

Ishwor Khadka

Converting Multipage Application to Single Page Application

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Thesis

20 March 2016

Author(s) Title	Ishwor Khadka Converting multipage application to single page application
Number of Pages Date	41 pages + 4 appendices 20 March 2016
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Specialisation option	Mobile Programming & .NET Application Development
Instructor(s)	Ilkka Kylmäniemi, Senior Lecturer
<p>Heli, Health-e-living, is an existing service dedicated to people who want to pursue a healthy life. The service provides tools to plan for a better way of living in close supervision of professional health practitioners. The service exists as a multipage application. Hence, the platform coverage of Heli was limited to browsers only. As a result, the necessity of taking Heli to wider platforms seemed relevant and the idea of converting the Heli service to a single page application (SPA) emerged as a solution.</p> <p>The purpose of this final year project was to build a single page application for the Heli service. EmberJS was used to build the front end of the application. Subsequently, the existing application was modified to serve as a data API. The modification was done in a way that the new changes would not interrupt the existing service until the SPA version was ready to be released.</p> <p>As a result of this project, Heli will have its current multipage application replaced with the SPA version. Eventually, it will be used to create native applications for multiple platforms. With that, it is expected to reach wider range of users regardless of the device of their choice. Users are expected to have a native application like user experience on every supported device.</p>	
Keywords	SPA, JavaScript, EmberJS, EmberCLI, Rails, web application

Abbreviations and Terms

Heli	Health-e-living
SPA	Single Page Application
MVC	Model View Controller
CSS	Cascade Style Sheet
JS	JavaScript
AJAX	Asynchronous JavaScript and XML
MVVM	Model View View-Model
RoR	Ruby on Rails
CORS	Cross Origin Resource Sharing

Contents

1	Introduction	1
2	Single Page Application	2
2.1	SPA vs Multipage Applications	2
2.2	Advantages of SPA	4
2.2.1	Maintainable Code	4
2.2.2	Minimizing DOM Dependent Codes	4
2.2.3	Independent of Server Side Code	5
2.2.4	Smoother Transition, Richer User Experience	5
2.3	Disadvantages of SPA	6
2.3.1	Chances of JavaScript Memory Leaks	6
2.3.2	Memory Issues with Handheld Devices	6
2.4	JavaScript and SPA Frameworks	7
3	Backend Technologies	9
3.1	Ruby on Rails	9
3.1.1	MVC in Rails	9
3.1.2	Environment Modes	13
3.1.3	Test Driven Development	13
3.1.4	Rails Philosophy	14
3.1.5	Why Does RoR Stand Out?	15
3.2	RESTful API	15
4	Frontend Technologies	17
4.1	HTML5	17
4.1.1	Semantic Elements	17
4.1.2	Audio and Video Tag	17
4.1.3	Forms in HTML5	18
4.1.4	New APIs	18
4.2	CSS3 and SASS	18
4.3	EmberJS	21
4.3.1	Base Object Blueprint	21
4.3.2	Model	22
4.3.3	View	22
4.3.4	Controller	23
4.3.5	Route	24

4.3.6	Component	25
4.3.7	Testing	26
4.4	EmberCLI	26
5	The Project	28
5.1	Framework Selection	28
5.2	Challenges	29
5.3	Communication with Rails Backend	32
5.4	Authentication	35
5.5	Authorization	37
6	Conclusion	38
	References	39

1 Introduction

Health-e-Living is an online personal health coach for all citizens wanting to improve their lifestyle with better health habits [1]. Health-e-living, abbreviated as Heli, is a system targeted to people who want to pursue a healthy life. Heli gives them the tool to do that. It helps them to track, plan their activities and diets in a close observation of a professional health practitioner. In a way, Heli is a personal health coach. It provides tools to set one's own goals and plan daily activities including diet and exercises in order to achieve the goals. Users can keep in touch with their selected health practitioner referred to as a caregiver in Heli system with the messaging tool that Heli provides. In addition, Heli also provides a nicely illustrated analysis of the progress.

The company behind the idea and development of Heli is called Extensive Life Oy. It is based in Tampere. I had an opportunity to do an internship with the company and had an involvement in the very initial development phase of Heli as an intern in 2012. The initial version was developed with Rails framework in the traditional way of server side page rendering. The new idea for Heli slowly evolved as it became limited to web users. There was obviously a need of taking it from browsers to native mobile platforms. With the availability of a backend data API, it also opened a possibility to make the web version better.

As a result, the idea of building a single page client application for Heli evolved. The sole goal of my final year project was to make the Heli service more efficient and provide a richer user experience.

2 Single Page Application

In the early stage of the World Wide Web, websites were a collection of a couple of static pages, often used as a digital way of advertising [2]. As time passed JavaScript, CSS and several other technologies were invented that changed the meaning of websites drastically. At present, websites are not only means for advertising, but they serve as a tool for complex services that would otherwise only be offered by only native applications. Single Page Applications are the recent trend, often referred to as web application rather than websites.

Single Page Applications, often abbreviated as SPA, are Web apps that load a single HTML page and dynamically update the page as the user interacts with the app [3]. As the name suggests, SPA is an application which consists of single container page. When the user interacts with the application or navigates to a page, the application does not make a full page reload. Instead, only the specific fragment of the page is loaded asynchronously and rest of the contents in the page remain unaffected.

AJAX is a JavaScript technology that allows web application to make an asynchronous request from the server. It is the basis of a single page application. After the introduction of HTML5, the SPA has become even more popular. One of the reasons behind it is the implementation of new sets of HTML5 specifications that greatly favour the asynchronous loading. There has been a lot of advancement in the ideas and implementations of SPA. Terms like Model-View-Controller (MVC), Model-View-ViewModel (MVVM) patterns, data bindings and template were introduced along the way. [4.]

2.1 SPA vs Multipage Applications

Multipage applications are web applications that retrieve an entire HTML page from the server in every single request. Before the introduction of AJAX, it was a high time for multipage applications. A lot of popular server side languages and frameworks based on PHP, Ruby, JAVA were available. They were responsible for rendering templates right in the backend and providing a raw HTML page as a response. With the introduction of AJAX, developers started to implement asynchronous loading in some parts of a webpage which seemingly needed smooth transitions and UI interactions. However, not all the pages were loaded asynchronously. The architecture of a web application

became mixture of both approaches of synchronous and asynchronous loading. Implementing asynchronous features needed significant amount of JavaScript codes on the client side. There were chances that the JavaScript code could grow, spread everywhere and quickly become very difficult to maintain. Unmaintainable and messy codebase is the last thing a good developer would want. This could be one of the biggest disadvantages of a traditional web page that implements AJAX. The goodness of asynchronous loading could only be obtained in an expense of messier and more unmaintainable JavaScript codebase. [5.]

The efforts that the library like jQuery and some server side frameworks put to make JavaScript code easier and tidier were quite noticeable. In spite of those efforts, the result was still not good enough in favour of multipage application compared to what SPA had to offer. [3.]

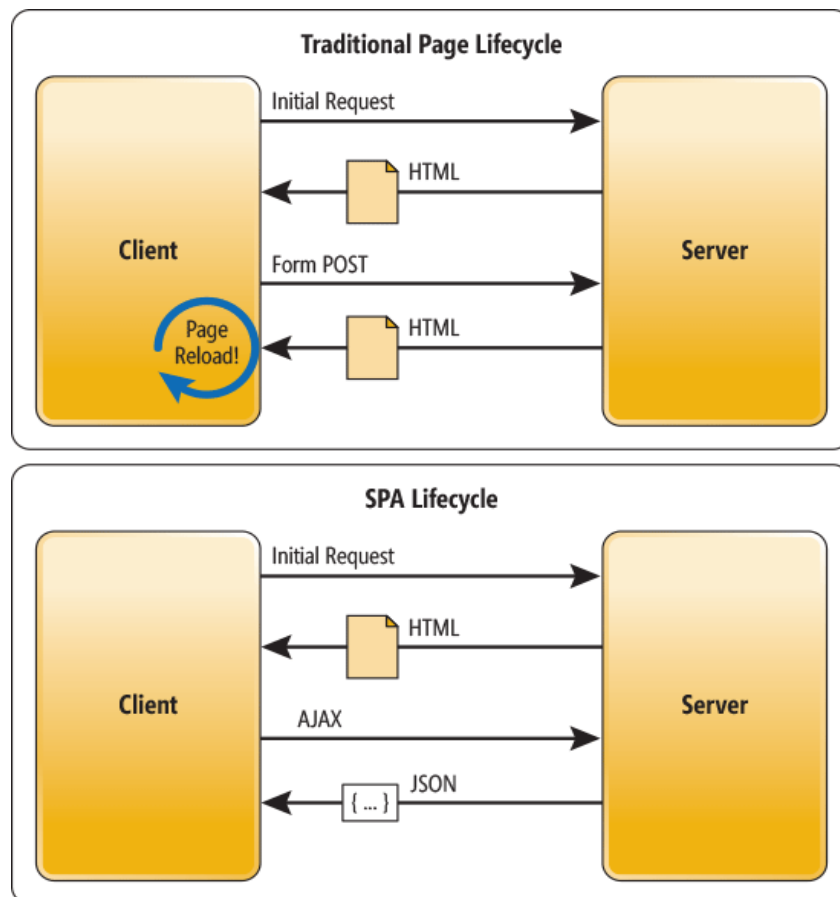


Fig 1. Difference between the lifecycle of a SPA and a traditional page. Reprinted from Wasson M (2013) [5].

