Designing Efficient Single Page Web Applications

by

Nyruthya Sanandan

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 15, 2018

Keywords – single page applications, design approach,
web application development

Abstract

Traditional web applications rely on the client-server model to request and render data on the browser. Although this architecture has been popular for over a decade, waiting for information as the server caters to different types of clients makes the user experience appear to be slow. Client-side machines have computing capabilities, that when used can share the computing load with the servers. Contemporary single page applications take advantage of these client-side machines to minimize the delay by providing client-end software with the capabilities of requesting only required data from the server and updating only portions of the presented information, without constant redundant data being on the network.

This study provides an initial design approach for contemporary applications. The proposed design approach helps developers to analyze all the aspects of the application that needs to be built – architecture, file structure, separation of code, and distribution of computing between client and servers. This study also provides a step–by-step method for designing contemporary applications. An application was built using the proposed method to validate the proposed design approach.

Acknowledgments

I take this opportunity to thank all those who helped me through the duration of this thesis. I would like to especially thank my advisory committee chair, Dr. David A. Umphress, Computer Science and Software Engineering department, for the support he has provided me with from the beginning of this research. His office, lab resources and ideas were always open to me when I needed them the most. He made sure I had everything I needed and guided me when I was in trouble.

I would like to thank my friends and colleagues who brainstormed with me and exchanged knowledge, thereby making the process intuitive.

Lastly, I would like to thank my family for their continuous encouragement throughout my years of coursework. This accomplishment would not be possible without them.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
JS	JavaScript

Problem Description

1.1 Introduction

The web browser has emerged as a general-purpose computing platform through which application functionality is delivered over a network. Driving this trend is technology that shifts the focus of work in a client-server architecture from relying on the server to deliver the computation results that are rendered by the client to relying on the server to deliver executable instructions that are carried out by the client. The trend has been disruptive to the software industry in terms of how functionality is provisioned to the user, where computation takes place, and how data is managed. While technical mechanisms are in place for leveraging browsers as a general-purpose platform, sound techniques for engineering software around those mechanisms has lagged. Missing, in particular, are guidelines for designing software that rely on cutting-edge web-based architectures.

1.2 Traditional and Contemporary Web Applications

Traditional web applications rely on a client-server relationship in which software running on a client renders in a human-readable format the result of computations performed by and sent from software running on a server. The client treats the server as messenger which delivers one HTML page at a time. The page contains content of interest to the user, together with instructions on how to render it and, optionally, computational instructions that can be carried out by the client. The interaction pattern is a repetition of the following: 1) the client requests information from a server, 2) the server replies with a response, and 3) the client displays the response, replacing the entirety of what was previously displayed. From the user perspective, the client becomes unresponsive from the time it makes a server request until the time the information is rendered. This causes a noticeable delay that can, depending on the speed of the network and processing capabilities of the server, range from distracting to debilitating.

Contemporary web browser technology attempts to minimize the delay by providing client software with the capability of 1) requesting information from the server without having to wait for a response, 2) updating a portion of information presented to the user without having to download and refresh the entire display, and 2) executing a predetermined, but safe and rich range of programming instructions accompanying the server response. In contrast to traditional web applications which rely on the delivery of separate, independent, and relatively-static HTML pages to accomplish tasks, contemporary web applications are delivered as software which runs on the client, interacting dynamically and asynchronously with the server to update selected portions of the information displayed to the user.

Because the majority of features are carried out by the server, traditional web application development has relied on classical software engineering design approaches which revolve around a centralized computer performing input and output. Contemporary web applications do not fit the classical model: computing is distributed among client and servers; the degree to which computing is distributed depends on the computational capabilities (i.e., hardware) of the client and the nature of the network connection to servers; the nature of computing allocated to the client hinges on the programmability of the client's software environment; and so forth. In short, contemporary web applications require an approach to design than do traditional web applications. But what is that approach?

1.3 Thesis Objectives

The goal of this thesis is to provide an initial design approach for contemporary web applications. The specific objectives are to

- Identify the properties of contemporary web application design that distinguish it from traditional web application design.
- Propose a semi-formal method for designing contemporary web applications.
- Validate the method with a sample application.
- Suggest areas of further work.

Review of Web Application Design

2.1 State-Of-The-Practice in Designing Web Applications

Traditional web development approaches do not take full advantage of the client-side computing capabilities. They only use them to render data received from a server into readable views. Today's browsers are run-time environments for applications that do not have to be installed on a device and can be accessed from anywhere. Traditional applications require the developer to create multiple pages of content, only one of which is visible to the user at any point in time. Every time a user requests more information, the client sends a request to the server, where it is processed. The client receives information and displays it on the screen. This happens every time a user visits a page that has not been cached, even if the new page only contains a few items that are different from the current page. When the server needs to send HTML files to the client instead of just data, it causes complex applications to wait for data and HTML files from the server each time a request is made and thus making the application slow.

2.2 Characteristics that Distinguish Contemporary Web Applications

Contemporary applications take advantage of existing client-side resources by distributing the computation across both the server and the client, thus reducing delays caused by the network and having the potential to offer a more natural user experience. Contemporary applications in the form of single page applications request one HTML page from the server and render the information in the client-side browser initially. During this request, the client also downloads JavaScript and CSS bundles from the server, which are then used to retrieve required data from the server as needed and render data into readable views on the client. The content is rendered on the client-side browser and JavaScript is used to load new content and make sure that only the new content is rendered. Everything else on the page remains unchanged. This is much faster as only small sections of the page are loaded instead of an entire page.

In addition to improving the user experience, single page applications are simpler to deploy because they have only one "index.html" file with HTML, one file with CSS, and one file with JavaScript. Thus, there are only three files to deploy. These three files can be uploaded to a static content server such as Apache or Amazon web storage to deploy the front end of the application. These files can be uploaded to different content servers too. When the client needs data from the server, the request can be processed by creating an interface using server-side code, such as PHP or NodeJS, that allows the server to respond to the client with requested data. This also makes versioning easy in a single page application when compared to a traditional application. Some servers scale for a large number of users and uploading the front-end to the static server ensures scalability of the application, in terms of the front end. Since the front-end and the back-end of the application are not tightly coupled, the back-end code can be reused to create native applications such as mobile platform, VR platform, etc.

Single page applications provide a spectrum of features that developers can use to make the most of both the client-side and server-side computing capabilities but, to do so, developers require more discipline than they would for a traditional application. Firstly, since single page applications use JavaScript code to retrieve and render data on the client-side, it is important to note that these applications require JavaScript to be supported and enabled on the client-end.

Another point to consider before creating a single page application is the feasibility of creating a non-traditional application based on the amount of data, the complexity of the application, and the service it will provide to the end users because a contemporary framework requires greater architectural and security expertise than a traditional application would. Since single page applications send and receive only parts of the data that are required at that point in time, content retrieval is much faster even with a rich UI. LinkedIn, for example, realized a 20% faster page load rate with a single page architecture when compared to a traditional architecture [Veeravalli, 2017]. If the application provides multiple services to its users, developers can divide these services into modules that are loaded individually so as to reduce the page load times.

Single page applications require special attention with respect to search engine optimization. When the content is readily available on the server in a traditional application, search engines can index and crawl the content without any extra effort from the developer. Single page application content is not available until the page is loaded on the browser, thus affecting search engine optimization. Although the majority of search engines do not handle client-side rendering, Google has been working to handle indexing and crawling JavaScript rendered code, which would enable developers to create single page applications without having any drawbacks. Google's search engine renders and indexes JavaScript using a two-wave process. The first wave requests source code and indexes any HTML and CSS that are present. The second wave, that can occur a few hours to a few weeks later, returns to the page and fully renders and indexes the JavaScript generated content [Burkholder, 2018].

