UI Frameworks in Rails

Finding the right implementation

Artur Veselovski

Bachelor's thesis
May 2021
Technology, communication and transport
Degree Programme in Information technology
Media technology

jamk.fi

Description

Author(s) Veselovski, Artur	Type of publication Bachelor’s thesis	Date December 2020
		Language of publication: English
	Number of pages 42	Permission for web publication: x

Title of publication
Modern UI frameworks in Rails

Degree programme
Information and Communication Technology

Supervisor(s)
Lappalainen-Kajan, Tarja; Salmikangas, Esa

Assigned by
Qvantel Oy

Abstract

The thesis covers user interface implementation within Ruby on Rails framework with MPA (Multi Page Application) and SPA (Single Page Application) patterns. The aim was to find out which pattern was best suited for certain use cases.

The goal was to build two identical prototypes using these patterns and explore pros and cons of each, evaluating in the process, which is more developer friendly, scalable and maintainable. In addition, determining which offers more convenient ways to implement advanced UI (User Interface) features.

The theoretical framework explored Ruby on Rails as a technology with its MVC (Model View Controller) model, and SPA and MPA concepts. React is explained as a library. Webpack and Webpacker roles in Ruby on Rails UI development are covered as well.

The setup of both prototypes, as well as possible encountered problems and observations are covered in the implementation.

As the result, it was determined that SPA was more developer friendly, and implementation of advanced features was easier. It offers good support and maintainability. MPA is sufficient for prototyping and simple web-applications, where UI is not a priority. Developer experience and maintainability would suffer with MPA if the application was to grow. No significant performance differences were observed between prototypes.

Keywords/tags (subjects)

Ruby on Rails, JavaScript, React, UI, MVC, SPA, MPA

Miscellaneous (Confidential information) jamk.fi

Kuvailulehti

Tekijä(t) Veselovski, Artur	Julkaisun laji Opinnäytetyö, AMK	Päivämäärä Toukokuu 2021
		Julkaisun kieli Englanti
	Sivumäärä 42	Verkkojulkaisulupa myön-netty: x

Työn nimi
Modern UI frameworks in Rails

Tutkinto-ohjelma
Tieto- ja viestintätekniikka

Työn ohjaaja(t)
Lappalainen-Kajan, Tarja; Salmikangas, Esa

Toimeksiantaja(t)
Qvantel Oy

Tiivistelmä

Opinnäytetyössä käsiteltiin MPA (Multi Page Application) ja SPA (Single Page Application) malleja käyttöliittymän toteutuksessa Ruby on Rails framework:in puitteissa. Tavoitteena oli selvittää, kumpi malli sopii mahdollisesti parhaiten mihinkin käyttötarkoitukseen.

Tavoitteena oli rakentaa kaksi identtistä prototyyppiä käyttäen SPA ja MPA malleja, tutkia näiden hyviä ja huonoja puolia, sekä arvioida, kumpi on kehittäjäystävällisempi, skaalautuvampi, ja kummalla on helpommin ylläpidettävä koodi. Sen lisäksi tavoitteena oli arvioida, kummalla on helpommin toteutettavissa vaativat UI (User Interface) ominaisuudet.

Teoreettisessa viitekehyksessä käydään läpi Ruby on Rails teknologiana ja sen MVC (Model View Controller) mallia, sekä SPA ja MPA käsitteinä. Tämän lisäksi käydään läpi React kirjastona, sekä Webpack ja Webpacker teknologioiden roolit Ruby on Rails käyttöliittymän kehityksessä.

Toteutuksessa käytiin läpi molempien prototyyppien kehityksen perusasetukset, mahdolliset ongelmatilanteet ja havainnot.

Lopputuloksena selvisi, että SPA oli kehittäjäystävällisempi ja vaativien ominaisuuksien rakentaminen oli helpompaa. Se tarjoaa hyvää tukea ja ylläpidettävyyttä. MPA sopii hyvin prototyyppien ja yksinkertaisten sovellusten tekemiseen, missä käyttöliittymä on toissijainen, mutta kehittäjäkokemus ja ylläpidettävyys kärsii, kun sovelluksen skaala kasvaa. Suorituskyvyssä ei ole todettu huomattavia eroja.

Avainsanat (asiasanat)

Ruby on Rails, JavaScript, React, UI, MVC, SPA, MPA

Muut tiedot (salassa pidettävät liitteet)

Contents

Figures

Tables

Abbreviations

AJAX	Asynchronous JavaScript and XML
CRUD	Create Read Update Delete
Db	Database
DOM	Document Object Model
JS	JavaScript
MPA	Multi-page Application
MVC	Model View Controller
MVP	Minimal Viable Product
SPA	Single Page Application
UI	User interface
URL	Uniform Resource Locator

1 Introduction

1.1 Qvantel Oy

The topic of this thesis was assigned by Qvantel Oy. Qvantel is one of the fastest-growing tech companies in Finland. With presence in 23 countries, Qvantel provides cloud-based BSS (Business Support Systems) solutions for Communication Service Providers across Europe, Northern Africa, APAC and the Americas. Some of Qvantel's clientele includes DNA, Very Mobile and MasMovil, amongst many other names across the world.

Thesis was done independently, with Qvantel providing its in-house UI library (weasel-styles) and UI prototype. Assignment demos are based on the provided prototype.

1.2 Problem

Qvantel has a lot of products in its portfolio, with teams across the globe working within different organizations. Organization and teams within have certain freedom of choice when it comes to deciding project's tech-stack. That leads to many projects using different approaches when it comes to "how things are built". This thesis focuses on one area, Ruby on Rails applications and its UI (User Interface) implementation.

Rails is widely used in Qvantel's projects and UI implementation is up to developers to decide on. This is good, because there is no "one size fits all" -solutions. Depending on the use-case, building an SPA (Single Page Application) might be the way to go, while in others it was argued to be "overkill", creating unnecessary overhead and complexity for very little benefit, if at all. Perhaps building the same application with traditional MPA (Multi Page Application) architecture, using less JS (JavaScript) is the way to go. With that in mind, applying the right technique for implementing UI in Ruby on Rails application, suitable for a given use-case is in question.

Which is more developer friendly, accessible and maintainable? Is SPA the de facto choice for implementing UI with Ruby on Rails applications or are the tools that Rails provides, coupled with “lightweight” and possibly “frameworkless” JS enough for an average Qvantel applications frontend?

1.3 Goals

Main goal was exploring building frontend applications with Ruby on Rails framework, using SPA or MPA approaches, with some hybridization. Finding cons and pros in terms of developer experience. The groundwork, complexity of architecture and required third party packages for consistently working MVP and final product. Ease of advanced UI manipulation implementation and backend-frontend communication, and finally maintainable and readable codebase.

Rails is a powerful tool for building web applications. Modern applications require modern “rich” user interfaces with advanced UI manipulation, which cannot be achieved with just Ruby on Rails and basic out of the box Bootstrap.js provided scripts.

Rails can be versatile in how UI is actually implemented. For simpler applications, some jQuery (a JavaScript library) or vanilla JS (clean JavaScript without additional libraries or frameworks) can be enough, but for bigger applications building an SPA is the standard approach in modern web application development. What about MPA, but with the help of a modern SPA centric JS framework or library?

There are many JS frameworks to choose from. For simplicity and scope of this thesis, most popular framework, React JS, was tested for SPA. React technically is a library and not a framework, but its ecosystem has enough good tools to achieve the same goals, as with a single framework. MPA had vanilla JS and jQuery with Bootstrap.js provided scripts for simple interactions. React was used in places, where “frameworkless” approach was deemed too unproductive in a real-world scenario.

2 Modern web-application development with Rails

2.1 Ruby on Rails

Ruby on Rails is a popular open source software, a server-side framework built with Ruby programming language. It follows MVC (Model-View-Controller) pattern and has everything that is needed to create database-backed web-applications. (Welcome to Rails 2021.)

Rails is shipped with Active Record, Active Model, Action Pack and Action View core frameworks, which can be used independently without Rails. The package also contains Action Cable for WebSocket live updates; Action Mailer and Action Mailbox for sending and receiving emails; Action Support, that offers Ruby and the framework helpful utility classes and extensions; Active Job for queuing backend services; Action Text for rich text content and Active Storage for cloud and local files handling. (Welcome to Rails 2021.)

In short, the model (Active Record) of the MVC is for business logic and database communication. Controller (Action Controller) handles user input, routes to the model and renders views (Action View) as HTML templates or JSON (JavaScript Object Notation). (Rails has everything you need n.d.)

2.2 MVC (Model-View-Controller)

MVC is a pattern of a software application architecture, that separates an application into 3 components, models, views and controllers. This pattern allows for separation of concerns (SoC) to be achieved in an application. (Smith 2020.) “Separation of concerns (SoC) is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern" (Separation of concerns 2021). MVC pattern was traditionally used with desktop graphical user interfaces (GUIs) and later has become popular among web-application design (Model-view-controller 2021).

Model is responsible for the application state and everything related to state updating & persistence. As such, Model is the component communicating with databases. Controller is what accepts user input, like requests to see or update some data. Views are handling the presentation of the data. Interactions of these components are represented in figure 1. Typical web application’s MVC processes and interactions are presented in figure 2.