As seen in figure 1 above, there is a distinct difference between the lifecycle of a traditional web application and a SPA. When it comes to a traditional page, for every request a full HTML page is sent from the server. Thus, a new page is loaded in the browser. This is also called a synchronous request. While in an SPA, only the first request receives a full HTML page. Right from the second one, the requests are responded to with JSON data which is meant to be rendered with a client side template. These are called asynchronous requests, as it does not disturb the execution of the running application. A call back function is called when the request is completed. [5.]

2.2 Advantages of SPA

SPAs have brought an entire new angle to the user experience in web applications. SPAs have plenty of mentionable advantages over multipage applications.

2.2.1 Maintainable Code

SPA frameworks implement design patterns like MVVM and MVC. For instance, with MVVM the application is divided into three layers which are Model, View and ViewModel. Model is the data source for the application. View is the presentation of the application which is visible to users. And ViewModel is the intermediary layer between View and Model. ViewModel contains public properties and event handlers which are meant to be exposed to the view layer. It is also described as a state of data in the application as it manages the current state and is responsible for any changes that happen as a result of user interactions. With any of these design patterns, there is a firm segregation between different components of the application. This way, all data logic goes to the Model layer, view specific logic to the ViewModel layer and presentation logic and template to the View layer respectively. In a way, they have their own independent existence. As a result of this, the codes are well structured and well placed. Hence it makes it easier to maintain codes in the long run.

Likewise, the features like custom components, directives in AngularJS make it possible to define custom HTML tags. They can be reused across applications. This in general reduces duplications and makes code more modular. [3.]

2.2.2 Minimizing DOM Dependent Codes

In web applications, the JavaScript codes are used to manipulate the structure and contents of the page dynamically. To be able to manipulate an element on a web page, the element needs some kind of identifier which the scripts can identify. As an identifier, the element can use attributes like ID and class. The script uses that identifier to select that element before any manipulation can be done on it. Even a slight change to the structure or the identifiers in the DOM can make the scripts fail. In order to make the script work again, it needs to be modified to synchronize with the changes as the script is tightly coupled with the identifiers. However in SPAs, the contents and the attributes in view can be bound with the value exposed from a controller. With this the DOM updates itself whenever there is a change to the bound value. The DOM is completely in sync with the data from the controller without having to store it in the DOM itself. It reduces the amount of codes which would otherwise be required to update the changes on a page. Likewise, the user interactions with different parts of the page are handled with logics present in the logical layer behind View layer. Event handlers are defined in the respective Controller or ViewModel layer for that page. Therefore, SPA provides a way to get rid of tightly coupled dependency of JavaScript codes with the physical structure of a page. [3.]

2.2.3 Independent of Server Side Code

SPA is the client side of an application that solely runs in a browser. It communicates to REST JSON API server. It does not matter how the server side code for the API is implemented, or what language it is using. As long as it is a RESTful API that provides JSON data, it is a valid API for a SPA. Therefore, SPA is independent of server side codes. If the backend code needs to be modified there is no need to change anything in the frontend as long as similar data is being served. For this reason, it makes it possible for either the frontend or backend part to be developed and tested independently. [3.]

2.2.4 Smoother Transition, Richer User Experience

One of the main advantages of SPA over traditional sites should be the way transition between pages can be controlled and smoothed per requirement. On traditional

websites, when a new page is navigated through, there is a short transitional period when the entire screen goes blank right before contents are displayed. From the user interface perspective, this cannot be considered good. However with SPA, the transitional blank out is easily avoidable. In addition, meaningful information can be conveyed while the content is being loaded from the server, just in case it takes a noticeable amount of time. Therefore, the UI can be improved significantly with SPA. [3.]

2.3 Disadvantages of SPA

SPA looks promising in terms of the development of quality web applications. The user experience is boosted up drastically giving a native application like UI. However, it cannot be neglected that there are some disadvantages as well.

2.3.1 Chances of JavaScript Memory Leaks

The entire application logic of SPA is written in JavaScript. SPAs are compiled by browser in runtime. According to JavaScript way of programming, it is very obvious that a significant amount of event listeners and processes scheduled to run continuously in fixed interval exist. Those listeners and processes are to exist in a particular page but not in others. If a developer or the framework loads them globally and fails to remove those when navigating to a different page where those functionalities are not required, there can be issues with memory leaks. Those processes and even listeners need to be disabled manually to free up the memory use. This could result in memory leaks and could decrease the performance efficiency of the application. [6.]

2.3.2 Memory Issues with Handheld Devices

The browser is responsible for compilation and evaluation of JavaScript code in the runtime. Obviously, if there is lot of code the browser needs more memory for the process. If the application is run on a device that has enough memory to lend for the process, this issue seems to be negligible. However, if it is run on a device which has less memory installed on it, there might be a problem. The application may suffer from issues related to lack of processing memory. Some of the visible effects could be that the application freezes constantly, that the application runs really

slowly or even that the application sudden crashes out. As handheld devices nowadays are available with enough memory installed by default, SPAs can have a bit of a relief. [6.]

2.4 JavaScript and SPA Frameworks

While HTML is used to define the structure and content of a web page and CSS encodes the style of how the formatted content should be graphically displayed, JavaScript is used to add interactivity to a web page or to create rich web applications [7]. JavaScript is the heart of an interactive web application and of an SPA. It is possible to create an SPA because of the features like AJAX and dynamic DOM manipulations provided by JavaScript. Since SPAs were introduced, a lot of JS frameworks have been developed for building SPAs. Those frameworks seem to be the best solution for implementing SPAs. Along the way, each framework had its own significance in giving a base for progressive advancement and development of current core ideas of SPAs. However, only some of the frameworks became popular while the rest did not get much attention. The frameworks that gained popularity among developers are Backbone, CanJS, SpineJS, BatmanJS, Meteor, EmberJS and AngularJS. [7.]

No framework is bad in itself. It is just the application requirements which make one framework more suitable than other. Currently, for SPA any of the above listed frameworks could be an option. Each of them incorporates MVC or MVVM design pattern in some form.

The major differences among some popular SPA frameworks are illustrated in figure 2.

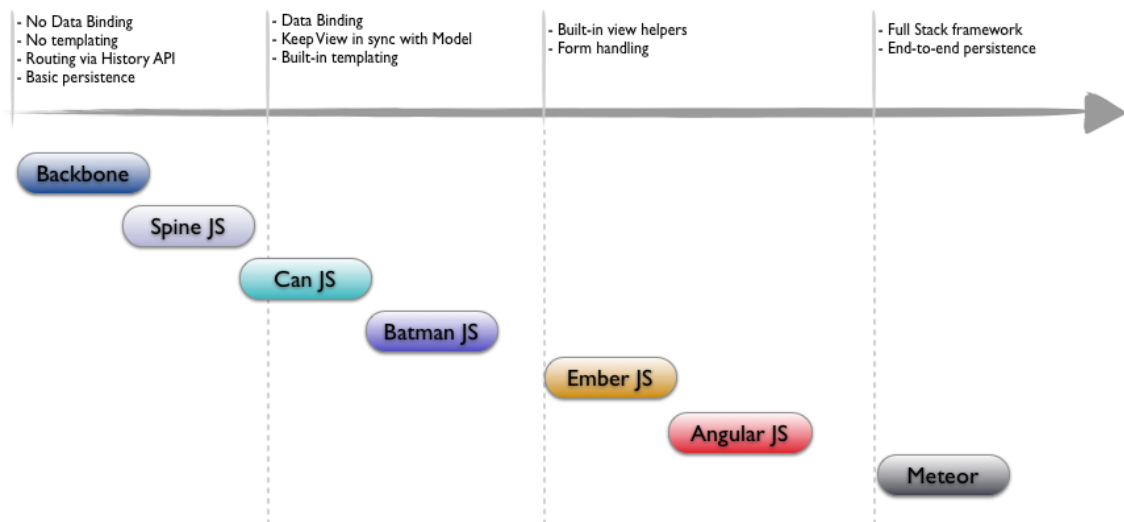


Figure 2. Comparison of different SPA frameworks. Reprinted from Podila P [7].