Contemporary web applications may also seem more complex in terms of development. But, this is because traditional applications have been built using the client-server architecture since the development of web applications began.

One of the other reasons why contemporary applications are not as popular is that there are few sources, in the form of books or papers or articles online, for guiding developers in designing single page applications. Many sources explain how single page applications work, how they are different from traditional applications, the pros and cons of building a single page application, and how a framework can be used to build one, [Sherman, 2018] but developers may find it difficult to process this overwhelming data and use this variety of information to actually build a contemporary application that is easy to build and maintain. We could find no defined method specific to designing single page applications, to include - the UI, the content of the application, the service it provides to its users, the amount of data that will be on the network, the client-side computing capabilities, source-code management, maintainability of the software and the security needed to be implemented.

The idea of creating a contemporary application is to distribute computing between the client and the server such that the application can run on the client side with minimum support from the server. Developers need to be able to decide what parts of the application need to be computed on the client and what parts need to be computed on the server. To do this, we can present six steps that start with the application residing entirely on the client. We then distribute computing such that only components that cannot reside on the client are transferred to the server for computing.

3.1 Architecture

The goal of this step is to divide the application into components; to determine interactions among components, the dependencies between the components, the flow of data between the components, and the direction of data flow; and to, formulate a workflow of the application.

3.1.1 Components based on source of content

The content to be delivered to the users of the application must be determined. The content could be static or dynamic, its source could be from user inputs from an HTML file, or from a file that is locally stored on the machine in which the application exists, or it could be data from a remote location.

If the content is received from user inputs on an HTML file, the data storage and retrieval can reside on the client side. Similarly, if the source of the content is locally stored, the data storage and retrieval can reside on the client side. However, if the content of the application is stored on a remote location, the client can only retrieve the content by making a request to this remote location server, thus handling the data retrieval of the application on the client-side and handing the data storage to the server. This is depicted in Figure 3.1.

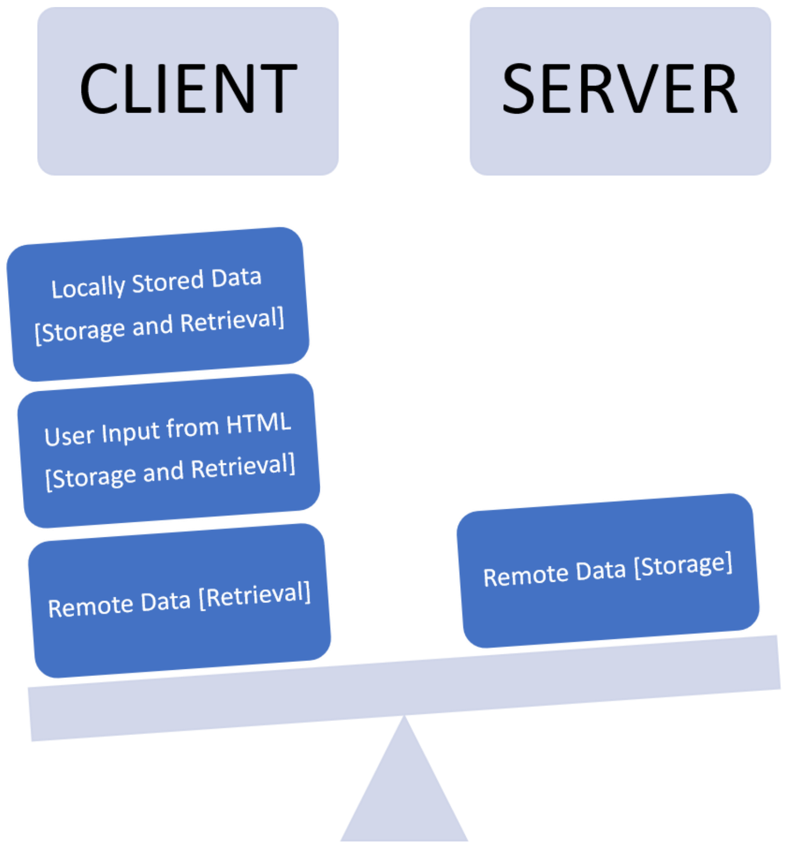

CLIENT
SERVER
Locally Stored Data [Storage and Retrieval]
User Input from HTML [Storage and Retrieval]
Remote Data [Retrieval]
Remote Data [Storage]

Figure 3.1: Distributing Data Storage and Retrieval between Client and Server

3.1.2 Components based on features

Once the content of the application is known, developers need to deduce what information is to be presented to the users of the application and how the information is presented to the users. This can be done by listing all the features the application provides to its users and prioritizing these features based on importance.

For example, if an application must present its users with news, its features could be, say, displaying: 1) international, national and local news, 2) political news, 3) weather forecast, 4) advertisements, 5) events for that week, 6) sports news, 7) comics, 8) puzzles, and 9) job opportunities. This application needs to present a variety of information to its users. Each category of information that is displayed to its users can be termed as a feature. For users that do not want to spend a lot of time reading all the information on the application, each feature of the application can be displayed to the user separately in sections and the users can pick what section of the news they would like to read. Prioritizing the list of features in an ascending order of importance to its users can help with displaying information for users who would like to read all the information offered to them. Giving international, national and local news, political news, weather forecast, and sports news a high priority and giving events for that week, comics, puzzles, job opportunities and advertisements a low priority would make users view the high priority items first and the low priority items towards the end.

Thus, two decisions are made in this step: 1) dividing content into features that the application provides, and 2) prioritizing the features in an order of importance of its users.

3.1.3 Component interaction

In this step, components ascertained from the previous action are closely inspected for interaction and dependencies. Developers need to know how one component of the system affects the other components. This is the first of two steps in discovering the valid and invalid states of an application. Single page applications require the developers to administer the outcomes of valid and invalid states because the client is responsible for navigation, and this means that there are no page loads and so the users will not get an error message unless the developers provide the application a means to let the user know that something went wrong. Developers need to analyze the consequences of a failed component and implement what happens in such a case.

For instance, in the above-mentioned news application, if the default view that is loaded with the index.html file is not displayed to the user, for reasons like slow internet speeds, or disabled JavaScript in the client machine, and this default view contains UI elements that let the user navigate to other parts of the application, the developer needs to decide what the application should do.

3.1.4 Data flow between components

Finding the data flow between the components is the second of two steps in discovering the valid and invalid states of an application. Single page applications rely on data flowing between its components, so that the application can use AJAX to make requests to the server for portions of information. If one component fails to provide data to other components as intended, the developers need to handle what must be done to make the application continue to provide intended services to its users.

For instance, in the above-mentioned news application, the locations of the users need to be found before the users can view the local news, or the weather forecast for their area. Thus, local news and weather forecast depend on finding the users' locations. If, for some reason, the application fails to acquire user locations, the developer needs to decide what the application should do.

3.1.5 Define the states of the application

Based on the details deduced in the previous two steps, the developers must be able to formulate the valid states of the application, how these valid states intend to communicate with each other, and how data flows from one state to another. Once the valid states of the application are defined, the developers need to anticipate how each valid state can fail and cause the application to reach an invalid state. Assuming all states that are not in the list of valid states to be invalid states, the developers can return to a valid state by navigating the application to a valid state as soon as an invalid state is encountered.

For instance, assume in the news application above that users are allowed to create an account and save two locations as their favorite locations for fast access of local news and weather forecast, and the application is unable to connect to the Internet to request data for the two favorite locations when requested. In this case, the application was in a valid state before the user requested information about the favorite locations and it moved to an invalid state after the user requested for the new information. The developers need to provide a way for the users to navigate from such invalid states to a valid state. This can be done by providing a way to request new information again or by navigating the user to the valid state before the invalid state was encountered.

