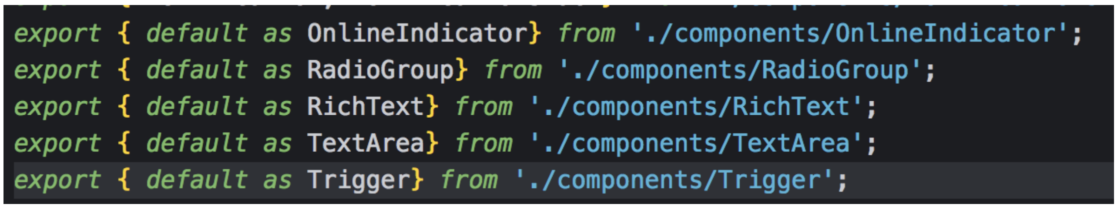


```
{
    "name": "vaimo/vaimo-pwa-ui-lib",
    "type": "pwa",
    "description": "React UI library for Progressive Web App",
    "license": "proprietary",
    "authors": [
        {
            "name": "vaimo",
            "email": "vaimo@vaimo.com"
        }
    ],
    "require": {
        "magento/framework": "*"
    },
    "autoload": {
        "psr-4": {
            "Vaimo\\VaimoPwaUiLib\\": ""
        }
    }
}
```

FIGURE 21. Information on the UI library in composer.json file

The **dist** folder, stands for distribution, in the main section of the **package.json** file specifies that it would use the distribution version of the module, which is the final version to be used in production mode.

Components are exported by the standard export syntax in Javascript, shown in figure 22.



```
export { default as OnlineIndicator} from './components/OnlineIndicator';
export { default as RadioGroup} from './components/RadioGroup';
export { default as RichText} from './components/RichText';
export { default as TextArea} from './components/TextArea';
export { default as Trigger} from './components/Trigger';
```

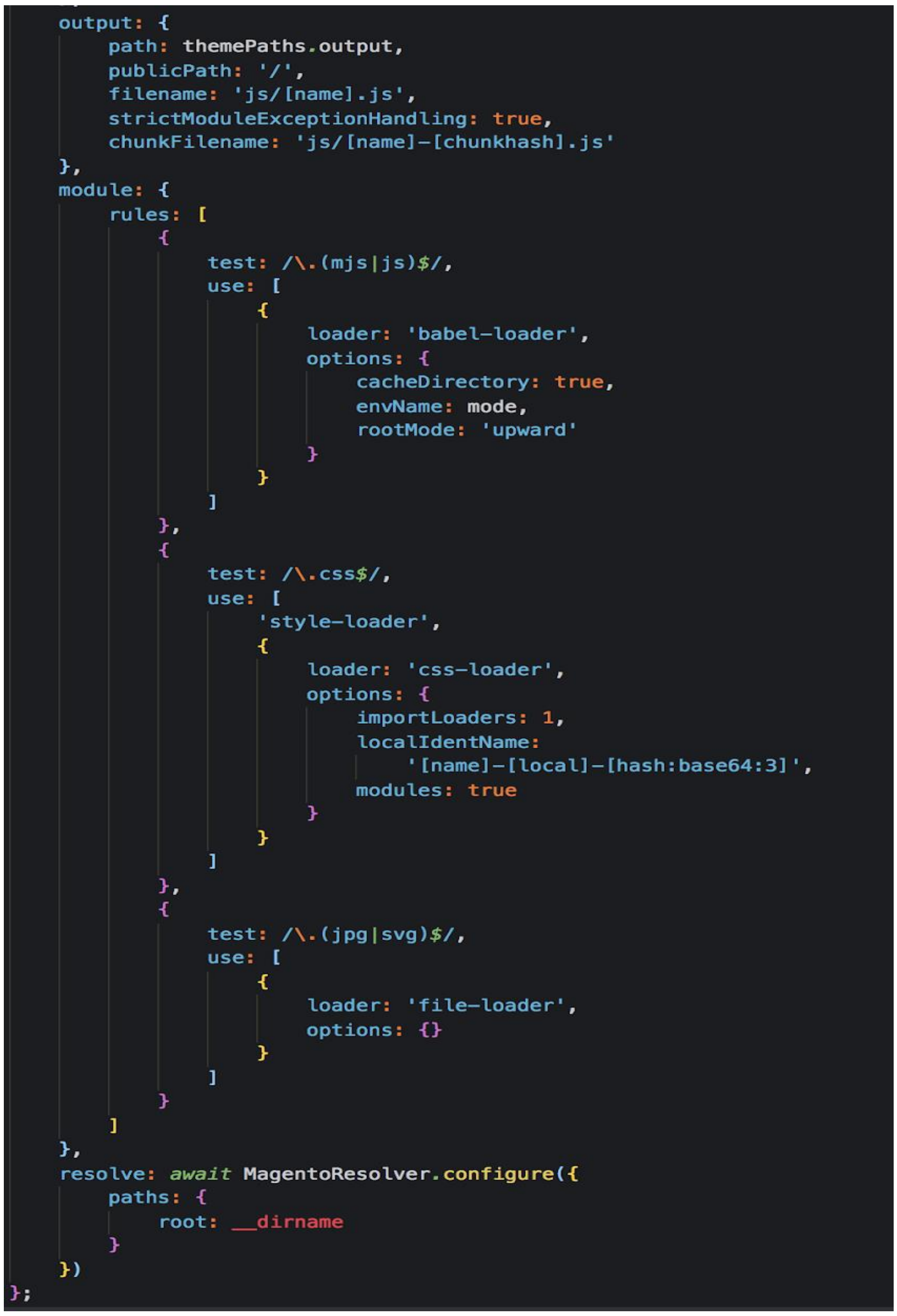
FIGURE 22. Exporting React components

The UI library repository then will be uploaded to the Vaimo bitbucket server, in the same way as other Vaimo modules.