As illustrated in figure 2, frameworks like Backbone and SpineJS lack some of the features like data-binding and templating. External libraries need to be used to incorporate the functionalities. From the CanJS framework, data-binding and templating have been included by default. Starting from EmberJS, additional features like built-in view helpers and form handling have been included which make them a complete framework for an SPA. Finally, the framework on the right is Meteor. Meteor includes not only a full stack front end framework but also backend components, unlike the other frameworks which depend on other backend API frameworks.

3 Backend Technologies

Backend technologies are web technologies which are not in direct interaction with the end users. The server side languages like PHP and Ruby on Rails and services like database and mailer come under backend technologies. Basically, a backend technology is responsible for providing server side operations, data persistence and distribution. In SPA, the backend is data API that provides data to the frontend.

3.1 Ruby on Rails

Ruby on Rails® is an open-source web framework that is optimized for programmer happiness and sustainable productivity. It allows developers to write beautiful code by favouring convention over configuration. [8.] Ruby on Rails is an open-source, full-stack framework for developing database-backed web applications according to the Model-View-Control (MVC) pattern written in the Ruby programming language. Ruby on Rails is often referred to as RoR among developers and is completely free to use. It consists of all the features and functionality which are required by any type of web application and also it has all the servers and database servers bundled in the package by default. [9.]

3.1.1 MVC in Rails

MVC is an application pattern which divides the application into three layers namely Model, Controller and View. Each of these layers has a set of roles in the application and is capable of communicating with each other. There is a restriction in the communication between View and Model though. If it is needed, they have to get assistance from the Controller which each of them can communicate with directly. This particular application pattern is very famous nowadays, as it makes the application neat and well organized. MVC pattern is being used in RoR since its invention. With every new version RoR has some of the roles moved between the three layers. [10.]

- Model

Model refers to the layer that is responsible for serving data to an application. It facilitates creation, manipulation and retrieval of data in the application. It is the place where the server side data validations happen. The associations among different Models are defined if the Model is related to another. In RoR, the Model is backed by database table which makes the Model to be able to save data. Therefore, in a RoR application, a Model is a ruby class which provides functionalities for creation, retrieval, validations and persistency of data. [11.]

In RoR, the Model is implemented with a class called ActiveRecord::Base. Therefore, all the Models in the application are inherited from this base class. An example of a RoR Model is shown in listing 1.

```
Class Post < ActiveRecord::Base
  attr_accessible: title, :content, :date, :user_id
  belongs_to :user
  validates :title, :presence=>true
end
```

Listing 1. Rails Model

Listing 1 illustrates what a basic Model in RoR looks like. The attr_accessible property defines the properties of the model, which may correspond to the columns in a respective database table. Similarly, belongs_to defines the relationship with the user Model. Finally, the last line defines a validation rule for the column called title. It enforces the title property to be present when a new record is created or when an existing record is updated.

Creating a record with the Rails Model is simple as shown in listing 2 below.

```
Post.create (title:"My First Post",content:"Hei guys!!! this is
the first post from me", date:"29-04-2013", user_id:1)
```

Listing 2. Creating a model record

Similarly, retrieving data from Model is also quite straight and easy to implement. Listing 3 shows some of the ways the retrieval is done from a Model.

```

Post.find(1)
Post.find_by_user_id(1)
Post.find_by_title("My First Post")

```

Listing 3. Retrieving data

As seen in listing 3 above, `Post.find(1).user` gives a record of the user to whom this post belongs. This is made possible because of the association defined in the Model.

- Controller

Controller is the middle man between the Model and View. If the Model needs to communicate with the view or vice-versa, they have to get help from the controller as it is the only layer which can have a direct communication with all the layers in MVC. The Controller is the place where the HTTP request sent from the client side is evaluated and which performs the action as requested. It is where the appropriate output is produced and returned back to the client. When a View needs data from the respective Model, Controller is the layer which gets the data from the model and passes back to the view in an appropriate format. [12.] A sample Controller is shown in listing 4.

```

class PostsController < ApplicationController
  respond_to :html, :js, :json, :xml
  def index
    @posts=Post.all
    respond_with @posts
  end
  def new
    @post=Post.new
    respond_with @post
  end
  def create
    @post=Post.new(params[:post])
    if @post.save
      redirect_to posts_path
    end
  end
end
end

```


Listing 4. Sample code for a Controller

As can be seen in the listing above, Controllers in RoR are inherited from a class called ApplicationController. The base class contains all the methods and properties which a Controller requires to perform its tasks. Each public function corresponds to a specific URL endpoint.

In a RoR application, when a URL is entered in the browser, it looks for a match in routes. A route is simply a collection of URLs mapped to the respective action defined in a Controller. When it finds a match in the routes, the request is redirected to the mapping action which does the required retrieval, creation and deletion of the data in the Model. Finally a response is passed back to the client in an appropriate format. [13.]

- View

View is basically what is seen on the webpage. In RoR, a view template is added with an extension such as “.erb” which stands for embedded Ruby. In Views, it is possible to write ruby codes along with the HTML tags, which is what a normal HTML page would contain. These ruby codes in the template are compiled by RoR into respective meaningful raw HTML. In Views, it is possible to use a helper Ruby function to generate commonly used HTML contents. For example there are AJAX helpers, HTML tag helpers, and form helpers which can be used in the view template, making it easy to take advantage of the ruby codes and utilize the data passed by the controller as a ruby object. In RoR, it is also possible to create custom helper functions. Utilizing these helper functions can help diminish the possibility of repeating the same code more than once. [14.] Listing 5 below illustrates what a simple View looks like.

```
<h1>All the posts</h1>
<%= @posts.each do |post| %>
  <h2><%= post.title %></h2>
  <p><%= post.content %></p>
  <p><%= post.date %></p>
<%end%>
```

Listing 5. Sample code for a Rails' View

Listing 5 shows a typical RoR View. The View is listing all the posts in a blog application. The contents of @posts are iterated to print title, content and date of each of the row. Overall, this is what a basic view looks like in RoR.

However, it is significant to note that if the Rails is used as a backend for a SPA, the HTML views are of no use. The controller responds to the request from the client with the required data in JSON format. Therefore, this layer of the framework is very likely to lose its significance in the near future.

3.1.2 Environment Modes

An application has these three phases as it progresses with the development. For instance, when the code is being written, it is in development phase. Testing phase comes after development where the features are tested by various ways. Finally, there is a production phase when the application is deployed to the server and made available to the users. In a RoR application, all these phases can be experienced when the application is being developed as an environment mode. It is possible to run any of the development, testing and production environment modes. Each of them has their own database so that the application in one mode is not mixed up with another. Each mode has its own configuration. For instance, in the production it is not a good idea to show an error message to the user, but at the same in the development mode, showing an error message helps developers to fix the bugs. [15.]

3.1.3 Test Driven Development

RoR promotes an application development approach called Test Driven Development (TDD). Test Driven Development is a development approach where the test is written beforehand the code for the application is written. According to the TDD approach, when a feature is decided for an application, the test is written first. When the test runs, it fails for sure because there is no code implementing the required feature in the application. Afterwards, the functionality is coded to make the test pass. The functionality is in place when the test passes. [16, 205-206.]

One of the first noticeable points that favours the promotion of TDD in RoR is the inclusion of a testing gem in the core bundle. RoR application has a directory dedicated for

test files. Also, there are plenty of contributed testing gems available like Cucumber and RSpec.

3.1.4 Rails Philosophy

RoR community has some philosophies that it strongly believes in. That may be the reason why RoR is in the position it is now. It is gaining even more popularity. Some of the important philosophies RoR stands up for are listed as below.