At the end of analyzing the architecture of the application, the developers need to create a workflow, considering all the valid states of the application and factors that may cause the application to encounter an invalid state.

For instance, the workflow of the news application is depicted in Figure 3.2. The blue boxes indicate components of the application that need to be present throughout the lifespan of the application, the green boxes indicate the components with high priority, and the box in grey indicates components with low priority. The arrows in black indicate valid state transitions and the arrows in orange indicate what the application does when the application encounters an invalid state while being in a valid state.

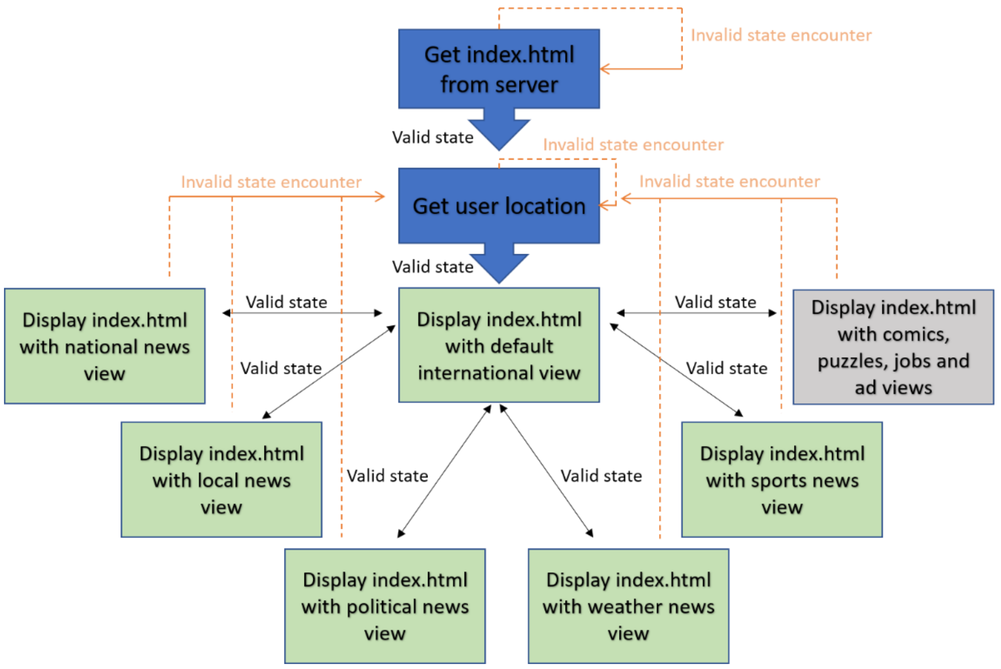

Invalid state encounter
Get index.html from server
Valid state
Invalid state encounter
Invalid state encounter
Invalid state encounter
Get user location
Valid state
Display index.html with national news view
Valid state
Valid state
Display index.html with default international view
Valid state
Valid state
Display index.html with comics, puzzles, jobs and ad views
Display index.html with local news view
Valid state
Valid state
Display index.html with sports news view
Display index.html with political news view
Display index.html with weather news view

Figure 3.2: Workflow of the News Application

3.2 Framework

The goal of this series of steps is to identify the architectural pattern and the structure for the application based on the architectural decisions in the previous step.

3.2.1 Architectural pattern

The outcomes of Section 3.1.4 and 3.1.5 not only allow developers determine the workflow of the application but also gain a high-level understanding of what the views of the applications contain. For instance, in figure 3.2, the boxes in green are not only the valid states of the application but, they can also be the individual views of the application. Thus, for the news application mentioned earlier, each category of news can be displayed to the user as a separate view. The term view refers to the content of the application that is visible to the user at any given point in time.

The model and the views of the application need to interact with one another to display the correct information from the model and make sure any changes in the view are reflected to the model. The interaction between a view and a model can be categorized into three basic types:

a) Controllers – A controller is a link between the model and the view. It provides the user with input by displaying the relevant views in the appropriate parts of the screen. A controller allows developers to collect data from the user interaction, process data, and update or store data in the model.

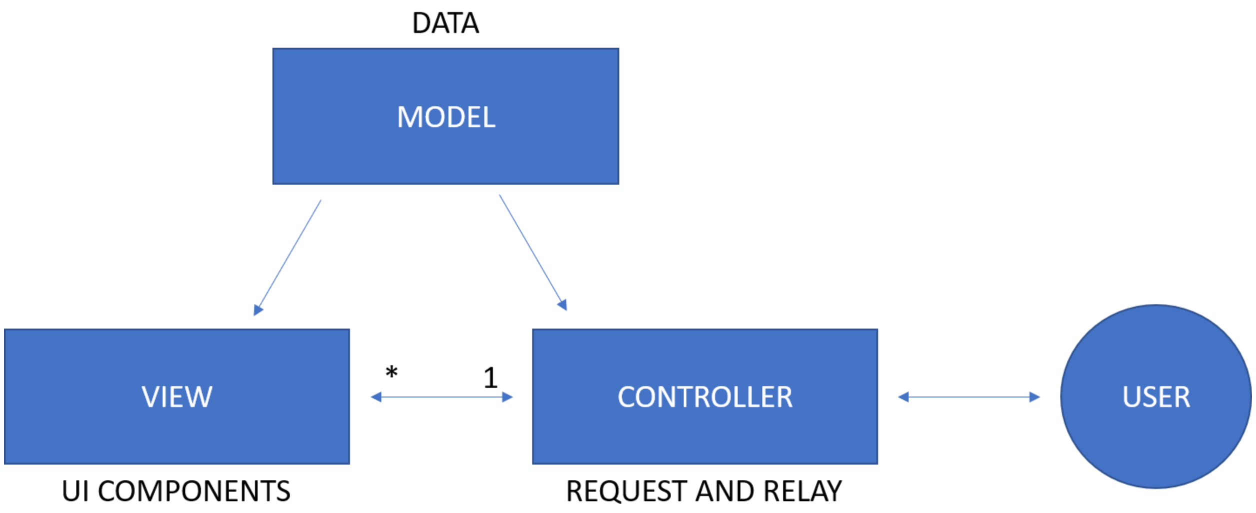


Figure 3.3: MVC Pattern

As shown in figure 3.3, the controller processes incoming requests from users via the view, then processes the request through the model and then passes back the results to the view. There is a many to one relationship between the view and the controller. This means that one controller can be responsible for multiple views.

b) Presenters – A presenter contains business logic of the user interface for the view. When a user invokes an event in the view, the event is delegated to the presenter, which is decoupled from the view and the view can interact with the presenter via an interface. In this pattern, the views contain little-to-no logic. Instead, the presenter observes the model and updates the views when models change. The presenter retrieves data, manipulates data and determines how the data should be displayed in the view. Thus, the concept of direct data binding is eliminated.