3.2 Project packages version and build tools

After the mindset of how the library should be structured and exported/imported is set, it is time for required version for its dependencies. To be able to get the library up running, a build version is needed. It should be added inside the **dist** folder mentioned before. The list of tools is:

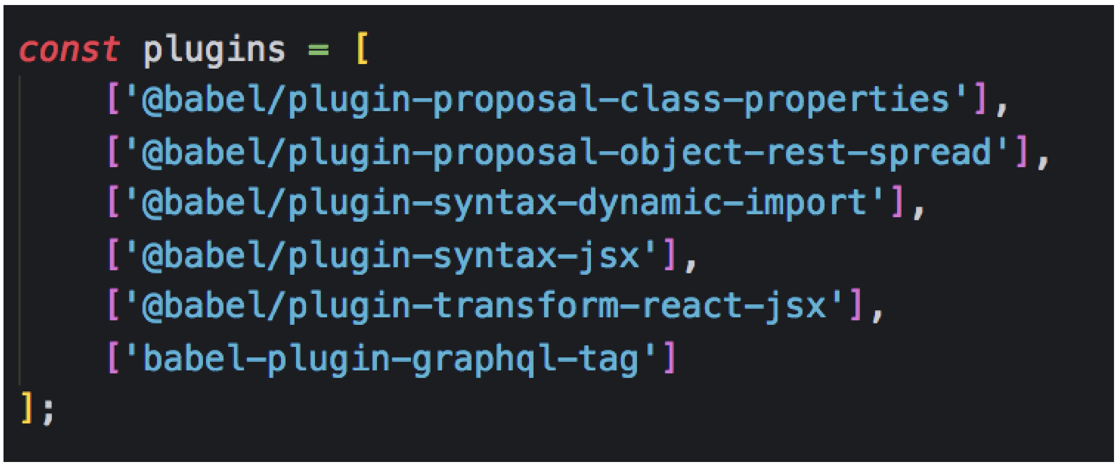
- **Webpack.** the build tool for the JS and CSS files, as well as a build tool it helps to indicate development and production mode. It contains the project environment configuration plugins to transform ES6 JS code, JSX syntax and CSS code to browser understandable languages, which are **babel-loader, style-loader, file-loader.** The **Webpack.config.js** file would also have the option to chunk Javascript files and compile each ReactJS component file to respective components. The config file will look like the one in figure 23.



```
    output: {
        path: themePaths.output,
        publicPath: '/',
        filename: 'js/[name].js',
        strictModuleExceptionHandling: true,
        chunkFilename: 'js/[name]-[chunkhash].js'
    },
    module: {
        rules: [
            {
                test: /\.(mjs|js)$/,
                use: [
                    {
                        loader: 'babel-loader',
                        options: {
                            cacheDirectory: true,
                            envName: mode,
                            rootMode: 'upward'
                        }
                    }
                ]
            },
            {
                test: /\.css$/,
                use: [
                    'style-loader',
                    {
                        loader: 'css-loader',
                        options: {
                            importLoaders: 1,
                            localIdentName:
                                '[name]-[local]-[hash:base64:3]',
                            modules: true
                        }
                    }
                ]
            },
            {
                test: /\.(jpg|svg)$/,
                use: [
                    {
                        loader: 'file-loader',
                        options: {}
                    }
                ]
            }
        ]
    },
    resolve: await MagentoResolver.configure({
        paths: {
            root: __dirname
        }
    })
};
```

FIGURE 23. Webpack configs

- **Babel.** A popular JS compiler that compiles ES6 JS syntax to the normal standard CSS for the older browser. Its loader, **babel-loader,** will be added and configured in the **webpack.config.js** file**.** An example of plugins needed for functioning a Progressive Web App is shown below (figure 24).



```
const plugins = [
    ['@babel/plugin-proposal-class-properties'],
    ['@babel/plugin-proposal-object-rest-spread'],
    ['@babel/plugin-syntax-dynamic-import'],
    ['@babel/plugin-syntax-jsx'],
    ['@babel/plugin-transform-react-jsx'],
    ['babel-plugin-graphql-tag']
];
```