- Do not Repeat Yourself (DRY)

Using the same chunk of code repeatedly is discouraged in any part of an application. To avoid duplication in coding, RoR provides functionality where a developer can create custom helper method to achieve a particular function. This way, developers can make use of the same function wherever the functionality is required, without having to write the code all over again. [9.]

- Convention Over Configuration

When a Rails application is created, it is already configured in a way it can be run straight away. The developers do not need to spend time configuring the application. The initial configuration is based on conventions of Rails. For instance according to Rails' convention, a Controller is meant to display the View template which has same name as the action in the controller. In case when there is a need for a Controller to render a different View template, the application has to be configured differently. That is why the Rails community has a belief that it is better to stand with the conventions rather than to have to configure every small part. That way the development can be started without wasting much time in initial configurations. [9.]

- Rails is Opinionated

RoR community believes that there is always a better way to implement a particular functionality in the application. The way a feature was implemented before and the way it is done now has changed a lot because the need to change the way of coding for the betterment was felt. This belief has driven RoR developers to practice a better implementation in every newer version. [9.]

3.1.5 Why Does RoR Stand Out?

Rails is considered as one of the best frameworks available for web development. There are some noticeable reasons for the success of Rails. One of the main reasons for Rails' success is that it is completely free of cost. The philosophies of Rails cannot be ignored either. Similarly, there is an interesting and powerful feature in RoR called Asset Pipeline. In a web application, there are lots of scripts and styling files which have to be included in a webpage. The script and style files can be hundreds in number. When a page is loaded, all script and style files have to be loaded. Obviously, if there are hundreds of files, the loading time increases drastically. However, in Rails with Asset Pipeline, all asset files including JavaScript and CSS can be compiled into one respective file. Then it is included in the page which decreases the loading time since the browser has to make only one request. [17.]

Likewise, Rails has a large active contributing community. That is why the documentation of Rails is very well written. It is easier to find tutorials as well. Also, there are large numbers of contributed gems available which can be integrated into a Rails application as required by the project.

3.2 RESTful API

REST is an architecture style for designing network applications. The idea is that rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines. [18.] REST stands for Representational State Transfer. It is a set of rules that standardize communication with the server.

According to the concept of REST, the HTTP request can be of one of GET, POST, PUT or DELETE types. RESTful applications use an HTTP protocol to read, update or delete a data resource with a correct request type. REST covers create, read, update and delete actions for every resource. RESTful API provides interfaces for these actions. It is platform independent so it does not matter where the API is hosted on or who the client is. Similarly, it is language independent as it is possible to make requests from any of the languages like JAVA, C# or JavaScript, as long as the request follows the REST specifications. It runs on top of the HTTP protocol. It can be used in the presence of firewalls as well. [18.]

The response from a RESTful API may vary based on the requirements of the application. It can be any of the HTML, XML or JSON types. In case of SPA, the response is generally a piece of data in JSON format.

4 Frontend Technologies

Front end technologies are the ones which are in direct interaction with the users. The components like HTML5, CSS3 and JavaScript make up the front end section of web application. These technologies are run in the browser. All the browsers that are available today are in constant effort to implement all the features that are defined in the official standard specifications. However, the way of implementation may vary. Therefore, there are many cases when one code might be working in one browser but not in others. This is the reason why the term “cross-browser” is popularly used in front end programming.

4.1 HTML5

HTML5 is the new version of HTML with new sets of semantic elements, attributes, behaviours and a larger set of technologies that allows more diverse and powerful web applications. There are some significant changes in this new version of HTML.

4.1.1 Semantic Elements

HTML5 introduces tags which are more meaningful in the context of a document layout, for instance <section>, <content>, <header>, <footer> and <aside>. This makes the layout of a web page standard and transparent in a general way. For instance, the header of a document goes to <header> and the content to <content>. HTML5 lets developers to register their own tag which can be meaningful to them. [19.]

4.1.2 Audio and Video Tag

For a long time, embedding a multimedia like video and audio in a website required third party plugins. With the introduction of HTML5 tags like <audio> and <video>, the task has been really easy as it does not require any external support. In video, MP4 files with H.264 codec and AAC audio codec are supported in majority of the browsers whereas webm and ogg format are supported partially. [20.] Similarly, in audio MP3 is supported in most browsers whereas ogg and wav formats are partially supported [21].

4.1.3 Forms in HTML5

The forms features in HTML5 provide better user experience by making forms more consistent across different websites and giving immediate validation feedback to the user about data entry. This is possible even in browsers which have JavaScript disabled in them. [22.]

4.1.4 New APIs

Some of the interesting APIs that came along with HTML5 are LocalStorage, GeoLocation, Drag and Drop, Web Workers and Server Sent Events. With these new APIs, it is now possible to achieve features which was impossible or required a lot of effort in the previous version of HTML. For instance with Web Workers, it is now possible to run huge processes in the background as if the process was being run in a different thread. [23.]

4.2 CSS3 and SASS

CSS stands for Cascade Style Sheets. In web applications, HTML adds content, JS adds interactivity and CSS is responsible for adding style and look to the content. It is one of the main components of front end programming. CSS3 is the new version of CSS. It includes several new features which the previous version lacked. Some of the main features are rounded-corners, shadows, gradients, transitions and animations and the new layouts like multi-column and flex layout are some of the main features. [24.] For instance, with animation it is now possible to build complex animations using only CSS. Not every browser supports all the new features in a standard form. For example, the animation feature is used in browsers with the browser specific vendor prefix because the standard animation property may not be supported by all browsers. The vendor prefixes are “-webkit-“ in Chrome and Safari, “-moz-“ in Firefox, “-ms-“ in Internet Explorer and “-o-“ in Opera.

SASS stands for Syntactically Awesome Style Sheet. SASS is a CSS pre-processor which provides designers with an easier way of writing CSS syntax. It provides programming language like features for CSS. For instance, it enables designers to use variables which can replace the repeated values in a style sheet. Features like

nested selectors, loops, conditional statements and mixins are variables which are to be noticed. [25.] For example listings 6 and 7 show how nested selectors are implemented in SASS and plain CSS respectively.

```
body {  
  color: black;  
  
  nav {  
    color: blue;  
  }  
}
```

Listing 6. Nested selectors in SASS

```
body{  
  color: black;  
}  
  
body nav {  
  color: blue;  
}
```

Listing 7. Nested selectors in plain CSS

As seen in listing 6, the color property of the body is set to black. For every nav element inside the body, the color property is overridden with blue. Similarly, listing 7 shows how a similar styling can be obtained with plain CSS. It is to be noted that listing 6 is compiled by SASS into plain CSS as in listing 7.

Likewise, mixin in SASS allows designers to write a reusable block of CSS statements. It is like a function in a programming language as it is possible to pass in parameters as well. This feature could be really handy for cases where, for cross browser compatibility issues, the same CSS property needs different vendor prefixed property definitions. Listing 8 below shows an example of what a basic mixin looks like in SASS.


```
@mixin border-radius ( $radius ){
  -webkit-border-radius : $radius;
  -moz-border-radius : $radius;
  -ms-border-radius : $radius;
  border-radius : $radius
}
body{
  @include border-radius(5px);
}
```

Listing 8. Mixin in SASS

The mixin seen in listing 8 can be used to have a cross-browser compatible border-radius in an element without having to write all vendor prefixed properties. It is applied to the body element as shown in the listing above.

Another interesting and very useful feature of SASS is inheritance. With this, CSS styles for an element can be inherited by another selector. The properties in the parent selector are overridden by the properties in the child selector. Inheritance in SASS is briefly explained by listing 9 below.

```
.message{
  border: 1px solid black;
  padding: 10px;
  color: white;
}
.error{
  @extend .message;
  color:red;
}
.success{
  @extend .message;
  color:green;
}
```

Listing 9. Inheritance in SASS

Listing 9 shows how inheritance is implemented in SASS. As shown, the normal message class has some properties defined. After that the error selector extends message styles. That way the error class selector inherits all the properties but color from the message class. Similarly with the success selector, the color is overridden with green.

4.3 EmberJS

EmberJS is a JavaScript framework for building an ambitious single page client-side application. It embraces the Model-View-Controller (MVC) software architectural pattern. These different layers bind together to complete the application flow. For these bindings to work properly, the framework has enforced naming conventions for each related layers. For instance, FooTemplate would connect to FooController, FooModel and FooRoute. Of course there are ways to override the conventions. However overriding conventions can result in a messier and more complex code base. In other words, following the naming convention is a better option to go smooth with the development. [26.]