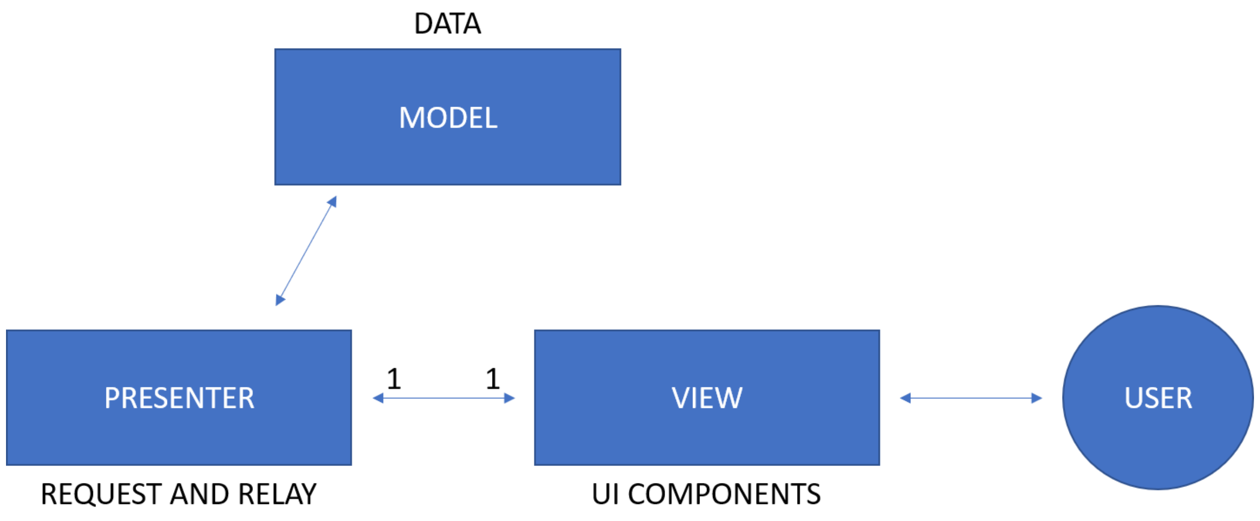


Figure 3.4: MVP Pattern

As shown in figure 3.4, the presenter addresses all user interface events on behalf of the view. It receives input via the view, processes the data through the model, and passes the results back to the view. The view and the presenter communicate to each other by an interface. There is a one to one relationship between the view and the presenter. This means that a presenter is responsible for only one view.

c) ViewModel – A ViewModel exposes the properties related to the model for data binding. It can contain interfaces to services and data to fetch and manipulate the properties the ViewModel exposes to the view. The view and the ViewModel communicate via data binding, method calls, or events. The view handles its own UI events and then maps the events to the ViewModel. The view is not responsible for maintaining its state. The state of the view is synchronized with the ViewModel when data from the view is bound to the data in the ViewModel interface.

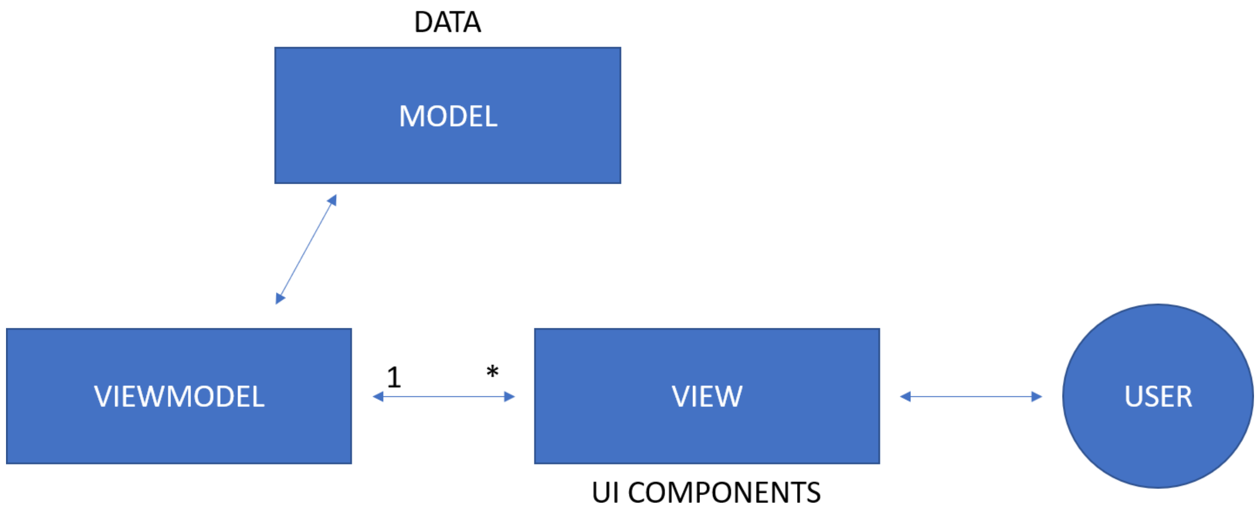


Figure 3.5: MVVM Pattern

As shown in figure 3.5, the ViewModel is responsible for displaying methods and functions that assist in maintaining the state of the view, manipulating the model based on the actions in the view, and triggering events in the view. There is a many to one relationship between the view and the ViewModel. This means that one ViewModel is responsible for many views.

The three patterns mentioned above guide developers in separating their code into sections such that the view is responsible for the user interface, the model is responsible for the application data, and the code to make the view and the model interact is separate from the view and model itself. The main point to consider while deciding a pattern is how the view interacts with the model. In the case of single page applications, since the developers are responsible for navigation, having a many-to-one relationship between the interaction and the view is recommended because the interaction, in the form of a controller, or presenter, or ViewModel, needs to compute what view is displayed to the user at any given point. Each view must have a way to interact with the model and the interface between the views and the model must be able to monitor the state of the application.

In the case of a web application, developers deal with three languages at once: JavaScript, HTML, and CSS. It can get difficult to perfectly match what parts of the application code fit in with parts of a design pattern. To overcome this uncertainty, developers must make sure that the data or content of the application is segregated into the model and data manipulation takes place in one section of the application code, the views contain HTML that is displayed to the users and the manipulation of the views take place in a different section of the application code, and the interaction between the view and the model and the manipulation to the interaction takes place in a separate section of the application code. Before developers can dive into writing code to build an application, it is necessary to understand how the code must be separated into sections, such that each section addresses a different aspect of the application. This separation of layers helps developers build an application that is clean, easy to follow and read, and easy to maintain.

After developers determine the architectural pattern of the application, they can map these architectural needs into one of the existing JavaScript frameworks. A JavaScript framework is a collection of JavaScript libraries that provide developers with built-in features to develop web applications. It provides a skeleton structure for creating an application that arranges the parts of the application where the framework is implemented. For example, if the news application mentioned in the previous section requires the weather view to have a carousel that displays the weather forecast of five cities to the user, a developer can build a carousel from scratch (HTML, CSS and JavaScript code) or use code from a framework to provide the feature to save time.

Narrowing the search to find a suitable framework can be an intimidating task as there are numerous JavaScript frameworks available to developers that offer a wide range of features to build a web application. Frameworks are tools that developers use to make their task simple. The specific needs of the application should always be the main criteria when selecting a framework. Table 3.1 highlights key features of three frameworks: Backbone.js, Knockout and AngularJS.

FEATURES	**KNOCKOUT**	**BACKBONE.JS**	**ANGULARJS**
Documentation	Good documentation that is easy to understand	Simple library. Documentation not as extensive as Knockout or AngularJS. Relies heavily on Underscore.js	More complex than Knockout and Backbone.js. Requires more time investment. Getting started is easy if developer is aware of Knockout because data binding is similar. Well organized tutorials.
Model framework	MVC and MVP	May not perfectly fit with the original MVVM definition but it is close	MVW framework
Model	Model needs to be explicitly declared by the framework as a JavaScript object. Models can have logic in addition to data, such as validation, default data and custom functions.	No restriction on what can be used as the data source (Implicit). If input comes from an input field (HTML form), add a custom attribute called data-bind to each field.	No restriction on what can be used as the data source (Implicit). If input comes from an input field (HTML form), add a custom attribute called ng-model to each field.
Front End	Models and views are created programmatically using JavaScript code by extending Backbone.js objects. This enables developers to inherit a lot of built in functionality.	Has a native templating mechanism that allows developers to capture HTML and use it as a template for other items.	Allows developers to add special behaviors to HTML code or the DOM. AngularJS also allows developers to create their own directives.
User interaction	Leaves many decisions to the users to pick tools required for their specific needs. Easily extensible UI by modifying and extending default function using custom binding and custom functions.	Does not force user to work in a particular way. Offers solutions for routing and data access.	Can function in a part of the application, so it does not get in the way of other frameworks. More abstract that Knockout and Backbone.js. It provides ready-made UI elements.