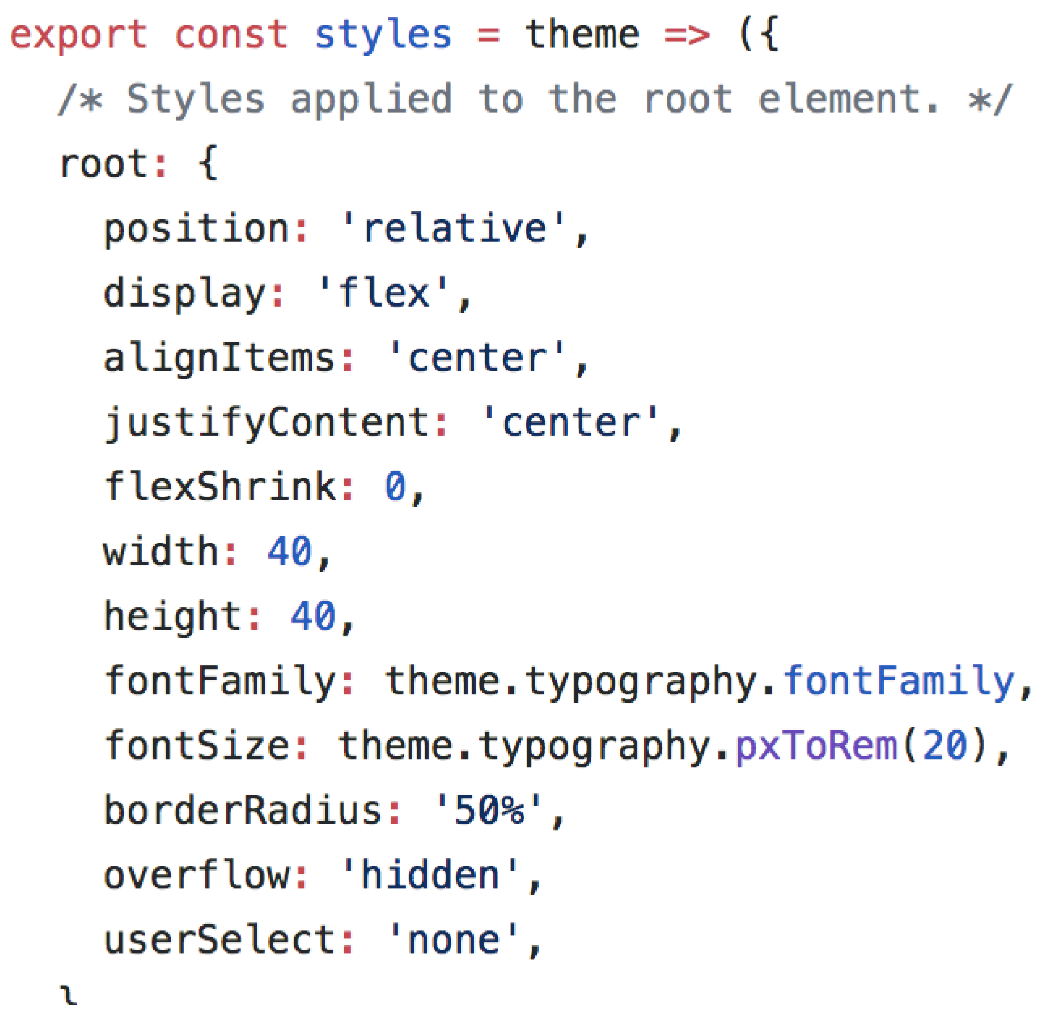
FIGURE 24. List of plugins that the UI need to write ES6 code.

- **NodeJs.** Required version is higher than **10.7** to be served to create a **Webpack dev server.** This is the most current stable version of NodeJs as well.
- **Yarn.** A faster version of **NPM** (a package manager for initialise and containing information about the project), and version number is higher than **1.12.0.** By using the **yarn add** command**,** all required modules and dependencies will be installed to project the **node_modules** folder.

3.3 Research on how other popular UI library is being implemented

Before going deeper in the implementation, an overall view of how the UI library project should be structured and what is the main coding pattern used are the main requirement. The best way is to understand how other popular and successful UI libraries are being implemented, such as: **Material-UI, React-bootstrap, React-Virtualize.**

Material-UI is a great UI library that has a high number of stars in its Github repository (around 46k stars recorded in April 2019). However, there is a huge unfavourable pattern being used in the project, **styled-component,** which will then style the component inside the JS file that contains React components, which does not fit to the Vaimo UI library requirement and perspective. Figure 25 below shows how **styled-component** structure the CSS style inside a React Component file (24). The screenshot is taken from Material UI source code, Avatar component.



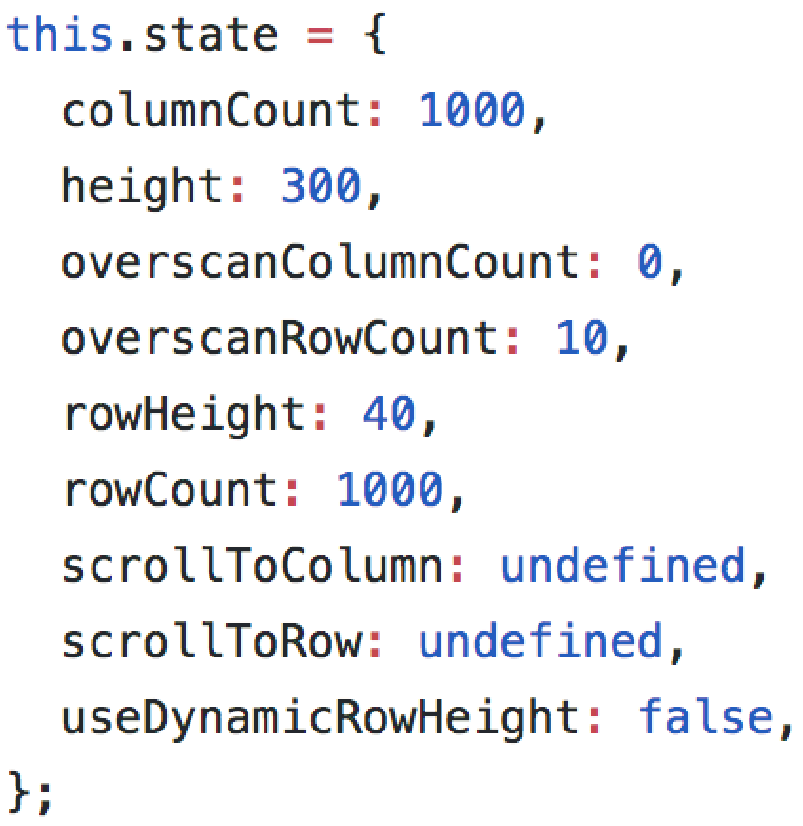
```
export const styles = theme => ({
  /* Styles applied to the root element. */
  root: {
    position: 'relative',
    display: 'flex',
    alignItems: 'center',
    justifyContent: 'center',
    flexShrink: 0,
    width: 40,
    height: 40,
    fontFamily: theme.typography.fontFamily,
    fontSize: theme.typography.pxToRem(20),
    borderRadius: '50%',
    overflow: 'hidden',
    userSelect: 'none',
  }
```

FIGURE 25. Styled component styling.