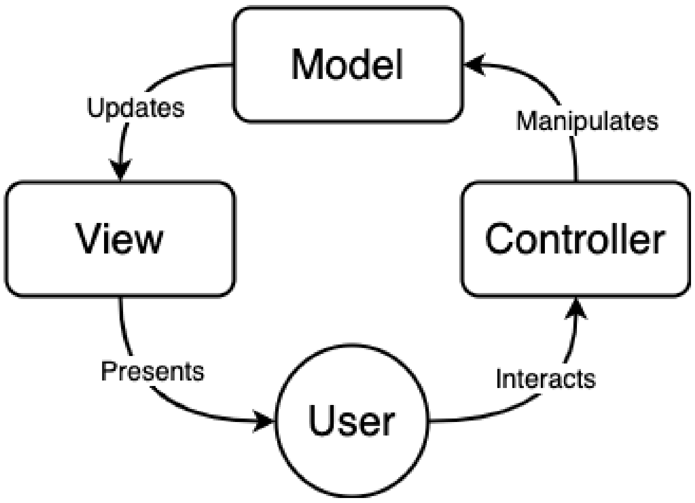


Figure 1 MVC components interaction

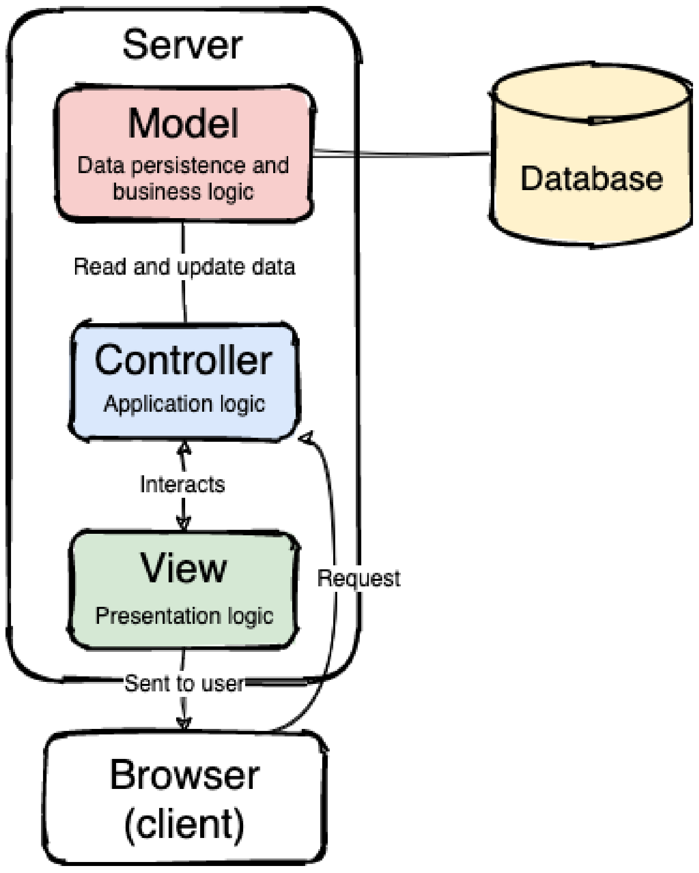


Figure 2. Web-application MVC

Separation brings quite a few benefits, some of which are scalability, maintenance and reusability. Implementation of data persistence and database technology (Db interaction logic on Model side and database software / hardware) can be changed without affecting other components, bringing scalability. Components have low dependency on each other, hence making changes and fixing bugs in one having no effect on the other. That increases ease of maintenance, and finally, Model can be used by several Views, bringing reusability. (Goodrich 2017.)

2.3 MPA (Multi Page Application)

Multi Page Application architecture is the traditional way of building web applications. Pages are reloaded on user input. Whenever users request new data via navigation, or submit data via forms, a new page is requested from the server. This works well with dynamic pages, such as blogs, and simple web applications. When applications are more complex with rich UI, pages may be loading with a lot of data. Generating those pages on the server and sending to the client to be rendered would take time, impacting UX (User Experience) negatively. (Tarnovskiy 2015.)

MPA improved with introduction of AJAX technology, which allowed to do requests in the background, refreshing parts of a page without reloading the whole page. This improved user experience, but at the cost of added complexity. (Tarnovskiy 2015.)

2.4 SPA (Single Page Application)

Single Page Application differentiates from a traditional MPA by serving one HTML file with a JS file containing the whole JavaScript application. It is essentially an evolution of MPA with AJAX. The HTML file served from the server act as a shell, into which the application is injected on the client-side in the browser when the JS file is run. (Tarnovskiy 2015.)

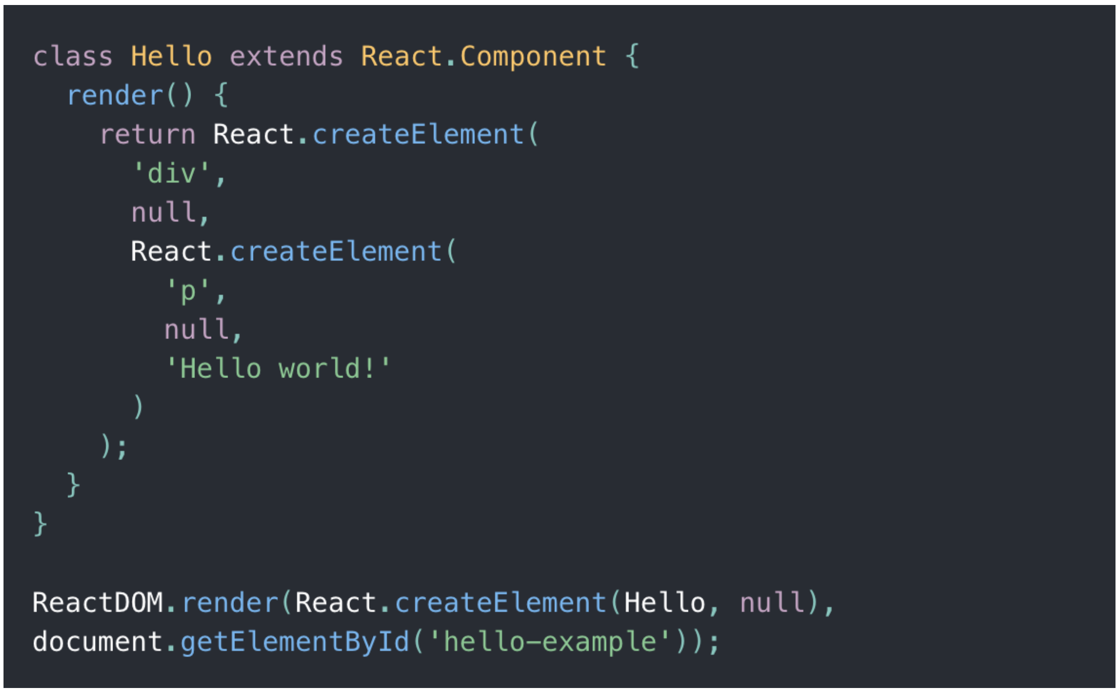
Commonly acknowledged advantages of SPA over MPA are faster load times (after initial load), better user experience and decoupling of frontend and backend development. Disadvantages include slower initial load due to big amounts of code being loaded at once, instead of chunks needed for a given page, as is with traditional MPA. (Tarnovskiy 2015.)

2.5 React

React is an open-source JavaScript library for creating user interfaces (UI). It was created by Facebook and was made open source in 2013. Being a library, rather than a framework, it does not offer an “all in one” solution for building applications whole. Instead React focuses on UI aspect with its component-based approach, managing state and rendering that state to the DOM (Document Object Model). That means requiring additional libraries to handle things that are traditionally handled by default within MPA, such as application routing. (React (JavaScript library) 2021.)

While React is specifically built for SPA architecture, it can be used within MPA as isolated components on a page, instead of encompassing the whole application. Being a library without above-mentioned “all in one” solutions can further help in that department. React then can be used as intended for displaying UI and managing its local state, while MPA would handle the rest.

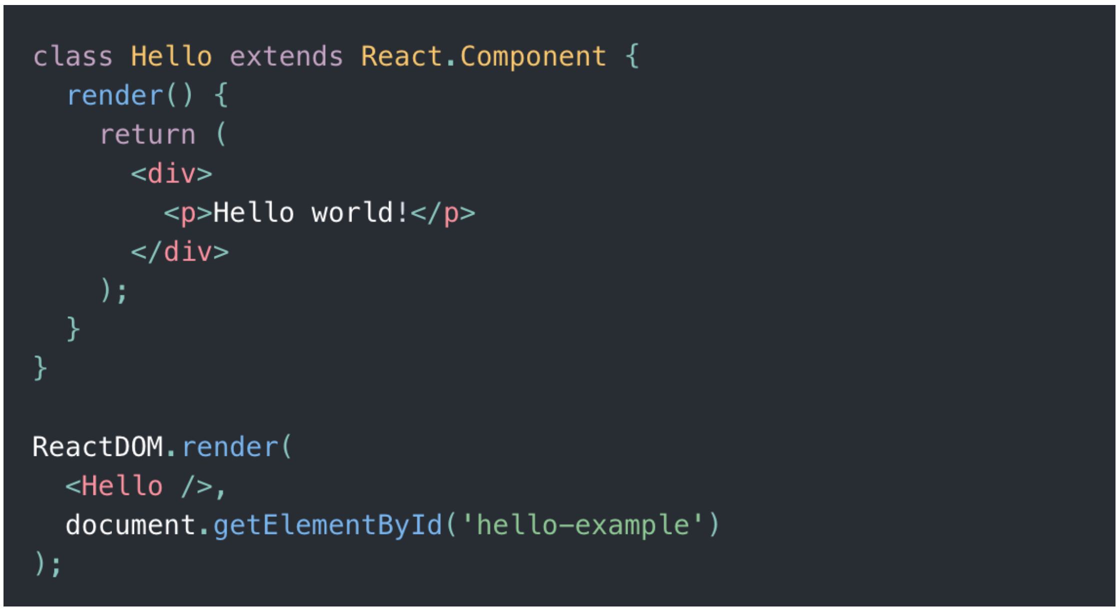
Most notable features of React are virtual DOM and JSX (JavaScript XML) syntax. JSX allows for easy to read and write HTML-like syntax for components. While not necessary to create React applications, JSX helps with Reacts nested component structure. Figures 3 and 4 demonstrate what JSX does underneath. Virtual DOM creates a virtual representation of DOM structure, which it uses to compare to actual DOM and calculate what needs to be updated on a component level, rather than re-render entire pages on every change. Rendering pages being a very expensive operation, this provides significant performance boosts. (React (JavaScript library) 2021.)



```
class Hello extends React.Component {
  render() {
    return React.createElement(
      'div',
      null,
      React.createElement(
        'p',
        null,
        'Hello world!'
      )
    );
  }
}

ReactDOM.render(React.createElement(Hello, null),
document.getElementById('hello-example'));
```

Figure 3. A simple “Hello” component without JSX.



```
class Hello extends React.Component {
  render() {
    return (
      <div>
        <p>Hello world!</p>
      </div>
    );
  }
}

ReactDOM.render(
  <Hello />,
  document.getElementById('hello-example')
);
```

Figure 4. A simple “Hello” component with JSX.

2.6 Webpack and Webpacker

Webpack is a pre-processor and static module bundler for client-side assets (JS, CSS, images, fonts). A tool that takes different pieces of JavaScript code, along with their dependencies, and combines (or bundles) those into one or many files, called “output”. That generated output is what is served with HTML from the server. Bundled code is what is actually run in the browser. (Gerchev 2020.)

Webpack can have “Loaders” and “Plugins” as additional configurations to achieve required goals. Loaders are needed to process non-JS code. By default, only JS and JSON formats are understood by Webpack. These loaders allow the bundle to contain CSS imports, images, etc. Loader, such as “sass-loader” for example, loads SASS/SCSS files and processes them into CSS, which is understandable for the browser. Tasks, that Loaders don’t perform, such as bundle minimization, is done by plugins. (Gerchev 2020.)