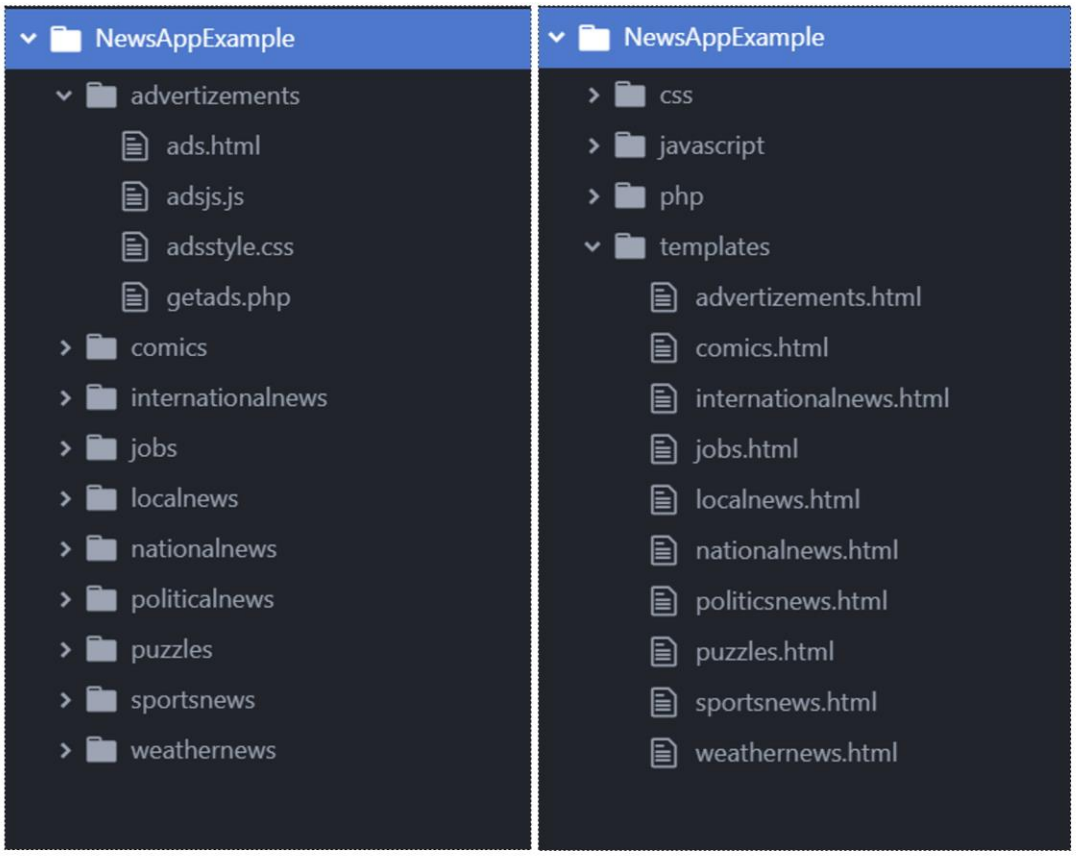
Debugging and testing	No specific debugging tools, but works with JavaScript testing tools such as Jasmine. knockout.toJSON() converts a knockout view-model to JSON object, so that users can see how data is changing in real time.	There are some plugins to help with debugging. Jasmine can be used for unit testing. Relatively difficult to debug.	Built-in testing tools. Batarang – an extension on Google Chrome to monitor the application's model and performance. Jasmine can be used for testing. Plugins to run automated test cases exist. Ideal for unit-testing because of dependency injection.
Data Binding	Data-binding available but syntax can get complicated when model size increases.	Requires more code to bind the model and the view. There is no data binding readily available.	Two-way binding keeps the model and the view synchronous. Without this, developers would need to write a lot of code that keeps the model and the view in sync constantly.
Routing	Does not provide users with routing; provided through third party libraries	Provides developers with directives to implement routing	Provides developers with directives to implement routing

Table 3.1: Key Features of Existing JavaScript Frameworks

3.2.2 File structure

Based on the architectural pattern selected in the previous step, developers need to determine the file structure of the application. Files can be categorized based on the features of the application or the type of files. If files are categorized based on the features of the application, each feature must have a folder that consists of the HTML, CSS, JavaScript and other types of files required to execute that feature. If developers need to modify a feature, they would know to make changes to files in its folder. If files are categorized based on the type of file, the application source must contain one folder for each type of file: HTML, CSS, JavaScript and other required files. If developers need to modify a feature, they would know to make changes in all the files that contain code for that feature in all the required folders.

For instance, in the news application mentioned in the previous section, as shown in figure 3.6 a, if the files were to be categorized based on the features of the application, the application would consist of 10 folders, one each for international news, national news, local news, political news, weather news, sports news, comics, puzzles, jobs and advertisements. Each folder would then consist of 1) a HTML file that displays details about that feature, 2) a CSS file that contains styles for the HTML layout, 3) a JavaScript file that controls the events in the UI and binds data to its model and 4) if required, a server-side script file that send and receive information to and from the server.



```
NewsAppExample
    advertizements
        ads.html
        adsjs.js
        adsstyle.css
        getads.php
    comics
    internationalnews
    jobs
    localnews
    nationalnews
    politicalnews
    puzzles
    sportsnews
    weathernews
```

```
NewsAppExample
    css
    javascript
    php
    templates
        advertizements.html
        comics.html
        internationalnews.html
        jobs.html
        localnews.html
        nationalnews.html
        politicsnews.html
        puzzles.html
        sportsnews.html
        weathernews.html
```

Figure 3.6: a) Files structured based on features of news application b) Files structured based on type of file for the news application

Using the same example, if the files in the news application were to be categorized based on the type of files, as shown in figure 3.6 b, the application would consist of 4 folders, one each for 1) HTML files to display each type of news, 2) CSS files to add styles to the HTML files, 3) JavaScript files to control the events in the UI and bind data to its model and 4) if required, server-side scripts to send and receive information to and from the server.

For large applications, developers may use a hybrid file structure that divides the application into features and for each feature, the files can be structured based on type of files. This makes adding new features easy and allows for multiple developers to work on the project together without changing files that do not belong to the feature that they are working on.

3.3 Model

Once the framework has been selected, the content of the application can be translated into a model for the application. Based on the application, the developer must choose what kind of storage is required. The data could be stored locally on the client machine, or through data storage services available online, such as Amazon cloud storage service or Firebase hosting, or user inputs from HTML forms itself, to name a few.