This technique has been raised and praised for its convenient, as all the style of a specific component will sit inside the component JS file itself. Though having some more other advantages, the styling inside JS component technique is not suitable for the UI library, since the code base seems to be messy when mixing separate concerns into a single file.

React-Bootstrap is a library that exploits the usage of **Bootstrap 4** in **React** components, by using a predefined class in **Bootstrap 4** and then make use of those classes in **React** components. This approach does not fit for the Vaimo UI library as the style for each theme storefront will be different most of the time, and it seems not being useful and reusable at some points, as each project will have to use a predefined class name instead of defining its own.

React-virtualize has quite the same approach as **Material-UI,** except it also controls the state of styling through its own **React component,** as shown in figure 26.



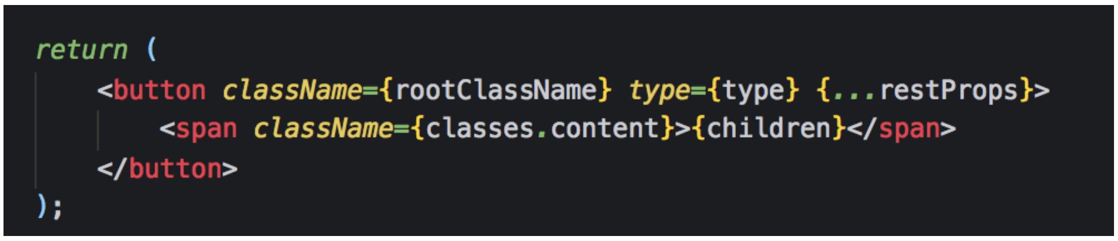
```
this.state = {
  columnCount: 1000,
  height: 300,
  overscanColumnCount: 0,
  overscanRowCount: 10,
  rowHeight: 40,
  rowCount: 1000,
  scrollToColumn: undefined,
  scrollToRow: undefined,
  useDynamicRowHeight: false,
};
```

FIGURE 26. Style is managed through application state

This would lead to a big component, as everything is controlled inside a single component. React Virtualized, Grid component (25) is an example, the file has over 1,600 lines, which contains all the logic for **Grid styling.** It could suit the need of **React-Virtualize** project, as React-Virtualize project need to style the component based on the application state, but not with the case of the Vaimo UI library, where style would just simply apply to a component like normal styling.

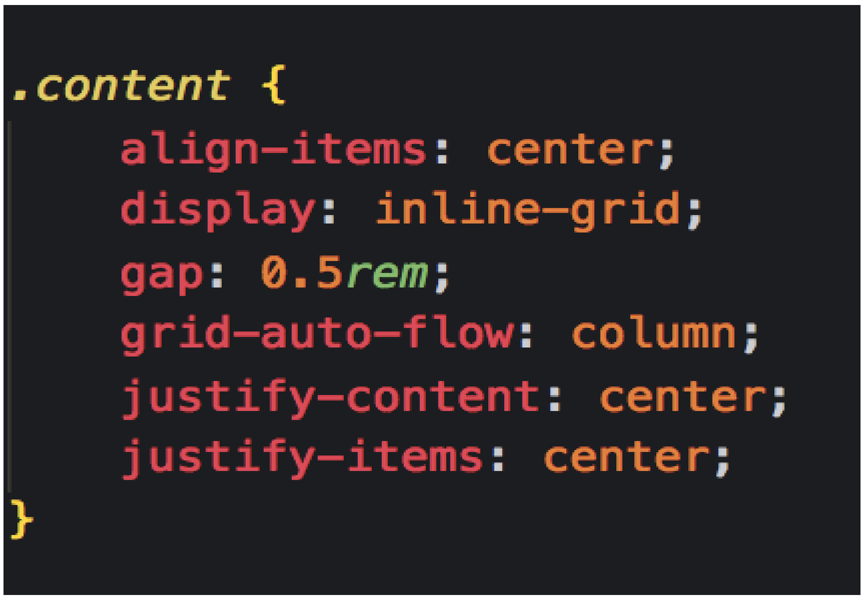
The library that the Vaimo developers chose is somewhat related to Venia-concept implementation. Being implemented by Magento team, it shares many similarities in the needs of the Vaimo UI library, such as the structure and patterns. It also has ready UI components with tests written. Each component has its own styling in a single CSS file and exploiting the use of **CSS-module,** which fits into the project case. Venia-concept system architecture is also well designed, and concerns of a web shop are separated logically. The structure of Venia theme and the UI library are close to each other, so that in the next chapter, only the final structure of the Vaimo UI library will be mentioned.

For example, the Button component has its own style, while **css-module** helps writing composable **CSS code** for **React component**. Webpack configurations for **style-loader** also has the option **localIdentName** to add a hash to component class name to make sure that there is only a single class with that name in the whole styling of the web application, thus improving class encapsulations for each component. In the component, after simply imports the CSS file, followed by adding the CSS classes to the rendered element, the styling would be applied. Figure 27 and figure *28* demonstrate the usage and implementation of component styling using **CSS modules**.



```
return (
    <button className={rootClassName} type={type} {...restProps}>
        <span className={classes.content}>{children}</span>
    </button>
);
```