Before one can start development with EmberJS, there are some core concepts which are very essential to get acquainted with. These concepts are briefly explained in the following section.

4.3.1 Base Object Blueprint

It can be noticed that standard JavaScript class patterns and the new ES2015 classes aren't widely used in Ember. Plain objects can still be found, and sometimes they are referred to as hashes [27]. In order to make objects support the core features of Ember like binding, EmberJS has its own object blueprint. It is available as Ember.Object. It supports features like class inheritance, proper constructor method and extension with mixins. At times, there are needs in a class to define a property where the value of the property depends on values of one or more of other properties. In such a case, the base object provides a way to achieve the feature easily by defining a computed property. Computed properties update themselves whenever any of the depending property changes. Ember Object also provides a way to define a call back action when the value of a property changes by defining a property observer. [27.]

4.3.2 Model

In Ember, Model is an extension of Ember.Object which represents the underlying data that the application makes use of. Typically in a real application, the data is retrieved from the backend API and is saved similarly by sending the data to the backend. The layer which is responsible for retrieving and sending data from and to the server has to be coded before it could be implemented. However, Ember has made it really easy for developers as it provides a library called Ember data. Ember data gives all the interfaces needed for the communication with the server. If there is a need of a different way of handling model data than the default way, it is possible to write own adapter for it. Similarly, Ember Model handles the Model relationships well. Some of the relationships that are supported are one-to-one, one-to-many, many-to-many and polymorphic. [28.] Listing 10 shows a basic example of a Model in EmberJS.

```
export DS.Model.extend({
  firstName: attr('string'),
  lastName: attr('string'),
  birthday: attr('date')
})
```

Listing 10. Model in EmberJS

As seen in listing 10, the model in EmberJS is inherited from the base Model class available as DS.Model. The model has attributes like firstName, lastName and birthday.

4.3.3 View

View is the part of an application which the end user actually sees. It renders the data provided by specific controllers. Ember makes use of the Handlebars library for templates. It gives a way of rendering static as well as dynamic contents. Dynamic contents defined in curly braces in the template are rendered with data binding. Data binding updates the view whenever the underlying data changes. Similarly, it provides many template helpers like if, for loop, input and link-to which are very useful in a usual way. With these helpers, it is possible to evaluate conditions and use loops in the template itself. If the default helpers are not enough in some cases, then it is also

possible to write custom helpers. [29.] A basic example of Ember view is illustrated in listing 11.

```

{{for-each post in posts}}
  <h1>{{post.title}}</h1>
  <p>{{post.content}}</p>
{{/for-each}}

```

Listing 11. View in EmberJS

As seen in listing 11, posts array is being iterated to display the title and content of each post. This is how the contents of an array are iterated in a handlebar template.

4.3.4 Controller

Controller is a layer in an Ember application which lies in between the routes and the view. It is responsible for providing data to the view templates. The properties exposed from a controller are available in the template. It is the place where event listeners are defined which can be used in the template to handle different kinds of user interactions. It extends Ember.Controller which is the only routable component in EmberJS. In future Ember applications, it is expected to be replaced with components. [30.]

Based on the type of data returned from the respective route, the controller can be inherited from two different base classes which are Ember.ObjectController and Ember.ArrayController. If the data is a single object, it has to be inherited from Ember.ObjectController. If the data is a collection of objects, it has to be inherited from Ember.ArrayController. Listing 12 shows a basic controller extended from Ember.ArrayController

```

export Ember.ArrayController.extend({
  actions:{
    onFormSubmit : function(){
      alert('form submitted');
    }
  }
});

```

```
}}
```

Listing 12. Array Controller in EmbjerJS

It can be seen in the listing above that an action called `onFormSubmit` is defined in the controller, which can be referred to in the View template. It could be used for instance to handle submission of forms.

4.3.5 Route

The router is responsible for displaying templates, loading data, and otherwise setting up application state. It does so by matching the current URL to the defined routes . [31.] The routes are defined by reopening the `App.Router` class as shown in listing 13.

```
export Router.map(
  function() {
    this.route('about', { path: '/about' });
    this.route('favorites', { path: '/favs' });
  })
```

Listing 13. Router class in EmberJS

The listing above shows how the URL `'/about'` is mapped to `AboutRoute` and `'/favs'` to `FavouritesRoute`.

Similarly, listing 12 shows what a basic route in EmberJS looks like.

```
export Ember.Route.extend(
{
  model: function(){
    return this.store.findAll('AboutModel');
  }
} )
```

Listing 14. Route class in EmberJS

As seen in listing 14, a function called Model is defined. This method is trivial as it is the place where the Model is loaded that is meant to be passed to the view. The data returned from this method is then assigned to the Model property of the respective Controller, which is then accessible for the use in View template. Likewise, there are other call back hooks that can be defined in an Ember route. They carry their own importance. Some of the hooks which are available are setupController, before-Model and afterModel.

For instance, setupController is run after the Model hook. It contains the value returned from the model hook as a parameter. Therefore, it can be used to manipulate the model before it is assigned to its controller.

4.3.6 Component

In EmberJS, the component is a subclass of Ember.Component. It is basically a View that is completely isolated from the application with its own life cycle. It always consists of a template and a view object where all of its properties and event listeners are defined. It is an injectable part of the application and is associated with an HTML tag. There is no access to the surrounding context in a component. If needed the context has to be passed as a parameter to it. This should be one of the best features of EmberJS as it can be injected and reused across EmberJS applications. [32.] Listing 15 shows a basic example of a component in EmberJS.

```
<script type="text/x-handlebars" id="components/blog-post">
  <h1>Blog Post</h1>
  <p>Lorem ipsum dolor sit amet.</p>
</script>
```

Listing 15. Component in EmberJS

Listing 15 shows a basic component with just a view template. This component can be reused wherever that particular HTML is required. However, this component is lacking its logical layer. The component can be more complex than this. It can have its own scoped controller class that contains the logics.

4.3.7 Testing

EmberJS encourages test driven development. It is designed in a way that makes every possible part of an application to be testable. It uses QUnit as the default testing library. It is also possible to use other libraries, but that would have to be done with third party addon libraries. [33.] Listing 16 illustrates an integration test in EmberJS.

```
module('Integration: Post page, {
  teardown: function() {
    App.reset();
  }
})
test('add new post', function() {
  visit('/posts/new');
  fillIn('input.title', 'My new post');
  click('button.submit');
  andThen(function() {
    equal(find('ul.posts li:last').text(), 'My new post');
  });
});
```

Listing 16. Integration test in EmberJS

Listing 16 illustrates an integration test for testing transitions. It starts by defining a module. The module explains the scope of the test. The blocks that are defined afterwards with starting with the key “test” represents the test case. For instance the test in the listing above is testing a feature about adding a new post. Basically this test is simulating what an actual user needs to do in order to add a new post. And the test helpers like visit, fillIn and click provided by Ember help simulate the case.

4.4 EmberCLI

EmberJS eases the development of conventional web applications. However, one of the drawbacks of EmberJS that developers using other frameworks like AngularJS point out is the way the application codes are loaded. EmberJS loads everything in the global namespace which is not considered as a good practice in coding. It loads all the codes at the time of initialization, even if a specific part of the code is not being used at that moment.

EmberCLI which stands for Ember Command Line Interface, is a command line utility for EmberJS. It provides a standard project structure and a new set of development tools. It allows Ember developers to focus on building apps rather than building the supporting environment for the application. To address the issue with the loading, it gives a way to write modular code which is injected only if it is used. For this it uses import API which is a feature of the new version of JavaScript. Similarly, another important addition is Addon. Addon is a miniature ember application which can be integrated into other Ember applications. It has widened the scope of EmberJS, as it is now possible to share and integrate any addon coded by other developers. [34.]

5 The Project

The final project was about building a Single Page Application for the Heli service. Heli is a service that is dedicated to people who wants to pursue a healthy life. Heli provides tools to achieve a better life. For this final project, EmberJS was used as a SPA framework and Rails was used as a backend API.

5.1 Framework Selection

With the decision of building a single page client application for Heli, there came a decisive point which was very to deal with. That decision was to select a framework on which the Heli single page application would be built. At the time of carrying out the project, Single Page Applications were popular. Because of this, there were plenty of SPA frameworks to choose from. They were all good in their own way. However, if a framework is selected which best meets the requirements of the application, the rest of the development process and implementing possible future extensions will become a lot easier. Therefore, choosing a SPA framework for Heli SPA was indeed a difficult but very important decision to make.