3.4 User Interface

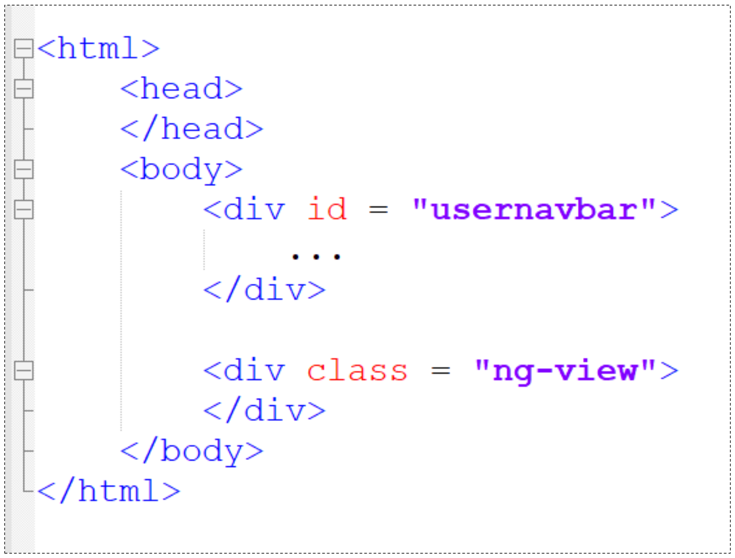
A single page application consists of one HTML page and bundles of CSS and JavaScript files. The index page of a single page application is similar to creating a web page but a portion of the page is reserved for the dynamic content that is downloaded by the browser using AJAX. The entire application runs as a single web page and the presentation layer of the application is managed from the browser. In this type of contemporary application, the "pages" of the application are not really pages. They are parts of the screen that the user can see at different times and are independent sections of the application's content called "views". These views are shown to the user on demand at specific placeholders by executing JavaScript code to render content to the view. The index page of the application generally has a container which is used to replace views as necessary. Views are attached to the DOM dynamically, such that one view is replaced by another view when the user navigates. The elements in the UI can be styled as required and the corresponding CSS files must be placed in the folders, based on the type of file structure chosen in the steps above.

Using the workflow from Section 3.1.5, developers must create individual views for each component identified. The parts of the application that need to run throughout the lifespan of the application can be put on the initial HTML page. This can include headers that contain navbars or sidebars, a label that displays a username, a sign-out button, etc. and footers that may contain quick links to forums or a different web page, if any.

For instance, for the news application mentioned as an example in the previous sections, the component of the application that need to run throughout the lifespan of the application was determined to be the location of the user, the high priority components were the blocks in green that displayed the various types of news (international, national, local, political, weather and sports) and the low priority components were determined to be comics, puzzles, jobs and ads. Individual views must be created for each of these components.

3.4.1 Index.html page

First, an index.html page needs to be created that contains a section for dynamic contents to load as a user navigates or requests for additional information. This can be done without the help of any frameworks by assigning an 'id' attribute to a <div> block and updating the contents of the block by accessing the div element using JavaScript code. But, JavaScript frameworks provide attributes that can be added to HTML tags to achieve this dynamic content loading. For example, as shown in figure 3.8, AngularJS allows developers to add custom HTML attributes to tags that make dynamic content loading easy. The controller defined in the JavaScript files manages routing and the content provided by routing is put into the "ng-view" directive that acts as a container.



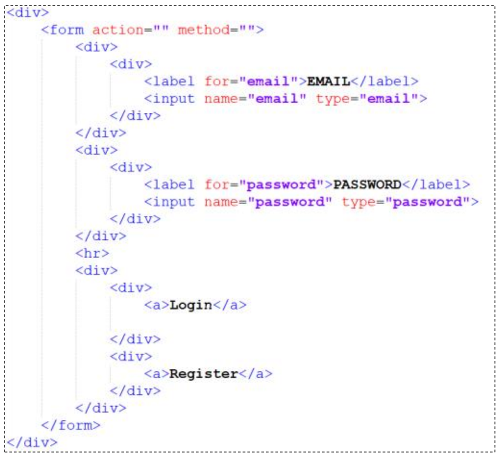
```html
<html>
    <head>
    </head>
    <body>
        <div id = "usernavbar">
            ...
        </div>

        <div class = "ng-view">
        </div>
    </body>
</html>
```

Figure 3.8: Using AngularJS to Create a Container for Dynamic Views

3.4.2 Dynamic views

After the index.html page is set up with a container, the dynamic views must be created. These views can be created using HTML code snippets or they can be created dynamically using JavaScript by updating the DOM elements.



```html
<div>
    <form action="" method="">
        <div>
            <div>
                <label for="email">EMAIL</label>
                <input name="email" type="email">
            </div>
        </div>
        <div>
            <div>
                <label for="password">PASSWORD</label>
                <input name="password" type="password">
            </div>
        </div>
        <hr>
        <div>
            <div>
                <a>Login</a>

            </div>
            <div>
                <a>Register</a>
            </div>
        </div>
    </form>
</div>
```

Figure 3.9: An Example of a Login View

Figure 3.9 shows a view created using a HTML code snippet. It does not contain <html>, <head> or <body> tags because the application appends these views into the container that was created in the index.html page.

The news application requires ten dynamic views. The views must be kept in the correct folder based on the file structure selected for the application. If the ads in the ads view of the application is retrieved from an ad server, the JavaScript code must request for information from the ad server and map the response from the server to the ad view, thus splitting computation between the client and the server. If the ad view, as a whole, needs to be retrieved from a server then, the JavaScript code must map the response from the ad server to the container in the index.html page. This would make the news app a hybrid web application in which a part of the application is a single page application and the other part of the application uses a traditional approach by requesting information from a server every time the user navigates to that view. This thesis does not cover hybrid applications and finding a design approach for a hybrid application will be added to the future efforts.

3.5 Weaving JavaScript and CSS into the Application

The JavaScript code in a single page web application is responsible for the client-side computing, monitoring the state of the application, creating dynamic views, and switching between the dynamic views as requested by the user.

3.5.1 Creating dynamic views

As shown in figure 3.10, dynamic views can also be built using JavaScript code by modifying the DOM elements. When the user requests for new data or navigates to a new view, developers must first ensure the application is in a valid state. If the state is valid, the DOM elements can be updated with the new view.

```
<script>
    var newelement = document.createElement("div");
    newelement.id = "loginview";

    var paragraph = document.createElement("p");
    var textforparagraph = document.createTextNode("Login");
    paragraph.appendChild(textforparagraph);
    newelement.appendChild(paragraph);

    var emailinput = document.createElement("input");
    emailinput.id = "loginemailinput";
    emailinput.setAttribute("type", "text");
    emailinput.setAttribute("value", "Email");
    newelement.appendChild(emailinput);

    var pwdinput = document.createElement("input");
    pwdinput.id = "loginpwdinput";
    pwdinput.setAttribute("type", "password");
    newelement.appendChild(pwdinput);

    var loginbutton = document.createElement("Button");
    var textForButton = document.createTextNode("Login");
    loginbutton.appendChild(textForButton);
    loginbutton.addEventListener("click", function(){
        ...
    });
    newelement.appendChild(loginbutton);

    var containerid = document.getElementById('dynamiccontent');
    containerid.appendChild(newelement);
</script>
```