Webpacker integrates Webpack with Rails applications. A wrapper, that provides standard Webpack configurations with reasonable defaults for Rails applications. (Webpacker n.d.)

3 Implementation

3.1 Demos

The implementation consisted of 2 simplified identical applications or demos, that were based on one of Qvantel’s UI prototypes. Prototype uses in-house CSS-framework, which is based off Bootstrap 4. Data was mocked to accommodate the UI prototype. The goal of the demos was to compare how SPA fared compared to a traditional MPA approach, by building working demos with standard UI and interactivity that is expected in Qvantel's or any other modern web applications. Interactivity, that is only possible with extensive JavaScript use.

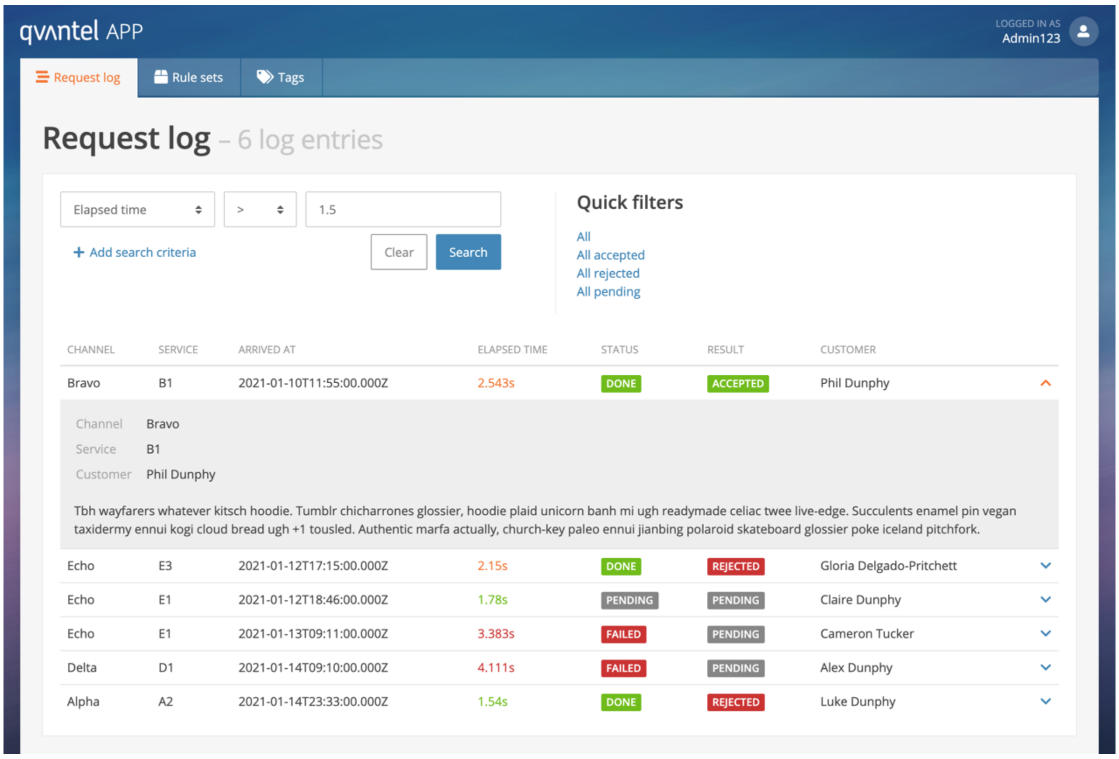
First, a traditional "old school" MPA that relies on Bootstrap.js, jQuery and Popper.js libraries for basic interactions and custom JS scripts to implement the UI prototype's functionalities. This was a reference application, for comparison with SPA. Goal was to find out how much of the UI could be built with Bootstrap provided-, and simple custom scripts, before it would become unproductive not to consider SPA or introduce a "hybrid" solution. As such, not all UI functions were implemented fully. Hybrid solution in this context meant loading React.js components inside a certain container on a page to handle specific cases, instead of the whole page as would be with SPA.

Second demo was built as SPA, using modern framework React.js from the ground up. It generates UI on one page served from the server (Rails), interacting with the server via AJAX calls. Demo utilized widely used JS modules (also called "packages"), that could be considered industry standards, with quite a few prominent companies backing and using them. This was the fully realized, 100% functioning application.

Emphasis was on two-way communication (or data flow) between Rails and frontend (UI), developer experience (happy developer is a productive developer) and maintainability of code with both approaches. Since Rails 6, Webpack, through Webpacker (Rails wrapper for Webpack) is the default way for implementing JavaScript, therefore both apps used Webpacker to setup and run JS and process SCSS (a pre-processor for CSS). Basic setup, which includes database initialization and seeding, "Devise" gem (ruby module) setup for authentication and other Rails related settings, were identical with both and as such is not covered in detail.

3.2 UI prototype

Besides the login screen, demos have 3 separate independent pages or views, with independent data. "Request log" (figure 5) has read-only data, simulating data coming from a third-party service, with complex and extensive search and filtering feature. “Rule sets” (figures 6 and 7) has a basic structure of “A contains many B, which contain many C”. Here main page lists categories of rule sets, which lists the rule sets with CRUD (Create Read Update Delete) functionality and simple client-side JS only filtering. Editing a rule set opens a rule set sub-page, which in turn contains rules and the rule set editing form. Rules also have CRUD functionality. “Tags” page represents the most basic tabled data display and management view, with CRUD, sorting, filtering and search. These types of views are aplenty in Qvantel’s apps, as well as other “management tool” types of applications (figure 8).



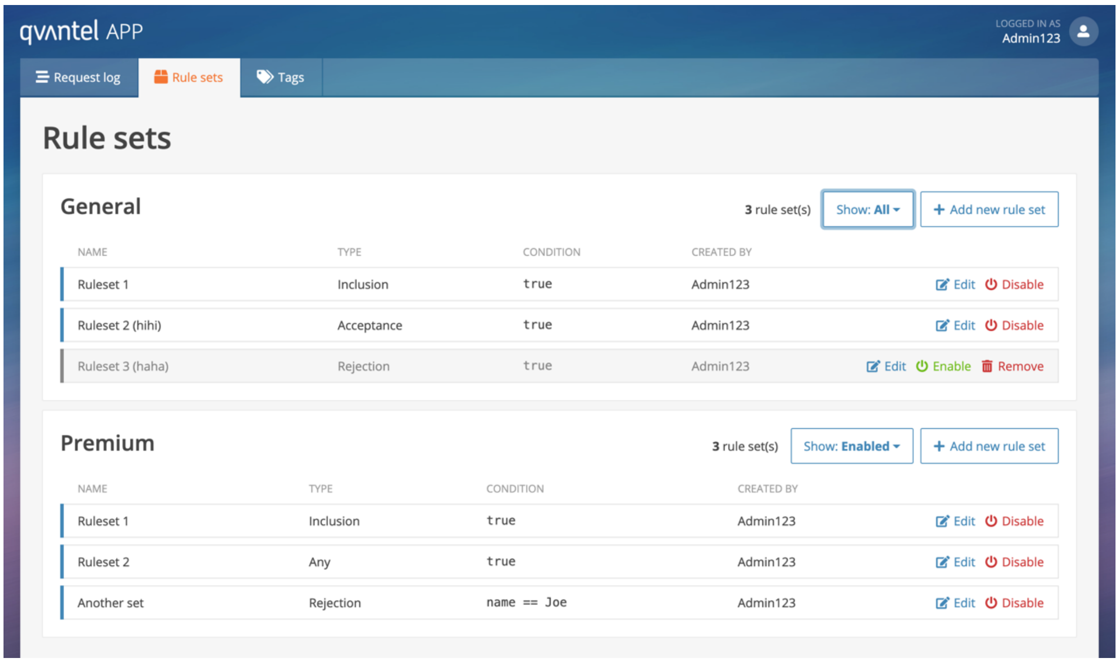
Request log – 6 log entries

CHANNEL	SERVICE	ARRIVED AT	ELAPSED TIME	STATUS	RESULT	CUSTOMER
Bravo	B1	2021-01-10T11:55:00.000Z	2.543s	DONE	ACCEPTED	Phil Dunphy
Echo	E3	2021-01-12T17:15:00.000Z	2.15s	DONE	REJECTED	Gloria Delgado-Pritchett
Echo	E1	2021-01-12T18:46:00.000Z	1.78s	PENDING	PENDING	Claire Dunphy
Echo	E1	2021-01-13T09:11:00.000Z	3.383s	FAILED	PENDING	Cameron Tucker
Delta	D1	2021-01-14T09:10:00.000Z	4.111s	FAILED	PENDING	Alex Dunphy
Alpha	A2	2021-01-14T23:33:00.000Z	1.54s	DONE	REJECTED	Luke Dunphy

Channel Bravo
Service B1
Customer Phil Dunphy

Tbh wayfarers whatever kitsch hoodie. Tumblr chicharrones glossier, hoodie plaid unicorn banh mi ugh readymade celiac twee live-edge. Succulents enamel pin vegan taxidermy ennui kogi cloud bread ugh +1 tousled. Authentic marfa actually, church-key paleo ennui jianbing polaroid skateboard glossier poke iceland pitchfork.