FIGURE 27. Mark up of a button component



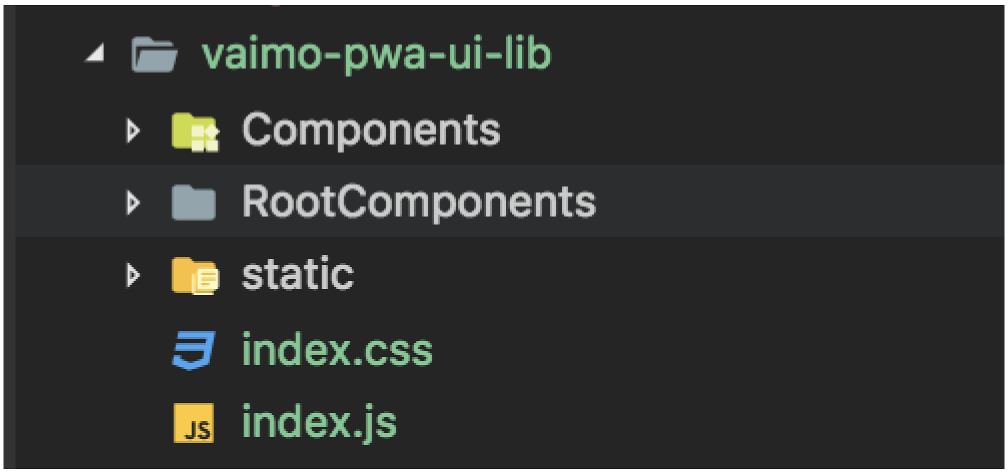
```
.content {
    align-items: center;
    display: inline-grid;
    gap: 0.5rem;
    grid-auto-flow: column;
    justify-content: center;
    justify-items: center;
}
```

FIGURE 28. Style for button content

In this case, the class **.content** will be applied and will only be applied to the **span** mark-up in figure 27 above.

3.4 Setting up the base for UI library

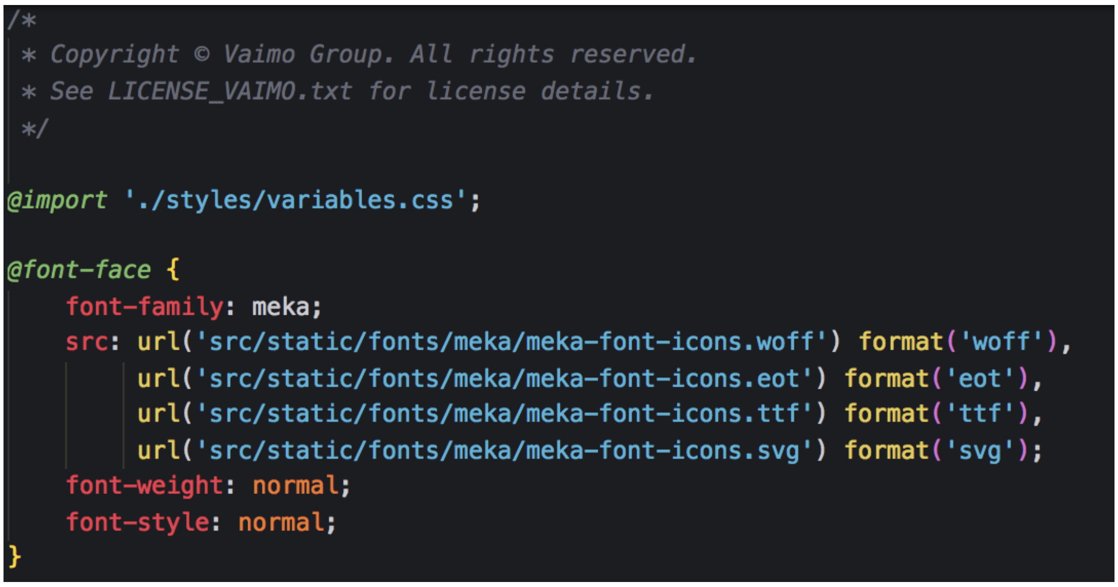
Following with the mindset above, the library should definitely have a folder containing UI components, such as Button, Navbar, a folder that has all the logic about how data should be fetched from the GraphQL query, such as Product, Category, Search, which then connects to other simple UI components, named **RootComponents**, a folder that contains website assets like templates, icons, images, and a **service-worker.js** file that will help with caching and register strategies for the Progressive Web App. An example of project structure is indicated in figure 29.



```
vaimo-pwa-ui-lib
  Components
  RootComponents
  static
  index.css
  index.js
```

FIGURE 29. The main structure of Vaimo UI library

There should be an **index.js** and an **index.css** file, which mainly served as the exporter of the library. The content of **index.js** was mentioned before, as its main function is to export **UI React components**. The **index.css** file will contain general styling for the whole website and will be the main file that imports all other dependencies, such as the font family and CSS reset rules, determined in figure 30 and figure 31.