Based on the popularity of the frameworks, AngularJs and EmberJS were two options. Both of these frameworks had similar ideas, with obviously different ways of implementation. However, ultimately EmberJS was chosen for the reasons described below.

- Resemblance with Rails

The first thing that can be noticed is the project structure. EmberJs has similar components to Rails like Model, Controller, Views and Routes. Therefore the default project structure looks quite similar.

Similarly, EmberJS tries not to make developers worry much about project configurations but focus on application development. This means that when an EmberJS project is created, the application is already working with a basic structure containing one landing page. At the time of creation, it already defines the application index route with all required components for it.

Rails has a feature called Asset Pipeline. At the time of deployment, it is better to have as few assets like JavaScript, CSS files as possible. RoR's Asset Pipeline is responsible for merging all the scripts and CSS files to a respective single file. EmberJS also has this kind of feature, but it is achieved with a JavaScript library called BroccoliJS.

- Usefulness of Ember Data

Ember data tightly integrates with EmberJS to make it really easy to retrieve records from a server, cache them for performance, save updates on the server and create new records on the client. Ember data can load, save data and relationships without any configuration, provided the data returned from API follow a definite convention. In case it is not possible to access the API for modifications, Ember data allows developers to make changes in the client side to adapt with the provided data. The adapter can be customized to meet the requirements. And for an application using the backend of Rails there is already a contributed adapter called active-model-adapter available, which is the case with Heli.

- Introduction of EmberCLI

EmberCLI has made development with EmberJS more effective with all the tools it provides. EmberCLI provides so many useful shell commands that are frequently used in the process of development like generating modules, building, deploying and running tests. It encourages modular coding with possibility to import modules asynchronously. Similarly, it adds a testing environment configured by default when a project is created. This makes testing really easy and allows developers to write tests straight away without having to configure the testing environment, which would actually be a significant amount of work. Apart from this, EmberCLI also includes many very useful test helpers, which were missing from the EmberJS project.

Ultimately, EmberJS was chosen as the framework for the SPA version of Heli. However, it cannot be neglected that AngularJS is as good as EmberJS in its own way. All of the goodness of EmberJS mentioned above are available in AngularJS as well. The only difference is that EmberJS provides all of these by default. Hence, it is easier to get started with the development without worrying much about configuring the project.

5.2 Challenges

As mentioned already, at the time the development of Heli SPA started, Heli was already an active service. However, it was built in the conventional way where the pages were rendered already in the backend and responded as an HTML page. However for a SPA, the application should be able to respond with JSON data. So changing the way the existing application responds to the request from the clients without hampering the former behaviour of the application was a challenge. Listing 17 shows what a conventional Rails Controller looks like.

```
PageController < ApplicationController
  def index
    @pages = Page.all
  end
end
```

Listing 17. Rails controller

As seen in listing 17, the PageController contains one action called index. This basically means that there is only one page that lists all page models available. And that page is available with URL /pages.

With Rails, it seems quite straightforward to add a response type for a request to a resource. And it is illustrated in listing 18.

```
PageController < ApplicationController
  respond_to :html, :json
  def index
    @pages = Page.all
    responds_with @pages
  end
end
```

Listing 18. A RoR controller with multiple response type

Listing 18 shows how a controller can be updated to make it respond with other than HTML. For instance, the listing shows how the Controller is able to respond with either HTML or JSON content. Two URL endpoints are available for this Controller as /pages.html and /pages.json returning the response in HTML and JSON format respectively.

However, the new changes to be brought to the existing application should not hamper its current behaviour in any way. So, it was indeed an important decision to make if the very existing controller should be modified to respond to JSON API requests. Alternatively, another possible solution for this change could have been to define new sets of dedicated routes and Controllers which would only serve JSON requests. Ultimately, the second option seemed more viable for the project. That way, the old logics needed not to be touched at all. The application would have a different logical path for the requests made for JSON data. In addition, routes and Controllers for the JSON API could have a different namespace. For instance, JSON API were namespaced under `API::V1` which made them available with URLs prefixed with `“api/v1/”`. Hence, the first version of Heli API was created with a `“api/v1”` as a prefix.

Similarly, one of the problems that came along was the limitation in making cross site requests from the front end. Since in development phase, the application is hosted in localhost, it was impossible for the application to make a request to the real API. The reason was that the browsers prevent cross-domain requests for security reasons. It was still possible to do the tests by providing the dummy data rather than making real HTTP requests. However, along the way it could be significant to test the application with the real backend API. Therefore, there was a need to find a way to avoid this limitation.

The W3C Web Applications Working Group recommends the new Cross-Origin Resource Sharing (CORS) mechanism. CORS gives web servers cross-domain access controls, which enable secure cross-domain data transfers. Modern browsers use CORS in an API container such as XMLHttpRequest to mitigate risks of cross-origin HTTP requests. [35.] CORS stands for Cross Origin Resource Sharing. In order to enable CORS, some extra headers had to be added to the pre-flight response sent from the API. For instance, Access-Control-Allow-Origin was to be present indicating the list of domains which are allowed to make the request. In addition, headers like Access-Control-Allow-Headers, Access-Control-Allow-Methods were included to specify the permitted headers and methods for the requests.

Fortunately with Heli API, it was easy to add CORS through an existing gem named rack-cors.

```

use Rack::Cors do
  allow do
    origins 'localhost:3000', '127.0.0.1:3000',
    resource '/file/at/*',
      :methods => [:get, :post, :delete, :put, :patch,
:options, :head],
      :headers => 'X-Auth-Token',
      :max_age => 600
  end
end

```

Listing 19. Configuring rack-cors gem

Listing 19. shows a chunk of CORS configuration codes added for rack-cors gem. The origins localhost:3000 and 127.0.0.1:3000 are allowed to make requests to the API, as these are the domains which would be used for development. And almost all the HTTP methods are permitted. In addition, a custom header called X-Auth-Token can be sent in headers which is used for authorisation.

Likewise, for the success of project along the long run, it was really essential to have test coverage for codes. For that, test driven development was followed as much as possible. Therefore, the codes in Heli SPA are tested.

5.3 Communication with Rails Backend

The Ember model is linked to the backend API with adapters. An adapter is the connecting channel between the Model and the actual source of data. The source of data does not need to be essentially a backend server. It can be any source of data that implements the adapter interface. For instance, there is a contributed EmberJS adapter called ember-localstorage-adapter which makes use of the browser's local storage for persisting data. There are a number of ready-to-use adapters available such as DS.RESTAdapter, DS.ActiveModelAdapter and DS.FixtureAdapter.

It is to be noted that separate adapters can be assigned to each of the Models. There could be a situation when an application might be using more than one source of data. In that case, EmberJS makes it really easy to implement multiple data source for an application by providing a way to assign a Model specific adapter. The adapters can be assigned as shown in listing 20 below.

```
ApplicationAdapter = DS.RestAdapter.extend({  
  host : 'https://api.example.com'  
})  
  
PostAdapter = DS.RestAdapter.extend({  
  host: 'https://post-api.example.com'  
})
```

Listing 20. Model specific adapter in EmberJS

As illustrated in the listing above, the entire application uses an adapter which is available on <https://api.example.com>. The Post model is using a different endpoint as <https://post-api.example.com>.

Similarly, it is essential to understand that there is another layer in the background which is playing an important role. The layer is between an adapter and a Model. It is a part of the adapter itself. It is called Serializer. When the data is received from an adapter it has to be converted to the way the Model is represented. Basically, it is responsible for converting plain JSON data that is delivered from the adapter to the respective Model and vice versa. It is also to be noted that the Serializer is customisable to one's requirement.

As Heli SPA was using the Rails backend for the API, the most suitable adapter available was `DS.ActiveModelAdapter`. It is built especially to be used with Rails API. However, using Rails only did not suffice the requirements for using this adapter. There were some conventions the API needed to follow.

The `ActiveModelAdapter` is a subclass of the REST Adapter designed to integrate with a JSON API that uses an underscored naming convention instead of camel casing. It has been designed to work out of the box with the Ruby gem called `activemodelserializers`. [36.] The backend Rails API needs to use that gem, which basically helps to structure the model data into JSON format.