Figure 5. Request log -page


qvantel APP

LOGGED IN AS Admin123

Rule sets

General

3 rule set(s) Show: All + Add new rule set

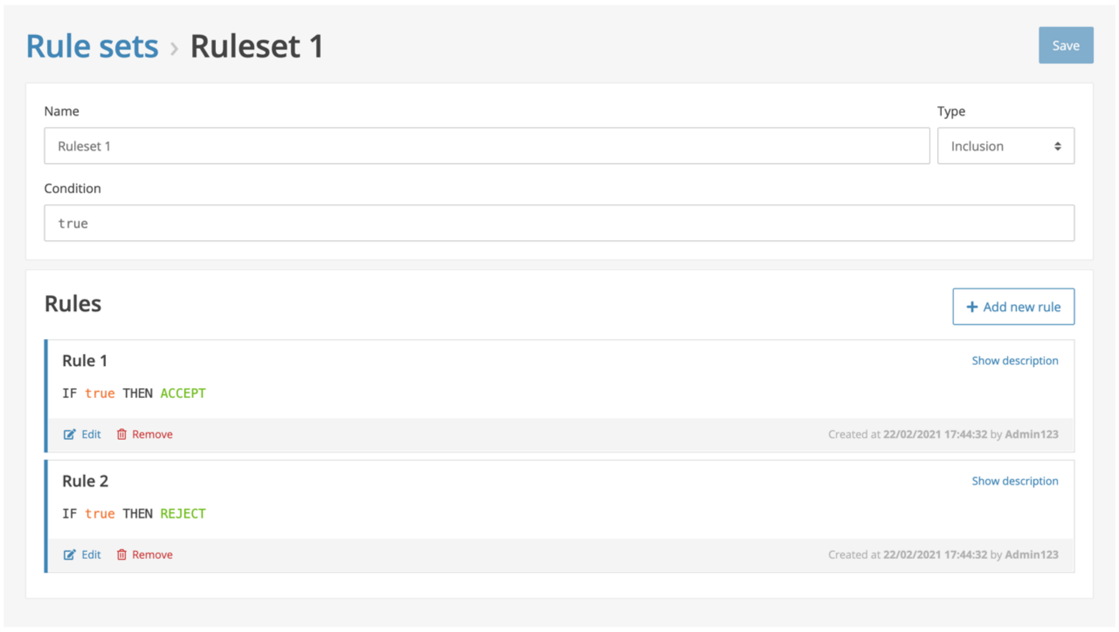
NAME	TYPE	CONDITION	CREATED BY	
Ruleset 1	Inclusion	`true`	Admin123	Edit Disable
Ruleset 2 (hihi)	Acceptance	`true`	Admin123	Edit Disable
Ruleset 3 (haha)	Rejection	`true`	Admin123	Edit Enable Remove

Premium

3 rule set(s) Show: Enabled + Add new rule set

NAME	TYPE	CONDITION	CREATED BY	
Ruleset 1	Inclusion	`true`	Admin123	Edit Disable
Ruleset 2	Any	`true`	Admin123	Edit Disable
Another set	Rejection	`name == Joe`	Admin123	Edit Disable

Figure 6. Rule sets -page


Rule sets › Ruleset 1

Name: Ruleset 1

Type: Inclusion

Condition: `true`

Rules

Rule 1

```
IF true THEN ACCEPT
```

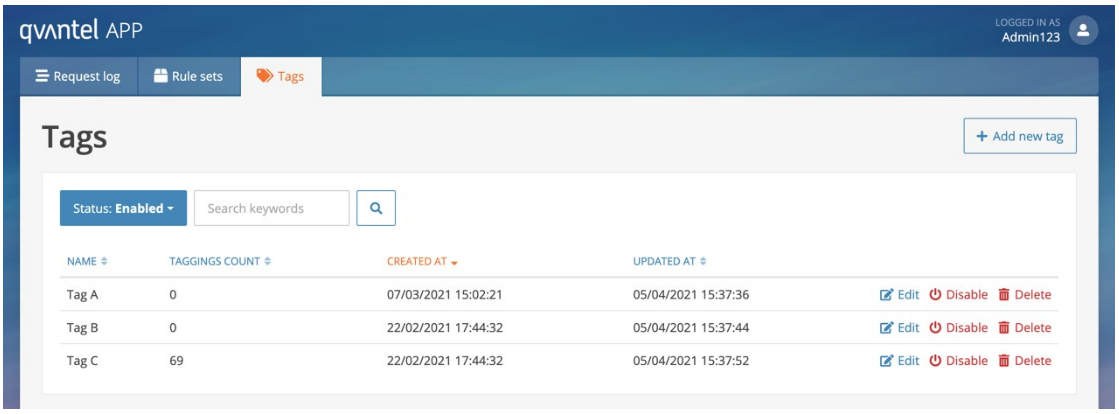
Created at 22/02/2021 17:44:32 by Admin123

Rule 2

```
IF true THEN REJECT
```

Created at 22/02/2021 17:44:32 by Admin123

Figure 7. Rule set sub-page

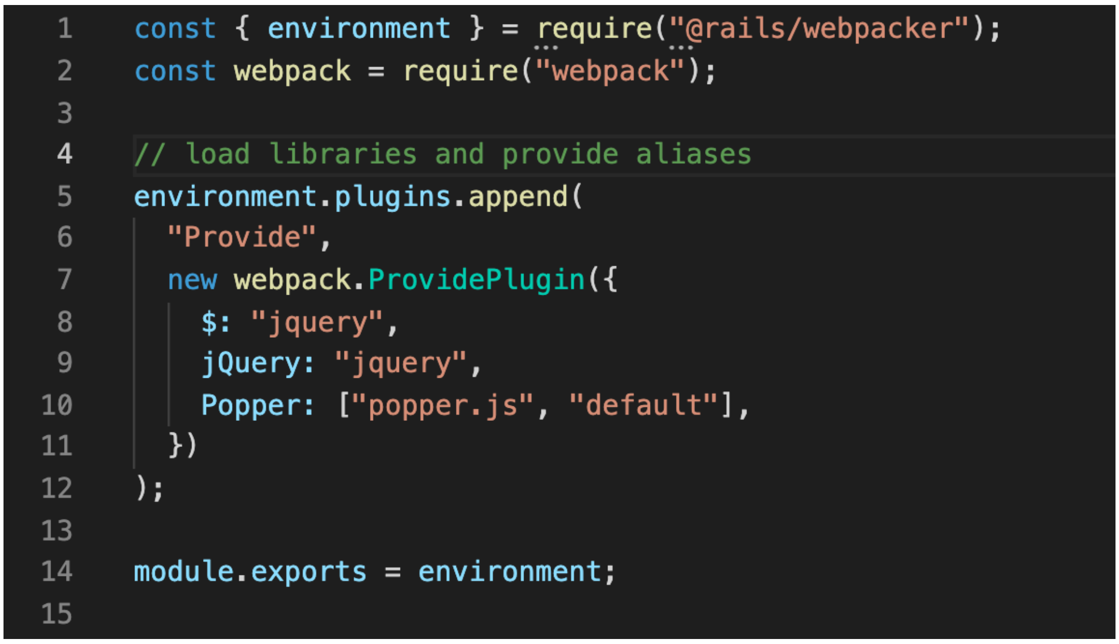


NAME	TAGGINGS COUNT	CREATED AT	UPDATED AT	
Tag A	0	07/03/2021 15:02:21	05/04/2021 15:37:36	Edit Disable Delete
Tag B	0	22/02/2021 17:44:32	05/04/2021 15:37:44	Edit Disable Delete
Tag C	69	22/02/2021 17:44:32	05/04/2021 15:37:52	Edit Disable Delete

Figure 8. Tags -page

3.3 MPA

After the basic setup, which was covered earlier, jQuery and Popper.js JavaScript libraries were installed and made available via aliases (figure 9). MPA demo was relying on these libraries, which are also used by Bootstrap UI library that is under the hood of Qvantel’s Weasel Styles. Bootstrap.js is comes included with Weasel Styles.



```javascript
const { environment } = require("@rails/webpacker");
const webpack = require("webpack");

// load libraries and provide aliases
environment.plugins.append(
  "Provide",
  new webpack.ProvidePlugin({
    $: "jquery",
    jQuery: "jquery",
    Popper: ["popper.js", "default"],
  })
);

module.exports = environment;
```

Figure 9. Webpack environment.js

Rails is a powerful tool for getting working applications out quickly. Considering this, approach here was to first recreate as much as possible of the demo, without JavaS-cript. After setting up the underlying structure and other “under the hood” configu-rations of the application, building the UI (by more or less copy-pasting UI prototype HTML) and populating it with the data via ERB - Ruby templating was very fast. In-cluding helper methods for UI displaying logic, for a fluent Rails developer this could have taken only about a day.

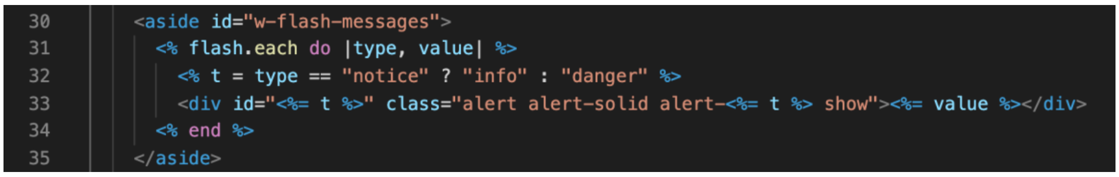
There were no JS interactions at this point, only those that could be achieved with Rails by making requests via links, such as filtering data in tables, and submitting forms (authentication). A lot of intended functionality, that requires custom JavaS-cript still missing.

3.3.1 Basic JS interactivity

Bootstrap library provides basic interactivity, such as dropdown toggles and opening and closing of modals. These were very easy to setup with minimal effort. Following is an example of setting up modals for creating new rule sets:

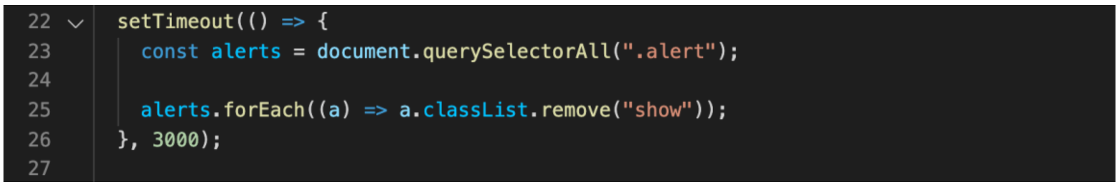
Modal ERB partial is created ("_new_ruleset.html.erb") with specific id HTML attribute. The partial is then included where it is needed, which in this case is in "rulesets/index.html.erb". Modal's id is then used as value for "data-target" attribute on a button that is to trigger the modal.

Other simple interactions could be done with small JS scripts and with combination of Ruby and JavaScript. Example of that are notifications (or alerts). Notifications do not appear in real time, but on page reloads upon user action. In this case Rails controller would serve notifications on requests if needed, which are displayed with ERB syntax (figure 10) and a piece of code on JS side would clean those up after a while (figure 11).



```erb
<aside id="w-flash-messages">
  <% flash.each do |type, value| %>
    <% t = type == "notice" ? "info" : "danger" %>
    <div id="<%= t %>" class="alert alert-solid alert-<%= t %> show"><%= value %></div>
  <% end %>
</aside>
```

Figure 10. Notifications with ERB ruby syntax



```javascript
setTimeout(() => {
  const alerts = document.querySelectorAll(".alert");

  alerts.forEach((a) => a.classList.remove("show"));
}, 3000);
```

Figure 11. Notification cleanup JavaScript

3.3.2 Advanced features and UI limits

There are no real “technical” limits. Everything in the prototype is possible to achieve. Whether it’s feasible was the question.

Limits (or challenges) would start to show with very basic things that are to be expected from a modern web application. Modal submission is a good example. Since the page reloads on submissions, as is expected with MPA, modal would lose its “is open” state. While this would not affect the business logic, the expected smooth closing animation of a modal upon successful submission would be skipped. Modal example in figure 12.