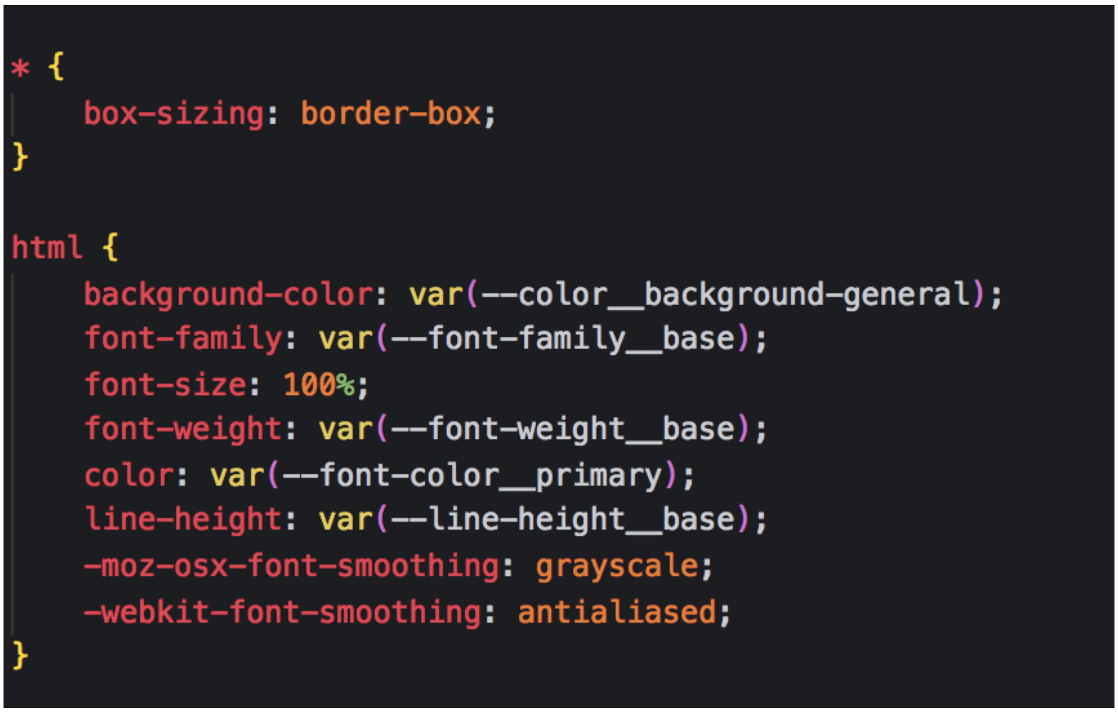


```css
/*
 * Copyright © Vaimo Group. All rights reserved.
 * See LICENSE_VAIMO.txt for license details.
 */

@import './styles/variables.css';

@font-face {
    font-family: meka;
    src: url('src/static/fonts/meka/meka-font-icons.woff') format('woff'),
         url('src/static/fonts/meka/meka-font-icons.eot') format('eot'),
         url('src/static/fonts/meka/meka-font-icons.ttf') format('ttf'),
         url('src/static/fonts/meka/meka-font-icons.svg') format('svg');
    font-weight: normal;
    font-style: normal;
}
```

FIGURE 30. Importing font family for global styling usage

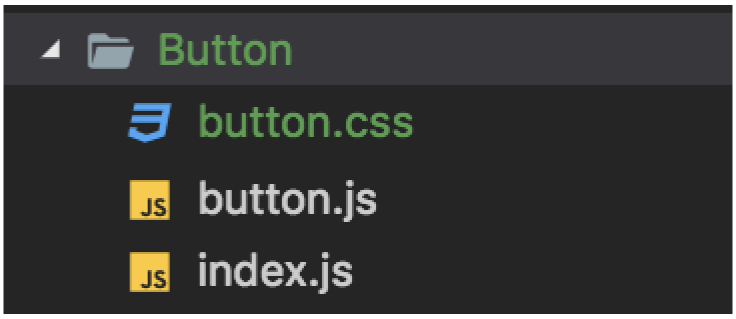


```css
* {
    box-sizing: border-box;
}

html {
    background-color: var(--color__background-general);
    font-family: var(--font-family__base);
    font-size: 100%;
    font-weight: var(--font-weight__base);
    color: var(--font-color__primary);
    line-height: var(--line-height__base);
    -moz-osx-font-smoothing: grayscale;
    -webkit-font-smoothing: antialiased;
}
```

FIGURE 31. Base rules for web application

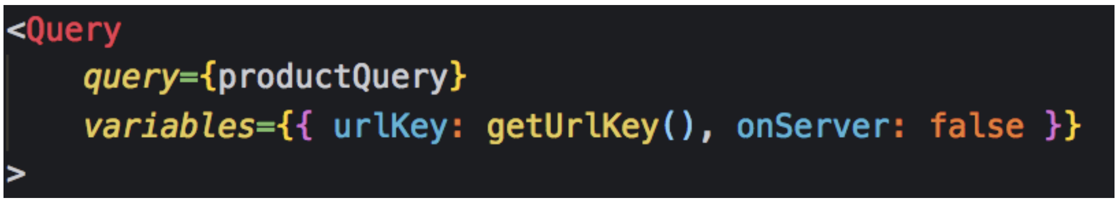
An example of an UI component should have this structure as the Button component, where all the main logic and render process stay inside a single ReactJS component file, **button.js**, with styling should be added in another **button.css** file and got exported by an **index.js** file, as in figure 32.



```
Button
    button.css
    button.js
    index.js
```

FIGURE 32. Structure of a typical UI component

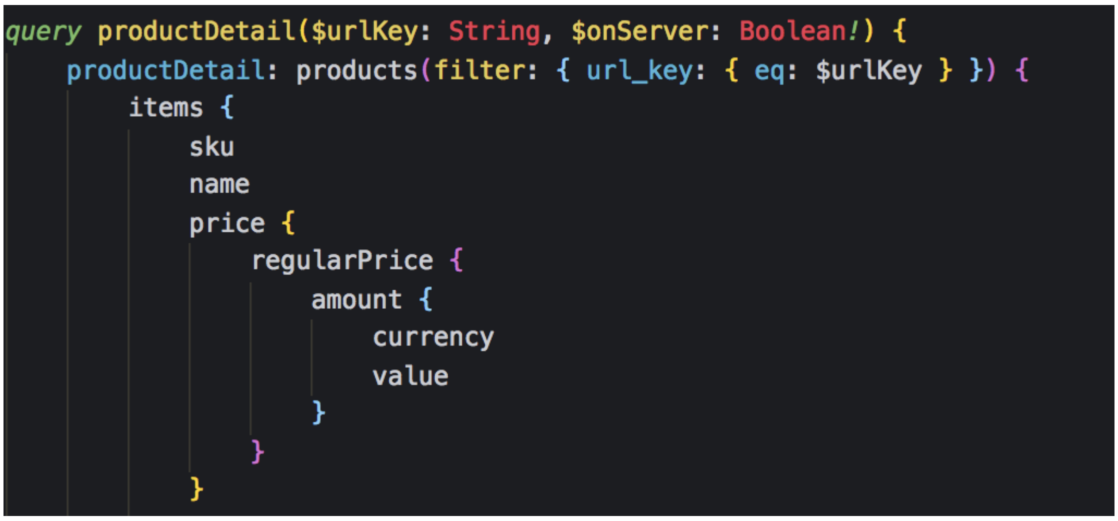
The **RootComponents** folder will consist of root components that have the logic and content needed from Backend and it will be fetched with the **GraphQL** query. An example of a root component could be the **Product** component, where **productQuery** is imported and getting used with the **Query** tag from the **react-apollo** package, demonstrated in **figure 33** and **figure 34**.



```
<Query
    query={productQuery}
    variables={{ urlKey: getUrlKey(), onServer: false }}
>
```