Installation of the gem is quite easy. For that the gem is to be added to the Gemfile in the project. After it is added to the Gemfile, the `bundle` command is run. When the command runs successfully, it is added to the project. Active Model Serializer needs an

extra class for every Model class that is intended to be sent to the adapter of the EmberJS Model.

```
Source 'https://rubygems.org'
Ruby '2.0.0'
Gem 'rails', '3.9.0'
gem 'bootstrap-sass', '2.3.2.0'
gem 'sprockets', '2.11.0'
gem 'bcrypt-ruby', '3.1.2'
gem 'active_model_serializers'
```

Listing 21. Project's Gemfile

Listing 21 illustrates a Gemfile which includes `active_model_serializers` in the project. In addition to that, a lot of other gems can be seen. These gems could be added by Rails by default in the core bundle or added later by developers when it was required.

Serializer is responsible for producing well formatted JSON data based on Model data. It is a part of the `active_model_serializers` gem. Listing 22 shows a serializer which serializes the post Model.

```
PostModelSerializer < ActiveRecord::Serializer
  attributes :id, :name
  embed :ids, include: true
  has_many :comments
end
```

Listing 22. Serializer for the Post model

As shown in listing 22, attributes like `id` and `name` are included in the JSON data. Apart from this, the `embed` and `has_many` part define that the JSON data should include `id` of each comments which belong to the post. For instance the JSON response from the above serializer may look like the code in listing 23.

```
{
  post:
  {
    id:1,
```

```
    name: 'Frist post',
    comment_ids:[10, 20]
  },
  comments:[
    {
      id: 10,
      content: 'this is great post'
    },
    {
      id: 20,
      content: 'This not good'
    }
  ]
}
```

Listing 23. Sample JSON response

One thing that should be noticed is `comment_ids` part in post JSON data. This comes as a result of the `embed` part in the serializer. The significance of `embed` part is that it allows the Ember Model to side load related comments in the same request. This way the application does not need to make separate backend requests later to retrieve comments for that post.

5.4 Authentication

Authentication is the verification of the credentials of the connection attempt. This process consists of sending the credentials from the remote access client to the remote access server in an either plaintext or encrypted form by using an authentication protocol. [37.]

Heli collects user's information that can be very vital and private to users. So it is really important that the user information is protected from being leaked out. Therefore, the strongest part of a service such as Heli should be authentication. The authentication system for the Heli backend was already in place before SPA for Heli was started, as Heli was already a running service. Heli was using cookie-based authentication. That means that when a user is authenticated, the session is saved in the cookie. Whenever

the user makes another request, the respective session id is sent along the request and the backend knows that the user is an authenticated user.

With Heli ember, cookie-based authentication was not the best solution as using cookies in asynchronous cross domain requests are not considered safe. Therefore, the backend authentication system needed to be customized to support the new flow. According to the new flow, when a user tries to visit a protected page for the first time the application looks for an authentication token saved in session storage. Obviously, it does not find it since it is the first use visit. Therefore the user is redirected to the login page. The user fills in credentials and sends a login request. The backend authenticates the user based on credentials provided. If the user is not authenticated for some reason, the backend will respond with response code 403. However, if the user is authenticated, it sends back an authentication token. Back in the Ember application, the authentication token is saved in the browser's session storage. Afterwards, for each request to the backend API, the token is injected in the header. And the backend authenticates the user based on the token.

An Ember mixin called `Authenticated` was created. This mixin contained all interfaces about the token check, response code check and handle token sync between the application and session storage. If a route was to be made protected with authentication, the route was extended with this mixin.

```
export default Ember.Route.extend(Authenticated, {
  _permittedUsers: ["patient"],
  model: function() {
    return this.store.find('health-
goal', {patient_id: this.get("current_session.user.userlinkable_id
")});
  }
})
```

Listing 24. Sample protected route in Heli Ember App

Similarly, a class called `CurrentSession` is injected in every controllers and routes that is meant to carry user session information. When a user logs in, it is populated with the user information including a session token id.

5.5 Authorization

Authorization is the verification that the connection attempt is allowed. Authorization occurs after successful authentication. [37.] Authorization is the phase after authentication. When a user succeeds with authentication, the resources are protected on the basis of role of the user. For instance, not every user is allowed to access certain pages, or user may have restrictions on certain actions related to that resource.

Implementation of authorization in Heli ember includes a frontend as well as backend check. Even if a hacker manages to pass by the frontend authorization check, it will fail in the backend. Frontend authorization is implemented by authenticated mixin again. Authenticated mixin includes a property that carries list of authorised user roles. If an unauthorized user tries to access the page, the user is prevented from landing on that page.

```
export default Ember.Route.extend(Authenticated, {
  _permittedUsers : ["patient", "caregiver"]
})
```

Listing 25. Sample route authorised only to patient and caregiver.

By default, the route is permitted for each type of users, who are defined in authenticated mixin. If the route is to be limited to patient and caregiver only, the `_permittedUsers` is defined as shown in the listing above. This way, before making an API request to backend the application checks if the user is permitted on that page and cancels the page transition in case of a failure. Similarly in the backend, if the authorization check fails the API call is responded with the 401 status code and the frontend handles the failure accordingly.

6 Conclusion

The main goal of this final year project was to build a Single Page Application for the existing Heli service. Heli is a service that is targeted to people who want to pursue a healthy life. Heli provides tools to achieve a healthier life. The service exists as a multipage application. There were plenty of limitations with the multipage version of Heli. Therefore, to avoid limitations like limited platform support and user experience, an SPA for the Heli service was built using EmberJS as the framework.

Considering the progress of the SPA Heli project so far, Heli has a working single page application developed with EmberJS. However, not all services of Heli have been implemented. The Heli SPA is not in a state to be deployed yet. The base of the application is in place including authentication and authorization. In addition, a couple of features including goals and health plans have been implemented. Also, the codes are properly tested as the Test Driven Development approach was followed.

After the completion of the project, Heli SPA is supposed to replace the multipage version of the Heli application. As it is designed to adapt to different screen sizes, the users are expected to have a richer user experience on all kind of devices. On mobile devices, the service is expected to have a native-application-like interface. In the future, platforms like Phonegap can be used to build native applications for mobile platforms.