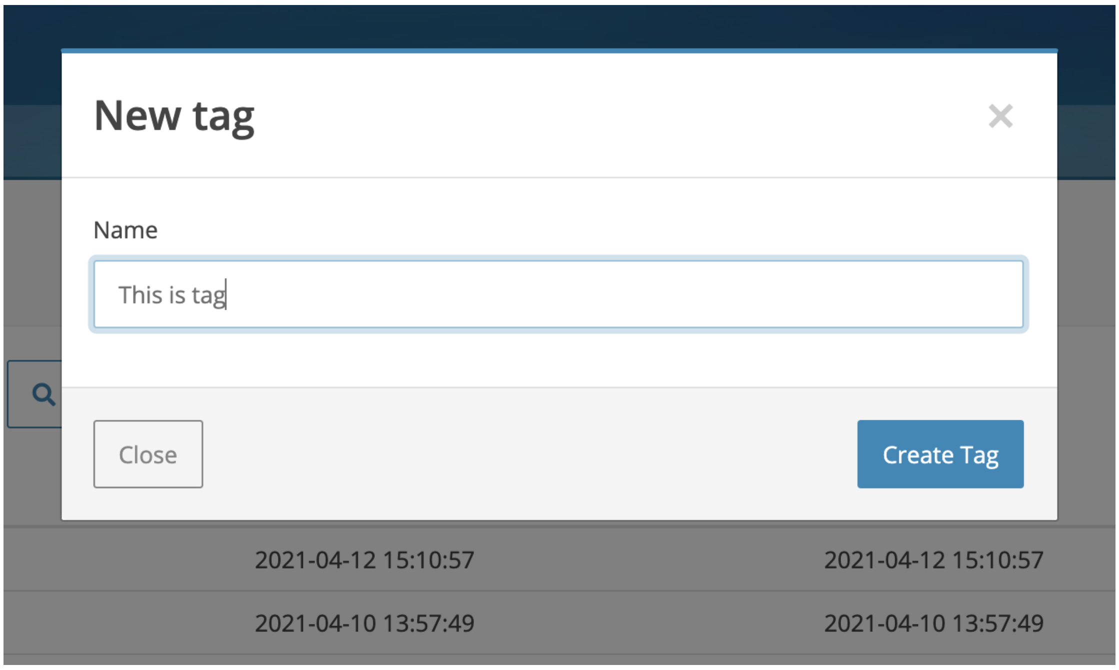
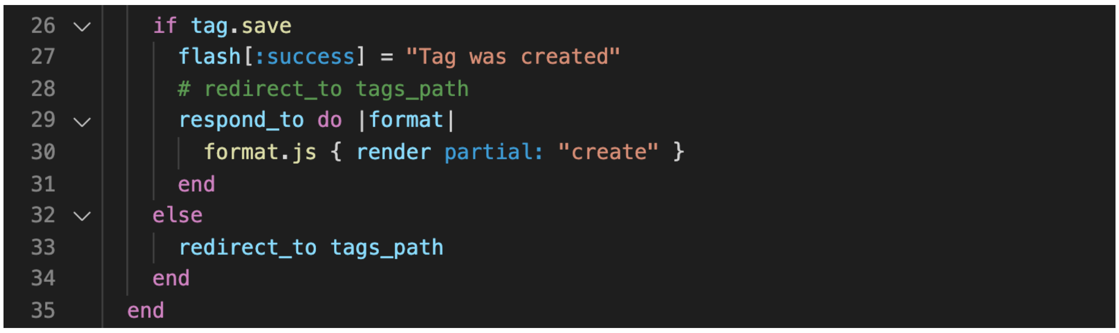


Figure 12. Modal example

Considering the intended flow of opening the modal with animation, closing it with animation upon submission and displaying the updated data, it is impossible to implement without resorting to AJAX submissions or responding in controller with a JS script that closes the modal. This means serving separate JavaScript files as a response to form submissions, instead of redirecting (figure 13). Even then, the new data would not be updated on the screen (because page reload wouldn’t happen), so it would require more effort anyway to make such a relatively straightforward UI interaction work as intended.



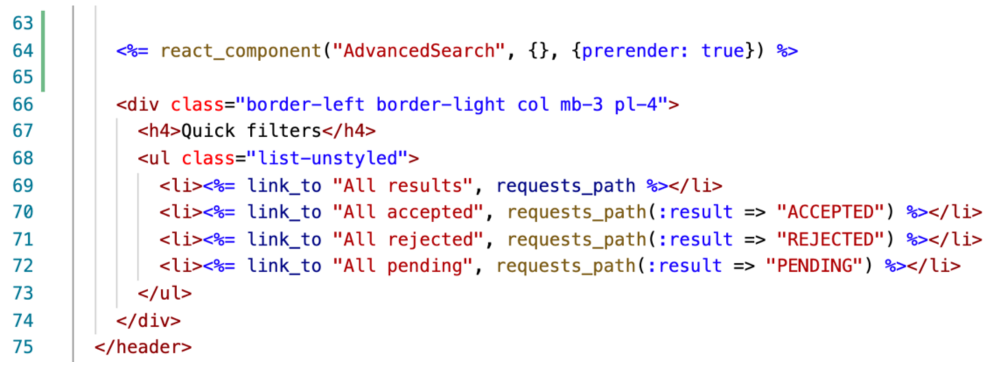
```ruby
  if tag.save
    flash[:success] = "Tag was created"
    # redirect_to tags_path
    respond_to do |format|
      format.js { render partial: "create" }
    end
  else
    redirect_to tags_path
  end
end
```

Figure 13. Responding with JavaScript file

One solution would be to hijack form submission with JavaScript to be handled with AJAX and then handle response, close the modal and then fetch data to be injected into DOM. At this point, introducing React.js and exploring the hybrid approach seemed reasonable.

Another issue relating to modal submissions would arise with editing a resource (a tag again in this example). In a typical Rails MPA “edit resource” -scenario, page would redirect to a path with resource id as a parameter. Controller then would initiate the resource model and a form could be rendered. With editing happening via modal, rather than a page redirect, this creates some additional challenges.

Again, the most reasonable & clean solution would seem to be to handle submissions and responses with JavaScript. This, along with other JS heavy DOM manipulations, as with “Requests” page advanced search feature and “Rule-sets” page multi-category client-side filtering start to add up and push towards hybrid solutions. At this point React was introduced, along with “react-rails” gem, which provides helpers, generators and hooks, that could be helpful. This gem allows controllers to render React components directly, avoiding ERB files entirely, in case the whole page was React. Most notably, it also provides a way to conveniently insert data into-, and pre-render the components. Example in figure 14.



```
    <%= react_component("AdvancedSearch", {}, {prerender: true}) %>

    <div class="border-left border-light col mb-3 pl-4">
      <h4>Quick filters</h4>
      <ul class="list-unstyled">
        <li><%= link_to "All results", requests_path %></li>
        <li><%= link_to "All accepted", requests_path(:result => "ACCEPTED") %></li>
        <li><%= link_to "All rejected", requests_path(:result => "REJECTED") %></li>
        <li><%= link_to "All pending", requests_path(:result => "PENDING") %></li>
      </ul>
    </div>
  </header>
```

Figure 14. Pre-rendering component with "react-rails" gem.

3.4 SPA

3.4.1 Technologies

The demo was built with popular JS modules (or “packages”) in React.js ecosystem. Setup with these packages more or less accurately replicates how a full “real” React application would be built. Most noteworthy packages in the demo, which can be expected to be found in many modern applications that are built with React, are listed in table 1.

Table 1. JavaScript packages (modules)

Module	**Description**
react, react-dom	JS framework / library used in this project
react-router-dom	Tool for managing in-app navigation
redux, react-redux	State management tool and React wrapper
redux-thunk	Middleware for performing side-effects (API / AJAX) calls
final-form, react-final-form	Tools for managing application forms
Axios (axios)	Tool for making AJAX calls.

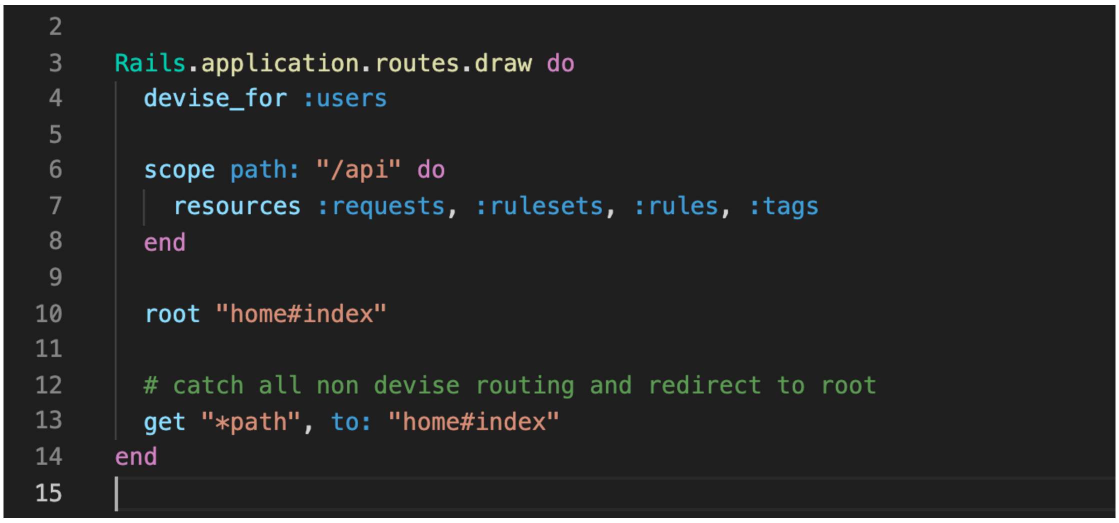
3.4.2 Setup

Rails doesn’t have the friendliest documentation, when it comes to adding JS frameworks and so some of the “gotcha’s” are hard to find, but other than that setting up React in Rails 6 with Webpacker is relatively simple. jQuery and Popper.js were not installed, because all interactivity would be handled with React. Webpacker provides a command to install, and initiate React, which was run after the basic setup (table 2, Initialize). After creating the root component, it is mounted to a page via helper method (table 2, JavaScript tag).

Table 2. Setup commands and code lines

Name	**Command / code**
Initialize	bundle exec rails webpacker:install:react
JavaScript tag	<%= javascript_pack_tag 'hello_react' %>

After the frontend application is mounted on the page, the control is given to React, including URL navigation. This is done by redirecting all non-authentication routing to “home” routes “index” method (figure 12), which just serves html, containing the mounting node.



```ruby

Rails.application.routes.draw do
  devise_for :users

  scope path: "/api" do
    resources :requests, :rulesets, :rules, :tags
  end

  root "home#index"

  # catch all non devise routing and redirect to root
  get "*path", to: "home#index"
end

```

Figure 15. Rails side path redirecting.