FIGURE 33. Query component from react-apollo to query product

A **productQuery** would ask for the data needed from Backend for the type **Product** in figure 34 shown below.



```
query productDetail($urlKey: String, $onServer: Boolean!) {
    productDetail: products(filter: { url_key: { eq: $urlKey } }) {
        items {
            sku
            name
            price {
                regularPrice {
                    amount {
                        currency
                        value
                    }
                }
            }
```

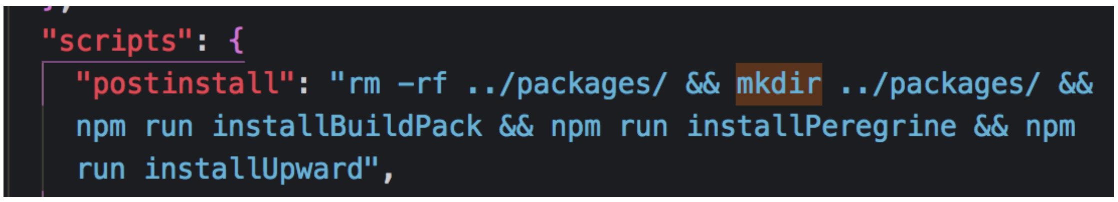
FIGURE 34. Query request structure for product

3.5 Integrating the Vaimo UI library to the Meka Project

The **Vaimo PWA UI library** could now be installed via the composer in the Meka Magento 2 project root, or any other Magento 2 project, by using the command: **composer require vaimo/vaimo-pwa-ui-lib**.

Though, there is an issue that has not been considered before, which is that the Magento 2 update on other packages, such as **Peregrine** or **UpwardJs**, could cause some conflicts in packages version in Magento 2 project, and it would be hard to maintain and control that packages version.

The solution to this is to create a new **pwa-setup** folder which will separate the concerns for all Magento 2 packages, by installing those packages into another separate **node_modules** folder rather than the **node_modules** folder inside the root directory that contain the project level dependencies. The command to do so is demonstrated in figure 35



```
"scripts": {
  "postinstall": "rm -rf ../packages/ && mkdir ../packages/ &&
  npm run installBuildPack && npm run installPeregrine && npm
  run installUpward",
```

FIGURE 35. Install PWA buildpack, Peregrine and Upward in a separated folder

This file keeps all the Magento's pwa-studio package dependencies up-to-date and all under one folder, thus helping to manage the pwa-studio package version much easier.
After successfully setting files up, the **Vaimo UI library** could then be imported and make use of.

3.6 Reconsidering the use case of the Vaimo PWA UI library

Though the **Vaimo PWA UI library** could now be used with ease, there are some issues with the implementation of the library. The main purpose of creating the UI library is to compose general and simple UI components that will be reused on Magento 2 project, which means that the library should not contain specific logic for any single component. Looking on the **SearchBar** component, which has **auto complete** functionality, the component was not made to be general. The component is still depending on data fetched from Backend with the GraphQL query. Whenever a user type on the search bar field, the component takes data from server side and suggests the expected result. Other examples of such components are: **Navigation**, **Header**, **Sign in.**

Thus, there came an idea of creating another library that is responsible for such cases, the **Vaimo Core Library**. The **Vaimo UI library** will now contain only generic components that do not depend on data or anything else, such as **Button, Loading Indicator, Checkbox, Radio,** while **Vaimo Core library** will take care of components that have data involved in. The process of creating the **Vaimo Core library** would be pretty much the same as the **Vaimo UI library,** except that it needs other containers or folders containing components that are responsible for data and state management.

The **Vaimo Core library** will need these folders: **reducers, actions,** and **store** just like any other Redux React application. In addition to that, it also needs **drivers** folder that contains an adapter to connect to **GraphQL** and **Apollo-client, queries** folder that contains all the queries, such as **getProduct.graphql, getCategory.graphql,** and the **RootComponents** that contains all other core components that have data logic revolved.

4 RESULT

The 2 libraries were created for Vaimo group to support new web technologies such as the Progressive Web App, ReactJS and GraphQL, the Vaimo UI library and the Vaimo Core library. The Vaimo UI library will consist of generic UI components with minimal styling, while the Vaimo Core library will take the responsibility for handling stateful components, query data, connecting to Upward Server and being able to get the data response from the Magento server. Both libraries have some ready components to be imported by any other projects, and those components have gone through unit testing and documented for usage. The Meka project is now able to exploit these two libraries.

5 CONCLUSION

In conclusion, the initial plan has only one change compared to the actual result. There are two libraries created instead of just a single UI library which hold all the logic and abstraction, which are necessary and handy for further development in the future. The expected result matches to the description and plan of the thesis, so that the libraries can be used in a real web store application, and the implementation are built on top of modern web stack. The library also has a stable version that has been tagged and released.