Figure 3.10: JavaScript code to add dynamic views to index.html page

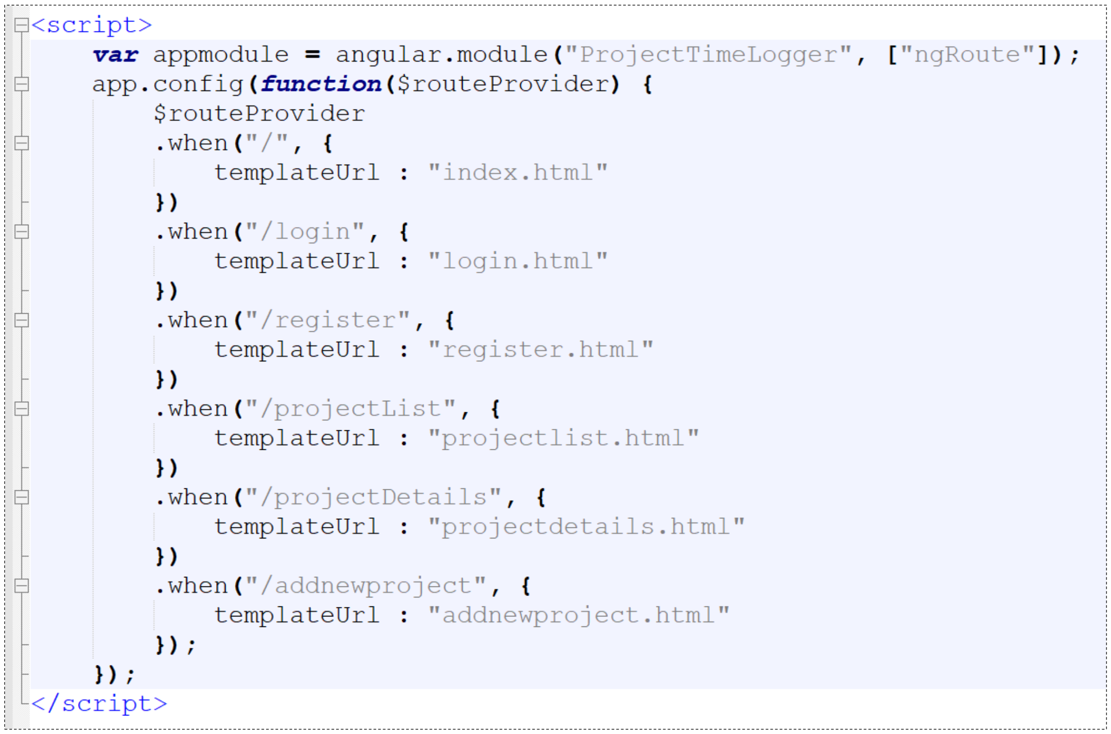
3.5.2 Monitoring the state of the application

Based on the workflow of the application identified in Section 3.1.5, developers must make sure that if the application encounters an invalid state, the JavaScript code leads the application to a valid state without causing the application to crash. The two important points to consider to monitor the state of the application are: 1) assign URLs to the address bar of the browser for every view and check the address bar of the browser for a valid URL when navigating from one view to another, and 2) if the request from a server was not fulfilled as expected, allow users to navigate to a valid view by cancelling the request or allow users to request for the service from the server, limiting the number of times the user can resubmit a failed request. AngularJS, for example, provides a way to assign URLs for the dynamic views when handing the routing in the application. This is shown in figure 3.11. When the JavaScript code encounters an invalid URL that is not listed in the app.config() function, it automatically routes the application to the view that is linked to the default “/” URL.

3.5.3 Navigation between views

Once the index.html page and the dynamic views are built and the model is set up, the interaction between them must be created based on the architectural pattern chosen in the previous section. If developers chose to use a MVC pattern, for example, one or more controllers need to be built for the application. The application needs one controller to control the routing (navigation of the views) and maintaining the state of the application. This controller is usually termed as the application controller and drives the other controllers of the application to function as expected. It hands over control from one controller to another depending on the state of the application and the view that is displayed to the user at any given time.

The news application from above must possess the user location before the index page can display news of any type. If the application uses an existing API to get the user location from a remote server, the application controller must request service from the remote server. If the controller receives information as expected, it saves the state of the application and hands control over to the international news controller, which then displays international news for the location of the user, provided the news is available locally. If the application uses an API to get international news for a particular location, the international news controller then requests for new information and processes this information and displays the information as defined by the international view. If the user requests for sports news while the application is displaying international news to the user, the application controller removes the international news view from the container and adds the sports news view to the container in the index.html page. The application controller then transfers control from the international news controller to the sports news controller. Figure 3.11 shows how AngularJS allows developers to add a controller for every view.



```
<script>
    var appmodule = angular.module("ProjectTimeLogger", ["ngRoute"]);
    app.config(function($routeProvider) {
        $routeProvider
        .when("/", {
            templateUrl : "index.html"
        })
        .when("/login", {
            templateUrl : "login.html"
        })
        .when("/register", {
            templateUrl : "register.html"
        })
        .when("/projectList", {
            templateUrl : "projectlist.html"
        })
        .when("/projectDetails", {
            templateUrl : "projectdetails.html"
        })
        .when("/addnewproject", {
            templateUrl : "addnewproject.html"
        });
    });
</script>
```

Figure 3.11: JavaScript file showing routing using AngularJS framework

A developer might be tempted to add all the JavaScript code into one file but there are disadvantages to doing so: 1) If all the JavaScript code is in one file, the file size would be large and the initial download for the application would take more time. 2) If a new feature needs to be added to the software, the JavaScript file would need to be modified and this could cause the file to get corrupt if the new code is not written correctly. 3) Collaboration would be a messy task if multiple developers need to work on the same JavaScript file. 4) With multiple JavaScript files, one for every view, source code can be divided and uploaded to different web servers flexibly.

3.5.4 Client-side computation

Once the JavaScript code is set up to implement routing, monitor the states of the application, and creating dynamic views, developers can start using the JavaScript code to perform client-side computing required for the application. Any computation that is not done by the server or received from the server is managed by the client-side machine. For instance, in the news application, the client requests for puzzles from the server for that day. If the response from the server is a solved crossword puzzle with information on which cells are meant to be empty, the client can compute what the unsolved crossword puzzle must appear to the user and display the unsolved puzzle to the user. The client can also compute what the application does when the user solves the crossword correctly or incorrectly. The client-side computing can include a way to help users with clues or solve the crossword puzzle if required.

3.5.5 Adding styles to HTML page and views

The index.html page and the views of the application can have style sheets linked to them individually or in one file. The <link> tag can be used to specify the source of the style sheet.

This also allows developers to use existing frameworks like Bootstrap to quickly create views with remote CSS files.

3.6 Accessing the model and Server-side code

The HTML, CSS, and JavaScript code help developers create a single page application that looks appealing and does not reload every time the user navigates. If there is any kind of data that must be accessed from the model that is in the form of a database, developers need to handle that using server-side code. Developers are free to choose a scripting language that they are comfortable to work with. There are many server-side scripting languages such as PHP, ASP, ASP.NET, JSP, or NodeJS. NodeJS allows the user to work in a full stack development environment using JavaScript, allowing developers to use a single programming language for the application. But, if developers are already aware of a different server-side scripting language and time is of an essence, they can choose the one that would take the least amount of time for them. If existing APIs are being used by the application, there would be no need to write server-side code.

For instance, for the news application, if the users can create accounts for themselves and store five locations as their favorites, the application could use a local database to store user details and location preferences and use existing APIs to find current user locations and the news for the locations. In such a case, code is necessary to access data from the local database and existing APIs can be called using JavaScript code.