With SPA React development approach was component and module based. UI was built one module at a time. That is “Requests” page would be fully functioning, before the other pages would have anything but a placeholder. React utilizes component-based approach. In this context 4 pages, requests, rule sets, rule set and tags would be main components or as they are called in React world, “containers”. Together with Redux (framework agnostic state management tool) provided state store and “dumb” components (also called presentational), these would make up modules. Notifications would form an independent module. Module visualization in figure 16.

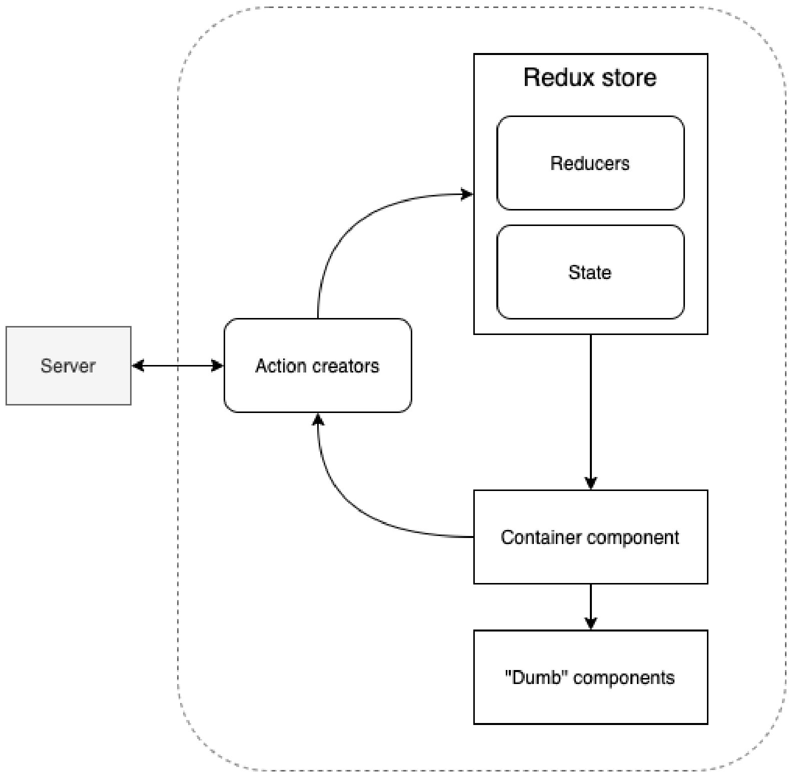


Figure 16. Application React-Redux module

Setting up this additional architecture and each module takes time. Because of that initial productivity in terms of read-only views outputted on the screen (with real data), was slower compared with MPA, where only a view (.html.erb) file needed to be populated. This changes once the foundation was in place and in terms of outputting fully functional views, SPA was faster. Familiarity with React however played a significant part here. With JavaScript in full control, and powerful UI building tools (React) development went smoothly with no issues. UI related Rails helper methods were moved client-side. Models remained identical on Rails side with both MPA and

SPA. Only difference in controllers was in responses, serving JSON instead of rendering HTML.

4 Results and experience

4.1 Developer experience

Client-side interaction with Rails-backend does bring additional layers of overhead with SPA. One is performing and handling AJAX calls with JS for all CRUD operations, rather than doing server-side navigation and form submissions. Other is properly managing client-side state of fetched and local data. In this case, these additional layers were not really a serious issue. Having state management on client-side, React or not, would be needed to some extend to build and manage applications UI properly. If not inside SPA, then on individual pages where needed.

AJAX calls and their management are also still needed with MPA solutions as well to properly implement desired UI functionality. Advanced search feature on “Request log” page was an example of that. Local filtering on “Rule sets” page is another. Both of these features would have needed AJAX calls and managing state locally to fully implement as intended. Some UI features rely on reactions from backend to function as intended, without reloading a page. Modals are a good example, illustrated in figure 17. In the end these additional layers gave the control, needed to make it possible to work on advanced UIs in a convenient way.

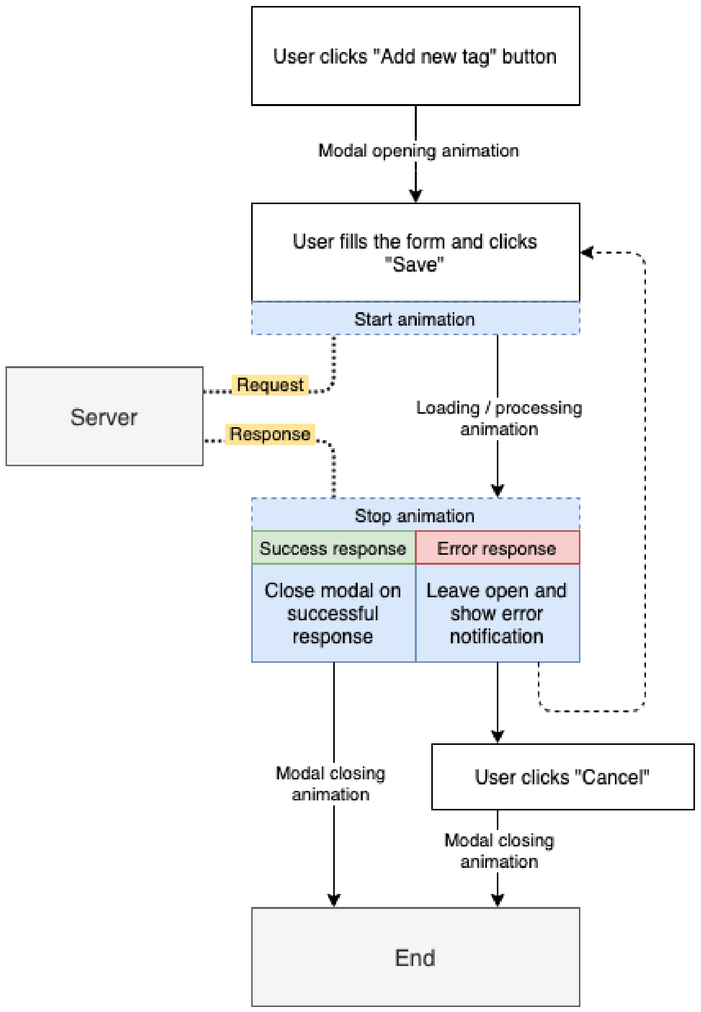


Figure 17. Basic modal submission process

With MPA, finding solutions to problems, that are clean and “obvious” was significantly harder. As could be expected, having more experience dealing with SPA and React in particular, obviously impacted the experience. Had this been reversed, the experience could’ve been different. Despite that however, massive support and community behind React and by extension SPA approached development, had a big impact on efficiency.

The appeal of choosing MPA approach comes in simple architecture. With the mentioned challenges in mind however, instead of staying “simple”, implementing seemingly elementary features was challenging, with rather intertwined ruby (backend-side helpers), ERB-templating and JavaScript code, which seemed harder to keep organized if application was to grow. While architecture remained simple, the readability and maintainability would suffer with tightly coupled backend and frontend code. It would take a very experienced highly skilled developer in Ruby, Ruby on Rails, and JavaScript (full stack developer) to keep the codebase clean, scalable and maintainable, while implementing advanced UI features with ease.

As a frontend developer, SPA approach thus offered the best developer experience of the two. SPA development with React offers massive support of community and resources that are easily available. When using most common node packages and following their defined coding practices, there are but a few possible problems and corner cases that a developer might encounter, which wouldn't be hard to solve with a little of Google and reading of documentation. All used major packages have extensive documentations behind them with lots of tutorials, articles and free courses. Cost of additional architecture pays off when these are taken into account.

4.2 Productivity

Setting up and producing views was noticeably a little slower with SPA than with MPA at first glance. However, with fully functional views and intended advanced UI features implemented, SPA was more productive, while at the same time MPA would slightly stagnate. In maintainability department, there seemed to be lack of clear structure with MPA, with different techniques used in JavaScript code (Vanilla JS, jQuery, React) and scripts being scattered, some on client-side, some served on controller response. Once SPA was setup up, with clear structure and powerful UI building tools (React), building features was trivial. That does not mean that it is impossible to have a clear structure and productive flow with MPA, but it would require more experienced full-stack developers to maintain.

4.3 Maintainability and support

With SPA React, answers to all encountered issues were easily found in documentations, libraries GitHub pages or the infamous Stack Overflow. This shows, that as already mentioned before, React, as well as other modern popular UI frameworks and libraries have a lot of support available. This makes for debugging and maintaining the applications easy. Important note is that when it comes to Rails with React development, questions and answers regarding issues are for the most part SPA specific. It’s noticeably harder to find answers or questions when trying to use React withing MPA on individual component scale. With that in mind, most support can be found for SPA development, when it comes to Rails with React.

JavaScript can be a messy language. There are hardly any limits to how things are done. When it comes to SPA approach with React, established coding standards and patterns with these tools help with keeping the application clean and maintainable, as long as developers keep true to said standards and patterns. It is not unreasonable to assume that new frontend developers will more easily make sense of a project, that is built with widely used modern tools with a lot of support available.

4.4 Performance

The usual argument against using a framework is additional cost by increasing the bundle size, which is a valid concern and especially well presented in Tim Kadlec’s “The Cost of JavaScript Frameworks” (2020) article on JS framework performance. “What is clear: right now, if you’re using a framework to build your site, you’re making a trade-off in terms of initial performance—even in the best of scenarios.” (Kadlec 2020). Performance of JS frameworks in general, was not the subject of this thesis, as this would’ve been big enough of a subject for a thesis on its own. As such, use of a framework (or libraries) to some extent in comparisons is assumed with both MPA and SPA.

Table 3 demonstrates bundle sizes generated with webpack-bundle-analyzer. Static is the uncompressed and non-minified code. Parsed is minified code that is run in the browser. Gzipped is the compressed file, which is sent to the browser.

Table 3. Bundle sizes generated with webpack-bundle-analyzer.