By getting the setup of libraries finished, Vaimo Finland has already imported components from the libraries and exploited them in their projects, such as Meka. The Meka development process has been utterly improved due to the standard that those libraries give, such as the Webpack development server and general components. These libraries are also expected to be used across Vaimo group, as the project is a part of Vaimo Product team, leading to exponential benefits in terms of a long way run. Developers are now open up to learn modern frameworks and libraries stack, which also help them to stay up-to-date with the state of the art in Information Technology.

Though, there are some spaces for improvement for libraries. For example, **multi option filter** that helps filtering products by product attributes, **translations**, a **Product Carousel** component are still under development. Those missing components would be implemented once other projects need that specific feature as currently, Meka project is quite small, simple, and lacks some of default Magento 2 features.

On the other hand, another important feature which is currently implemented, is component testing. This is significantly crucial to make a library stable and easier to spot any arising error. The initial plan was to use Jest and Storybook, as these 2 libraries are popular, stable, and are backed by technical giants.

REFERENCES

1. Adobe. Magento 2, Magento 2 service isolation design. Date of retrieval 24.09.2018, at https://github.com/magento/architecture/blob/master/design-documents/service-isolation/magento-services.png
2. Z. Michael Luo, 17/10/2017, MVC vs. MVVM: How a Website Communicates With Its Data Models. Date of retrieval: 15.05.2019, at: https://hackernoon.com/mvc-vs-mvvm-how-a-website-communicates-with-its-data-models-18553877bf7d
3. Magento 1 vs Magento 2 template structure. Date of retrieval: 12.05.2019, at: https://belvg.com/blog/magento-1-vs-magento-2-template-structure-certification-exam.html
4. KnockoutJS, documentation, at: https://knockoutjs.com/documentation/introduction.html
5. Fielding, Roy Thomas, 2000. Chapter 5: Representational State Transfer (REST). Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine, at: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)
6. Bradley Nice, 2017. A Quick Intro Into Progressive Web Apps, at: https://medium.com/level-up-web/a-quick-intro-into-progressive-web-app-7c9de2391a2d)
7. Status of Progressive Web App, ”Is service worker ready?”. Date of retrieval, 25.05.2019, at: https://jakearchibald.github.io/isserviceworkerready/
8. Matt Guant, updated on 01/05/2019, Service Workers: an Introduction. Date of retrieval 15.05.2019, at: https://developers.google.com/web/fundamentals/primers/service-workers/)
9. Mozilla Developer Network, Web App Manifest. Date of retrieval, 23-05-2019, at: https://developer.mozilla.org/en-US/docs/Web/Manifest
10. Addy Osmani, 2017. A Pinterest Progressive Web App Performance Case Study. Date of retrieval: 09.04.2019, at: https://medium.com/dev-

channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154

11. Alberto Gimeno, Mar 27, 2018. The deepest reason why modern JavaScript frameworks exist. Date of retrieval: 24.05.2019, at: https://medium.com/dailyjs/the-deepest-reason-why-modern-javascript-frameworks-exist-933b86ebc445
12. ReactJs, React Virtual DOM. Date of retrieval 28.05.2019, at: https://reactjs.org/docs/react-dom.html
13. Vrann Tulika, 18.10.2017, GraphQL in Magento 2. Date of retrieval 05.05.2019, at: https://www.slideshare.net/vrann/graphql-in-magento-2-presentation-from-mm-nyc-2017
14. GraphQL. Date of retrieval 20.05.2019 at https://graphql.org/
15. Adobe. Magento 2. Current state on Magento 2 implementation, design document. Date of retrieval 22/01/2019, at: https://github.com/magento/architecture/blob/master/design-documents/service-isolation/current-state.png)
16. Adobe. Magento 2. Desired state on its system implementation, design document. Date of retrieval: 22.01.2019, at: https://github.com/magento/architecture/blob/master/design-documents/service-isolation/desired-state.png)
17. Adobe. Magento 2, Application shell, Magento 2 research documentation. Date of retrieval: 13.05.2019, At: https://magento-research.github.io/pwa-studio/technologies/basic-concepts/app-shell/
18. Matt Carroll, 14.03.2018. The flutter Redux problem. Date of retrieval: 02.04.2019 at: https://medium.com/fluttery/the-flutter-redux-problem-fa9d59ec97b8
19. Google Workbox Webpack Plugins. Updated on 01.05.2019, at: https://developers.google.com/web/tools/workbox/modules/workbox-webpack-plugin
20. Adobe, Magento 2. Where UPWARD belongs in the PWA architecture, magento research. Date of retrieval: 14.05.2019, at: https://magento-research.github.io/pwa-studio/technologies/upward/)

21. The twelve factor app, 2017, various contributors. Date of retrieval: 15/05/2019, at: https://12factor.net/backing-services
22. Upward spec, Rationale. Date of retrieval: 15.05.2019, at: https://github.com/magento-research/pwa-studio/blob/develop/packages/upward-spec/RATIONALE.md)
23. UpwardJS, upward spec, upward.yml. Date of retrieval: 05.05.2019, at: https://github.com/magento-research/pwa-studio/blob/master/packages/venia-concept/venia-upward.yml
24. Aron Stanic, Inchoo interview, 15.01.2019, Date of retrieved: 04.05.2019, at https://inchoo.net/magento-2/magento-pwa-studio-contributor-adrian-bece/
25. Material UI, Avatar Component. Date of retrieval: 02.05.2019, at: https://github.com/mui-org/material-ui/blob/next/packages/material-ui/src/Avatar/Avatar.js)
26. React Virtualized, grid component. Date of retrieval: 02.05.2019, at: https://github.com/bvaughn/reactvirtualized/blob/master/source/Grid/Grid.js