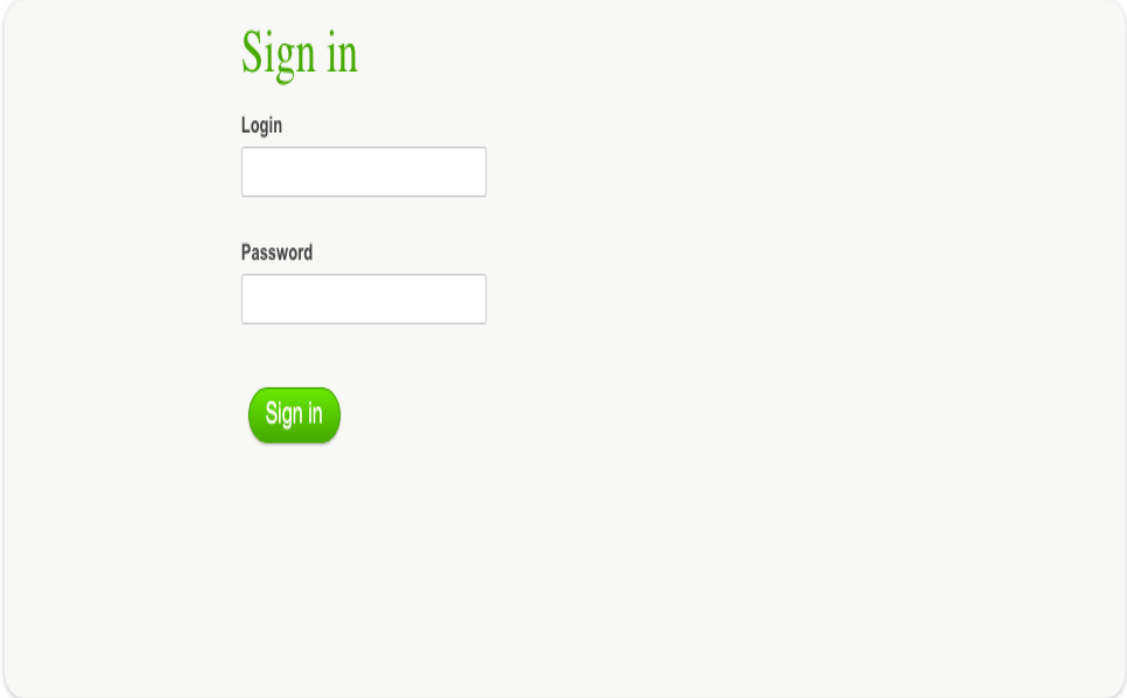
References

1. Health-e-living [online]. Finland: Extensive Life Oy.
URL: <http://www.extensivelife.com/index.php?page=heli>. Accessed 9 January 2016.
2. Sluyters R. Introduction to the Internet and World Wide Web [online]. The United States: Oxford University Press.
URL: <http://ilarjournal.oxfordjournals.org/content/38/4/162.full>. Accessed 5 March 2016.
3. Takada M. Single Page Apps in Depth [online].
URL: <http://singlepageappbook.com/goal.html>. Accessed 4 March 2015.
4. Shoemaker C. HTML5 History: Clean URLs for Deep-linking Ajax Applications [online]. The United States: EPS Software Corp.
URL: <http://www.codemag.com/article/1301091>. Accessed 10 March 2015.
5. Wasson M. ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET [online]. The United States: Microsoft; November 2013. URL: <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>. Accessed 4 March 2015.
6. Shimanovsky S. Multi Page Web Applications vs. Single Page Web Applications [online]. New York: Eikos Partners; 9 July 2015.
URL: <http://www.eikospartners.com/blog/multi-page-web-applications-vs.-single-page-web-applications>. Accessed 10 August 2015.
7. Podila P. Important Considerations When Building Single Page Web Apps [online]. 21 January 2013.
URL: <http://code.tutsplus.com/tutorials/important-considerations-when-building-single-page-web-apps--net-29356>. Accessed 10 August 2015.
8. Ruby on Rails. Rails [online].
URL: <http://rubyonrails.org/>. Accessed 25 August 2015.
9. Ruby on Rails. Getting Started with Rails [online].
URL: http://guides.rubyonrails.org/getting_started.html. Accessed 25 August 2015.
10. Chrome. MVC Architecture [online].
URL: https://developer.chrome.com/apps/app_frameworks. Accessed 25 August 2015.
11. Ruby on Rails. Active Record Basics [online].

- URL: http://guides.rubyonrails.org/active_record_basics.html. Accessed 29 August 2015.
12. Ruby on Rails. Action Controller Overview [online].
URL: http://guides.rubyonrails.org/active_record_basics.html. Accessed 29 August 2015.
 13. Ruby on Rails. Rails Routing from the Outside [online].
URL: <http://guides.rubyonrails.org/routing.html>. Accessed 29 August 2015.
 14. Ruby on Rails. Action View Overview [online].
URL: http://guides.rubyonrails.org/action_view_overview.html. Accessed 30 August 2015.
 15. Ruby on Rails. Guide to Testing Rails App [online].
URL: <http://guides.rubyonrails.org/testing.html>. Accessed 30 August 2015.
 16. Ruby S, Thomas D, Hansson DH. Agile web development with rails. 3rd ed. Texas: The Pragmatic Bookshelf; 2009.
 17. Ruby on Rails. The Asset Pipeline [online].
URL: http://guides.rubyonrails.org/asset_pipeline.html. Accessed 10 September 2015.
 18. Elkstein M. Learn REST: A Tutorial [online].
URL: <http://rest.elkstein.org/>. Accessed 8 April 2015.
 19. W3Schools. HTML5 Semantic Elements [online].
URL: http://www.w3schools.com/html/html5_semantic_elements.asp. Accessed 28 March 2015.
 20. W3Schools. HTML5 Video [online].
URL: http://www.w3schools.com/html/html5_video.asp. Accessed 28 March 2015.
 21. W3Schools. HTML5 Audio [online].
URL: http://www.w3schools.com/html/html5_audio.asp. Accessed 28 March 2015.
 22. Mozilla Developer Network. Forms in HTML [online].
URL: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms_in_HTML. Accessed 28 March 2015.
 23. Mozilla Developer Network. HTML5 [online].
URL: <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>. Accessed 28 March 2015.
 24. Mozilla Developer Network. CSS3 [online].
URL: <https://developer.mozilla.org/en/docs/Web/CSS/CSS3>. Accessed 29 March 2015.
 25. Sass. Sass Basics [online].
URL: <http://sass-lang.com/guide>. Accessed 29 March 2015.

26. Ember. CORE CONCEPTS [online].
URL: <https://guides.emberjs.com/v1.10.0/concepts/core-concepts/>. Accessed 10 November 2015.
27. Ember. The Object Model [online].
URL: <https://guides.emberjs.com/v2.0.0/object-model/classes-and-instances/>. Accessed 9 January 2016.
28. Ember. Introduction [online].
URL: <https://guides.emberjs.com/v2.0.0/models/>. Accessed 9 January 2016.
29. Ember. Handlebars Basics [online].
URL: <https://guides.emberjs.com/v2.0.0/templates/handlebars-basics/>. Accessed 9 January 2016.
30. Ember. Introduction [online].
URL: <https://guides.emberjs.com/v2.0.0/controllers/>. Accessed 25 January 2016.
31. Ember. Defining Your Routes [online].
URL: <https://guides.emberjs.com/v2.0.0/routing/defining-your-routes/>. Accessed 25 January 2016.
32. Ember. Defining A Component [online].
URL: <https://guides.emberjs.com/v2.0.0/components/defining-a-component/>. Accessed 25 January 2016.
33. Ember. Introduction [online].
URL: <https://guides.emberjs.com/v2.0.0/testing/>. Accessed 25 January 2016.
34. Ember CLI [online].
URL: <http://ember-cli.com/user-guide/>. Accessed 17 February 2016.
35. Mozilla Developer Network. HTTP access control (CORS) [online].
URL: https://developer.mozilla.org/enUS/docs/Web/HTTP/Access_control_CORS. Accessed 19 January 2016.
36. Ember. DS.ActiveModelAdapter Class [online].
URL: <http://ember-doc.com/classes/DS.ActiveModelAdapter.html>. Accessed 17 February 2016.
37. Microsoft. Authentication vs. Authorization [online].
URL: [https://technet.microsoft.com/en-us/library/ff687657\(v=ws.10\).aspx](https://technet.microsoft.com/en-us/library/ff687657(v=ws.10).aspx). Accessed 27 February 2016.

Appendix 1: Login Page



Sign in

Login

Password

Sign in

Appendix 2: Admin's Health Goal Templates Page

Health Goal Templates

FirstTemplate	 	Unit	Achievable Type	Actions
SecondTemplate	 		ExerciseClass	 
Third Template	 		PracticeClass	 
Fourth Template	 			
Fifth Template	 			
Sixth Template	 			





Appendix 3: Patient's Health Goal Management Page

The screenshot shows a web application interface for managing health goals. At the top, there is a green navigation bar with the following menu items: Goals, Plan, Diary, Progress, Messages, and Info Blocks. On the right side of this bar, there is a 'Patient Test' button and a circular icon with a right-pointing arrow.

The main content area is titled 'Goals' and is divided into two columns:

- Current Goals:** This section contains a 'Show All' checkbox and a list of goals. One goal, 'SecondTemplate', is visible with a trash icon to its right.
- Addable Templates:** This section contains a list of templates: 'FirstTemplate', 'Third Template', 'Fourth Template', 'Fifth Template', and 'Sixth Template'. Each template has a green plus icon to its right, indicating it can be added.

At the bottom left of the main content area, there is a green 'Save Changes' button.

Appendix 4: Integration Test Code

```
untitled accordion-panel.js template.hbs health-goal-templates-test.js
1 import Ember from 'ember';
2 import {test,module,skip} from 'qunit';
3 import startApp from '../helpers/start-app';
4
5 var App;
6
7 module("Integration - Health Goal Templates Page",{
8   beforeEach:function(){
9     App = startApp();
10  },
11  afterEach:function(){
12    Ember.run(App,App.destroy);
13  }
14 });
15
16 test("unauthenticated users cannot access the page",function(assert){
17   visit("/health-goal-templates");
18   andThen(function(){
19     assert.notEqual(currentRouteName(),"health-goal-templates");
20     assert.notEqual(currentURL(),"/health-goal-templates");
21     assert.equal(currentRouteName(),"login");
22     assert.equal(currentURL(),"/login");
23   });
24 });
25
26 test("patient should not be able to access the page",function(assert){
27   loginAsPatient();
28   andThen(function(){
29     visit("/health-goal-templates");
30   });
31   andThen(function(){
32     assert.notEqual(currentRouteName(),"health-goal-templates.index");
33   });
34 });
35
File 0 Project 0 ✓ No Issues tests/integration/health-goal-templates-test.js* 36:1 LF UTF-8
```