Applications	**Static**	**Parsed**	**Gzipped**
SPA (w-app.js) (application.js)	1.17 MB (804.25 KB) (391.61 KB)	408.6 KB (338.56 KB) (70.03 KB)	117.65 KB (99.55 KB) (18.1 KB)
MPA (without React)	919.77 KB	239.57 KB	69.95 KB
MPA with React (application.js) (server_rendering.js) *	1.68 MB (1.14 MB) (555.31 KB)	645.6 KB (397.81 KB) (247.79 KB)	201.56 KB (120.4 KB) (81.16 KB)

Unsurprisingly MPA bundle without frameworks has the smallest bundle sizes, since it has the least amount of code. Figure 18 visualizes the composition of that bundle. In case of MPA with React (figure 20), “react-rails” gem creates additional bundle (server_rendering.js) with repeating packages, inflating total bundle size. Without “react-rails” gem, the bundle would be similar in size to SPA (visualized in figure 19) when comparing SPA total with that of MPA React application.js bundle.

Despite similar sizes compared to SPA, MPA with React bundle does not contain the rest of React ecosystem tools and packages (table 1), only React itself. That should be enough for MPA implementation in basic cases. This is where React being a UI library, rather than a full framework comes into play. While this can change, with both SPA and MPA bundles growing with additional packages and tools introduced down the line, the bottom line remains. If using React, bundle sizes will not matter in SPA or MPA debate.

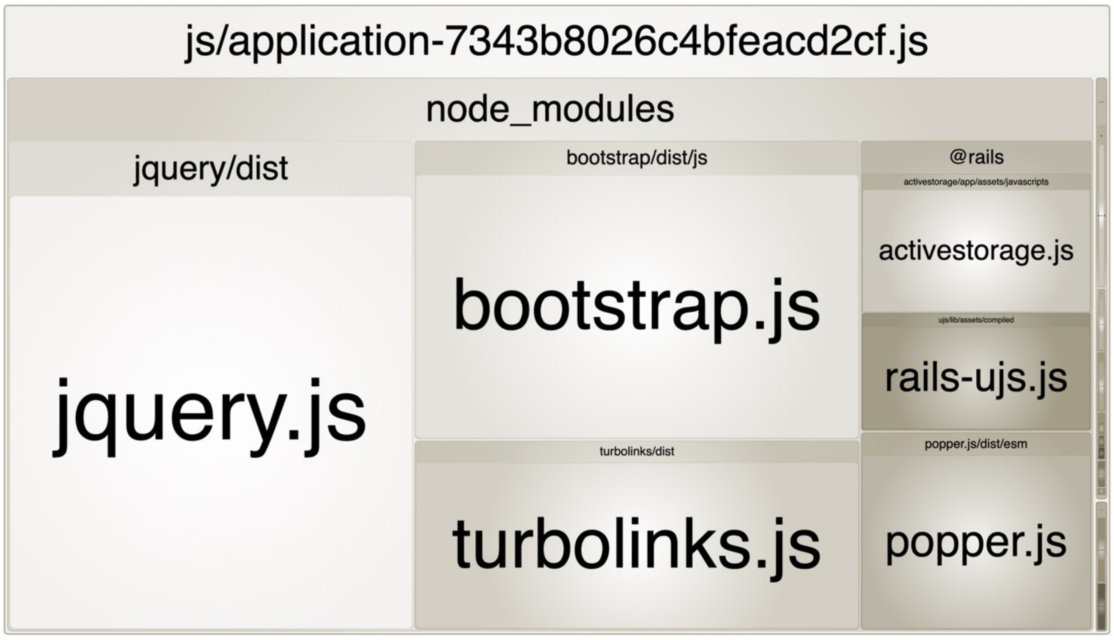


Figure 18. MPA (vanilla) bundle visualization with webpack-bundle-analyzer.

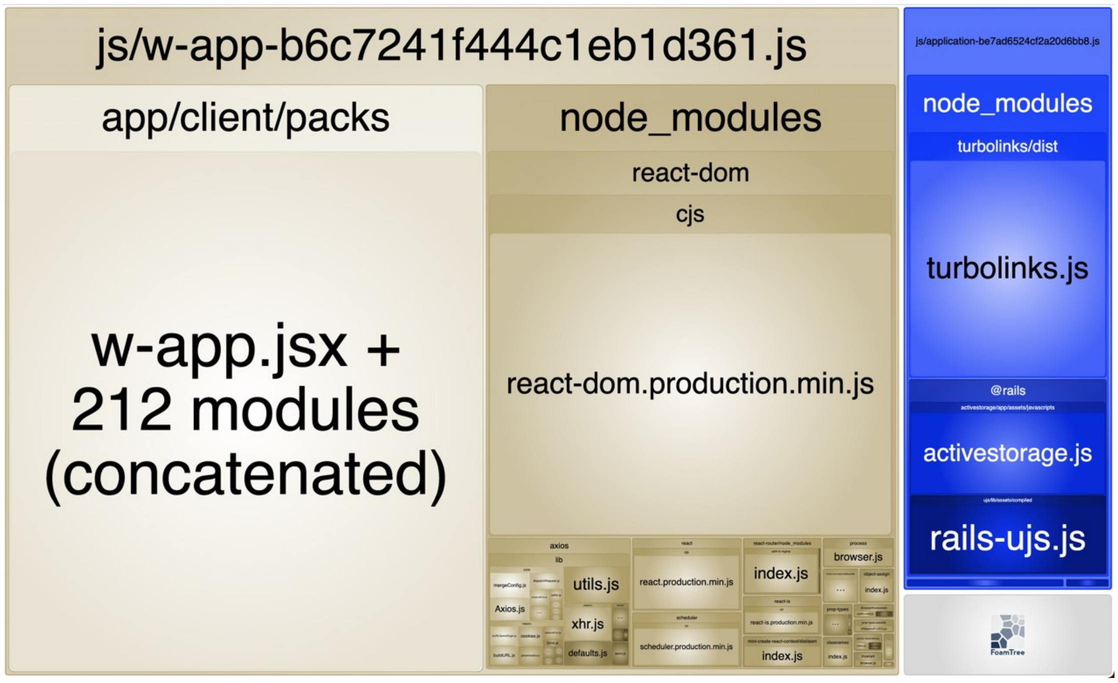


Figure 19. SPA bundles visualization with webpack-bundle-analyzer.

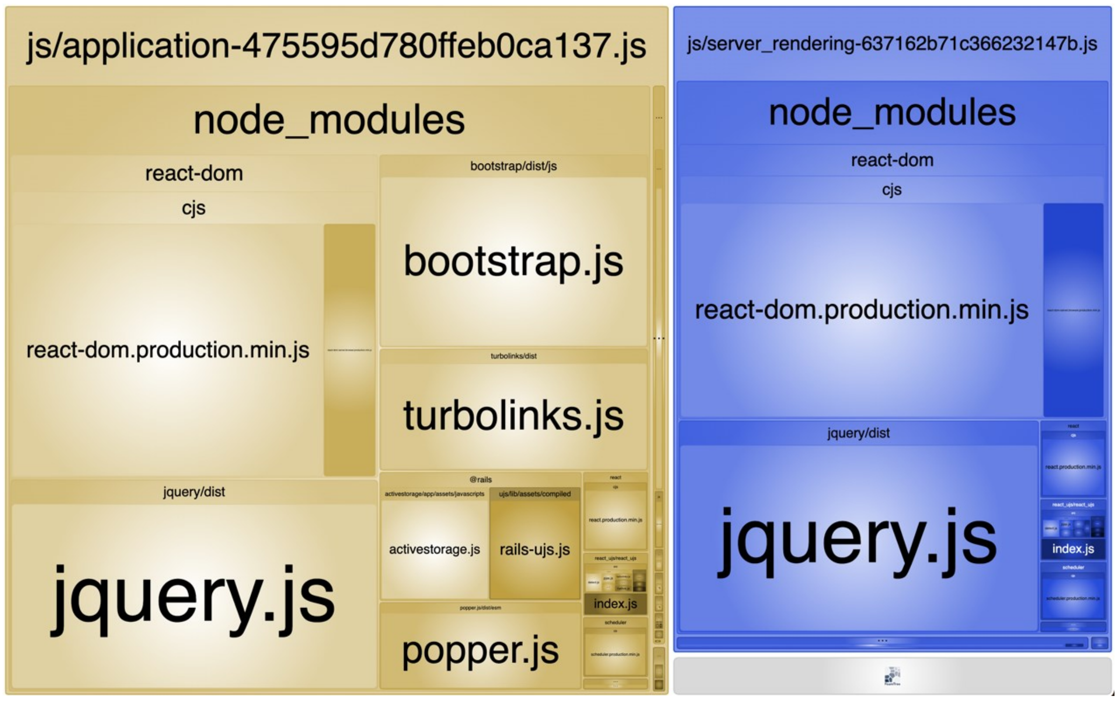


Figure 20. MPA with React bundles visualization with webpack-bundle-analyzer.

Optimizing can help with bundle sizes. Rails Webpacker already does some optimizations with excluding “prop-types” (when using React library) from production builds. There are a few ways to further optimize if needed, one major and potentially most impactful being avoiding global imports of libraries (Galanciak 2018). Example of that in table 4. There were no such cases available in the demo, that could have had a big impact, but moment.js or lodash.js are good examples of such libraries, both frequently used libraries that can bloat the bundle if not being careful. It is also not implausible to assume that MPA with React and “react-rails” could be optimized to eliminate duplicate packages with some research.

Table 4. Global vs. specific imports.

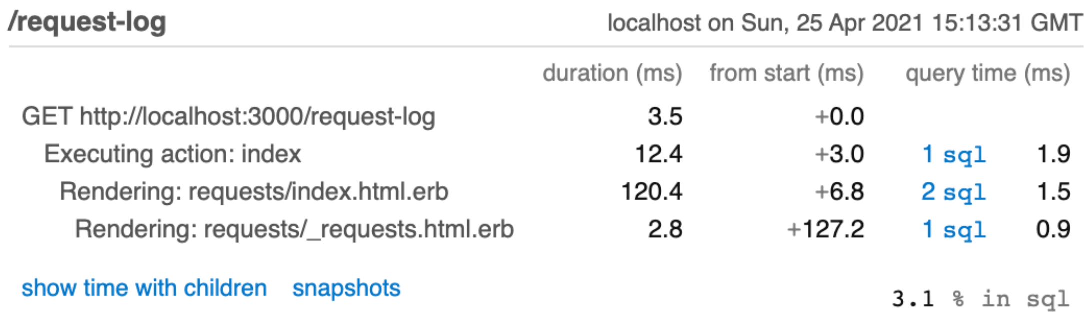
Type	**Import code**
Global import	Import _ from “lodash”; // _.concat(array, [values])
Specific function only	Import concat from “lodash/concat”; // concat(array, [values])

Code splitting could be explored as an option with especially big projects. Overall size of bundles will be slightly bigger, but it will reduce the size of initially loaded bundle (Bernardeau 2020). The recommended way is split to a chunk per route, which is essentially an MPA per page JS file - equivalent. React library in particular provides tools for this “out of the box” with “React.lazy” and “Suspense”.

Rails 6 has a “mini-profiler” plugin, which conveniently shows time spent on each action from request to render. MPA “Request-log” page report example in figure 21.

Simple testing was done with the plugin by running both applications through a certain user flow process manually 5 times, writing down results. The flow contained logging in and navigating to “Request log” page, filtering requests a few times. Navigating to “Rule sets” and singular set pages. Navigating to “Tags” page, creating and deleting a tag. These actions (fetching and rendering views), deletion, and creation took on average between 10ms and 15ms in SPA. Same actions did not have as consistent results in MPA, with “Request-log” logging on average 140ms, while “Tags” page anywhere between 20ms and 90ms.

These results are in line with what could be expected. With more complex rich UI, generating those pages on the server and sending to the client to be rendered would take time (Tarnovskiy 2015).



```
/request-log                                    localhost on Sun, 25 Apr 2021 15:13:31 GMT
                                            duration (ms)  from start (ms)  query time (ms)
GET http://localhost:3000/request-log                 3.5             +0.0
  Executing action: index                            12.4             +3.0    1 sql     1.9
    Rendering: requests/index.html.erb              120.4             +6.8    2 sql     1.5
      Rendering: requests/_requests.html.erb          2.8           +127.2    1 sql     0.9

show time with children   snapshots                                           3.1 % in sql
```

Figure 21. Mini-profiler report on "Request log" page.

Qvantel applications are B2B (Business-to-business), not B2C (Business-to-customer). The applications themselves are internal tools for use by Telco companies. This means that neither SEO (Search Engine Optimization), nor load speeds and bundle size on the frontend would matter as much as it would with customer facing applications, where every bit of optimization matters. Quickly loading “snappy” applications with good rich UI are easier to sell, but the differences between MPA page reloading, and SPA initial and background loading in this case were small enough not to be a factor. MPA “Request log” page 140ms versus SPA 15ms may seem like a big difference, but those numbers are still unnoticeable to the naked eye. In case of bundle sizes and initial load times, Telco companies will not typically have network speed concerns for this to become an issue.

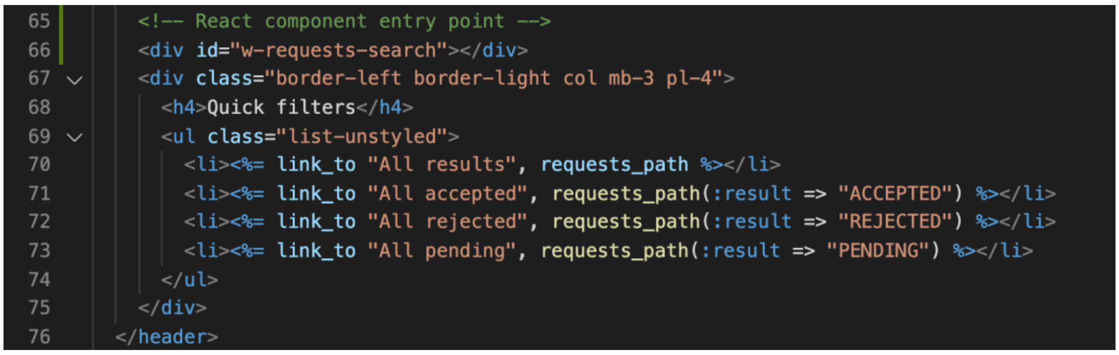
5 Conclusions and follow-up

5.1 Conclusions

It is very hard to deduct which solutions would be clearly better overall. There is no definitive data backed answer, when it comes to “developer experience”, “accessibility of code” or “maintainability”. For the most part these depend on the developer and their experience. A simplified conclusion to make would be that SPA is the way to go with bigger applications with a lot of rich UI elements and a lot happening on the screen, due to its readiness to handle such scenarios. It was designed specifically to deal with these kinds of applications. MPA, due to its speed and convenience, when it comes to quickly setting up, would be a better solution for simpler web applications with simple views and “proof of concept” type of applications, where UI is not of big importance, as Rails MPA is good for focusing on business problems, not technical ones.

Going with default Rails application (MPA), without a modern JS framework, works with very “simple” applications with no advanced UI features planned. “Simple” in this case specifically means UI interactions and functionalities. The underlying application (or backend) can be big and complex, but as long as UI of the application is meant to be simple and crude, and its users don’t care about having as good of UX as possible, MPA without a framework will work very well. CRUD views like “Tags” page (figure 8), could be an example of that. Going SPA might even be counterproductive in such a case, as the cost of setting it up may not pay out when applications UI remains simple. It will still have the benefits mentioned, just unnecessary.

MPA does not equal “no framework” however, there is no reason not to use a frontend JS framework or a library, whether it is MPA architecture or not, if it solves a problem in the best possible way. It was much more convenient and productive (which equals better developer experience) to implement rich UI features (modal flow illustrated in figure 17) with React. In Rails this can be done very easily by creating a React entry file in “/packs” folder, which automatically makes it available as a tag to be inserted where needed (Figure 22). This is the “default” way, only using Webpacker setup. Alternatively, and how it was done in MPA demo, with the help of “react-rails” gem, render components directly (Figure 14). Good news is that this “framework” can be added at any point later in development, it does not have to be decided right away. Being essentially a “plug-and-play” solution with Rails Webpacker, it can be introduced when needed.



```
  <!-- React component entry point -->
  <div id="w-requests-search"></div>
  <div class="border-left border-light col mb-3 pl-4">
    <h4>Quick filters</h4>
    <ul class="list-unstyled">
      <li><%= link_to "All results", requests_path %></li>
      <li><%= link_to "All accepted", requests_path(:result => "ACCEPTED") %></li>
      <li><%= link_to "All rejected", requests_path(:result => "REJECTED") %></li>
      <li><%= link_to "All pending", requests_path(:result => "PENDING") %></li>
    </ul>
  </div>
</header>
```

Figure 22. Request log search - React component entry point.

SPA works very well with Rails, at least as of version 6. Rails in itself did not present any serious obstacles or challenges for implementing SPA architecture, nor create additional complexity compared to a typical SPA with external backend - setup. With that in mind, SPA route would seem to be the best option for most Qvantel applications, as most will need some levels of advanced UI features and will have complex views with a lot going on, not just simple CRUD views. It is especially good for projects with dedicated frontend and backend developers. SPA architecture has clearly decoupled frontend and backend, which is an asset for such teams. Backend developers can concentrate on backend, while frontend developers build frontend.

Going this route also gives full control to JS and frontend developers from the beginning and in a more general sense ensures scalability in terms of easier integration of another frontend to the same backend. Mobile application for example. Great support and resources, and established patterns and tools greatly help maintaining the project and offer good developer experience.

The benefits outweigh the argued additional cost of setup. Typical argument against SPA is its cost. “Single-page apps are much more expensive to build than multi-page apps” (Navis 2018). This however largely depends on team composition and their tech-stack knowledge. For someone well versed in React and SPA architecture, building SPA within Rails was much easier and faster than MPA, despite initial setup being at a glance faster with MPA. In the end the tools and patterns SPA frameworks offer to keep the codebase accessible and maintainable will only be as good as the developer using them. Same applies to MPA. “Technology is a tool, not a philosophy”, a cliché sounding line that is often seen in developer job postings, applies here well.

Having a good comprehensive application design and UI prototype beforehand and having a clear vision of features that will be introduced later can help identify whether MPA or SPA is the right choice, but in reality, this kind of perfect scenario simply does not exist. Even with a ready-made well-crafted UI prototype (such as the prototype on which the demos were based off), new features, random changes and scope creep will happen.

Starting with MPA, then gradually adopting SPA, one page at a time can be a great solution for cases where UI is initially supposed to be simple but changes in the future, which happens more often than the first scenario. Fortunately, option of going “hybrid”, and eventually SPA if necessary, is very much viable with Ruby on Rails, as Rails application (backend) does not need to be converted to “API only” application.

Same controllers can be used to serve both HTML for server-side pages and JSON for JS API route fetched React pages (or other JS frameworks). Pages that don’t need to be part of SPA, can remain separated. Login page is a demonstration of that, rendering “Devise” login page when user is not authenticated and loading SPA view once logged in.

5.2 Follow-up

Other frameworks could be explored with Rails. Hybrid approaches especially could be explored more in depth with different frameworks, other than React and its ecosystem. Vue.js and Angular.js are two very popular frameworks that could offer different perspective on how SPA, MPA or hybrid could be implemented with Rails.

There are also JS libraries, such as Stimulus.js, which are designed to work with Rails. These could have offered a better perspective on MPA with advanced UI features.

Spending more time getting comfortable with Rail development and MPA approaches of implementing UIs in big applications in general could provide new outlook on SPA versus MPA debate. Many obstacles encountered with MPA implementation possibly could’ve been avoided with more experience.

References

Galanciak, J. 2018. Analyzing, visualizing and optimizing JS bundle size in Rails/Webpacker apps. Referenced 19.04.2021. https://razorjack.net/visualizing-optimizing-javascript-bundle-size-rails-webpacker/

Gerchev, I. 2020. A Beginner's Guide to Webpack. Sitepoint.com website. Referenced 19.04.2021.
https://www.sitepoint.com/webpack-beginner-guide/

Goodrich, G. 2017. Understanding the Model-View-Controller (MVC) Architecture in Rails. Referenced 20.10.2020.
https://www.sitepoint.com/model-view-controller-mvc-architecture-rails/

Model-view-controller. Wikipedia.org website. Referenced 21.10.2020.
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Navis, G. 2018. The Architecture No One Needs. Referenced 19.04.2021.
https://www.gregnavis.com/articles/the-architecture-no-one-needs.html

Rails has everything you need. Rubyonrails.org website. Referenced 20.10.2020.
https://rubyonrails.org/everything-you-need/

React (JavaScript library). Wikipedia.org website. Referenced 19.04.2021.
https://en.wikipedia.org/wiki/React_(JavaScript_library)

Separation of concerns. Wikipedia.org website. Referenced 21.10.2020.
https://en.wikipedia.org/wiki/Separation_of_concerns

Smith, S. 2020. Overview of ASP.NET Core MVC. Referenced 21.10.2020.
https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1

Tarnovskiy, S. 2015. Mixing MPA and SPA: worst of both worlds. Referenced 19.04.2021. https://blogs.perficient.com/2015/01/26/mixing-mpa-and-spa-worst-of-both-worlds/

Webpacker. Rails guides website. Referenced 19.04.2021.
https://edgeguides.rubyonrails.org/webpacker.html

Welcome to Rails readme. Referenced 20.10.2020.
https://github.com/rails/rails