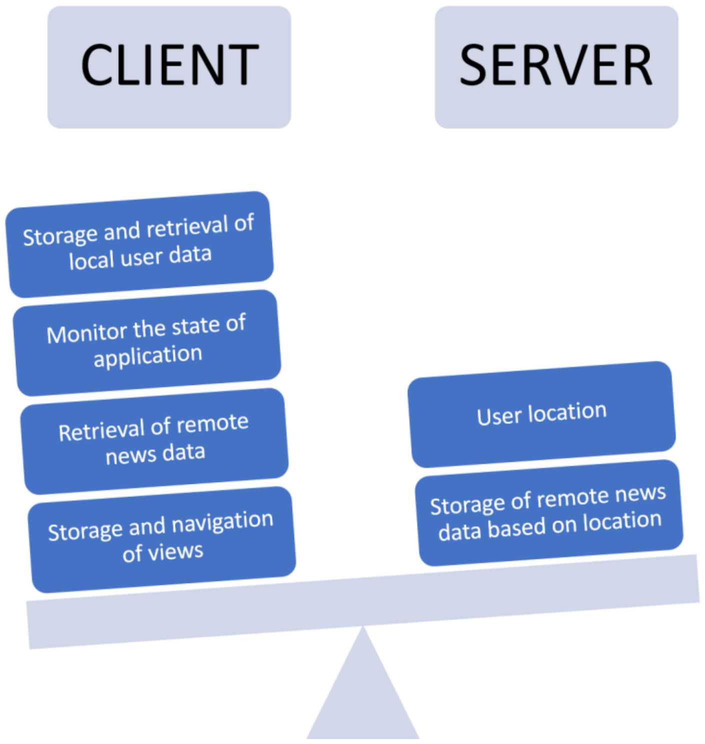

CLIENT
SERVER
Storage and retrieval of local user data
Monitor the state of application
Retrieval of remote news data
Storage and navigation of views
User location
Storage of remote news data based on location

Figure 3.12: Distribution of Computation between Client and Server for News Application

Figure 3.12 shows the distribution of computation between the client and the server for the news application based on all the assumptions made in each step. In summary, developers must start by assuming the application to be built is a client-side application and analyze various aspects of the application development process such as 1) architecture, 2) file structure, 3) framework, and 4) dividing the application into a model, a view and an interface that lets the model and the view communicate. Once these aspects are studied and the developers have a workflow of the application, a contemporary application can be developed.

Designing a Contemporary Web Application

To validate the proposed design approach in the previous chapter, an application, named ProjectTimeLogger, was developed that allowed its users to create a profile (register), sign in to the application and log how much time the user spent on working on a project. The requirements of the application are shown in Table 4.1.

REQUIREMENT NUMBER	**REQUIREMENT SPECIFICATION**
R1	Users should be able to create a profile and login to the application.
R2	Users must be able to add projects to the application.
R3	Users must be able to see a list of all the projects added to the application along with a few details of the application – last modified date, the current iteration of the project, total amount of time spent on the project and the last activity performed on the project.
R4	On selecting a project from the list, the user will be shown more information about that project – total time spent on that project, amount of time spent on the current iteration of the project, amount of time spent on the current activity throughout the project and amount of time spent on the current activity in the current iteration of the project.
R5	The user will be able to start, pause and stop a timer which will log the time spent on that project, which in turn will add to the total time spent on the project.
R6	The user will also be able to enter time manually, if they do not want to run the timer.
R7	The user should not be able to change previously logged time.
R8	The user should be able to mark the end of an iteration. This should increment the iteration number.
R9	The user should be able to mark a project as completed. Once a project is marked completed, the user should not be able to make any changes to the project. However, the user may view all the logged data of a completed project.
R10	A user must be able to customize the activities that can be performed during the process of development of the project.

Table 4.1: ProjectTimeLogger Requirements

4.1 Architecture

4.1.1 Determine components based on source of content

Each requirement in table 4.1 is an individual component of the application. Requirement R1 from table 4.1 suggests that user information for login and register needed to be stored in a remote database, so that the application can be used from any client-side machine. Requirement R2 requires users to be able to add projects to a list, which suggests that a list of projects must be stored for every user. Since this data must be available to the user on any client-side machine, the list of projects must be stored in a remote location. Requirements R3 and R4 suggest that the user must be able to view the details of any selected project. Thus, this data needs to be stored remotely as well. Requirements R5 and R6 states that the user should be able to start a timer or manually enter time to log the amount of time spent working on a project. Thus, the application needs to compute time (start, stop and reset timer, or enter time in HTML input field) and store the new time in the remote location for future use. Requirements R8, R9 and R10 suggests that more details about a project need to be stored for the user to access from any client-side machine. Figure 4.1 shows this distribution of data storage and retrieval for ProjectTimeLogger.

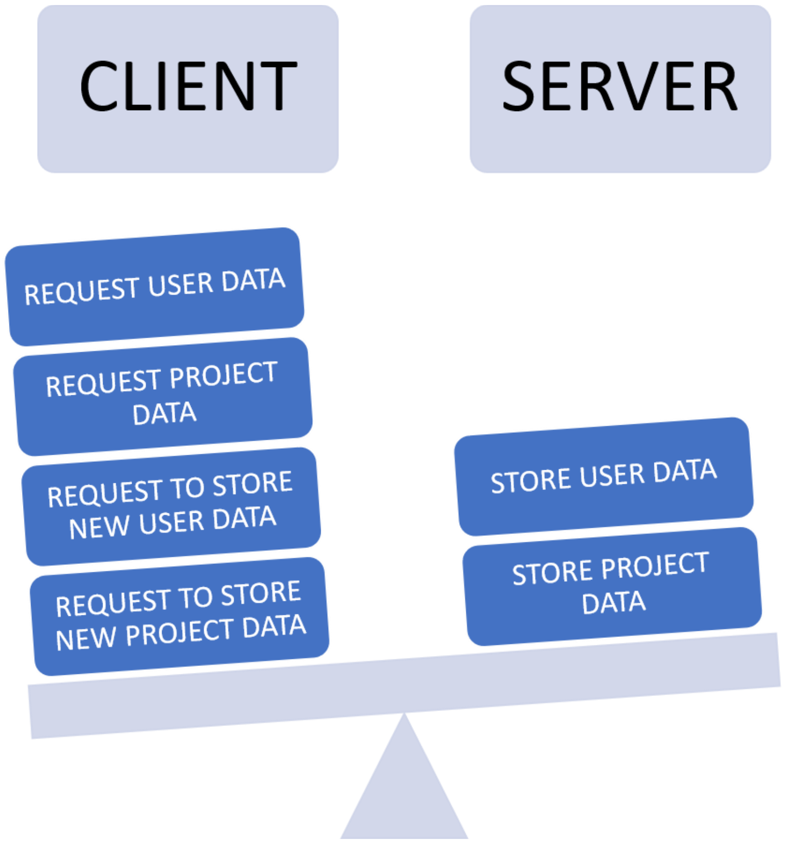


Figure 4.1: Distribution of data based on source for ProjectTimeLogger

4.1.2 Determine components based on features

The components of the application were defined based on the requirements and they were each given a priority as shown in Table 4.2.

PRIORITY	REQUIREMENT NUMBER	REQUIREMENT SPECIFICATION
P1	R1	Users should be able to create a profile and login to the application.
P2	R2	Users must be able to add projects to the application.
P3	R3	Users must be able to see a list of all the projects added to the application along with a few details of the application – last modified date, the current iteration of the project, total amount of time spent on the project and the last activity performed on the project.
P4	R4	On selecting a project from the list, the user will be shown more information about that project – total time spent on that project, amount of time spent on the current iteration of the project, amount of time spent on the current activity throughout the project and amount of time spent on the current activity in the current iteration of the project.
P5	R5	The user will be able to start, pause and stop a timer which will log the time spent on that project, which in turn will add to the total time spent on the project.
P7	R6	The user will also be able to enter time manually, if they do not want to run the timer.
P6	R7	The user should not be able to change previously logged time.
P8	R8	The user should be able to mark the end of an iteration. This should increment the iteration number.
P9	R9	The user should be able to mark a project as completed. Once a project is marked completed, the user should not be able to make any changes to the project. However, the user may view all the logged data of a completed project.
P10	R10	A user must be able to customize the activities that can be performed during the process of development of the project.

Table 4.2: Prioritized ProjectTimeLogger Components

4.1.3 Determine component interaction

Users must register and login before they can use ProjectTimeLogger to log time. This must be the entry point to the application. If the registration of a new user fails, the application must notify this error to its user and allow the user to try and register again. If the login of a user fails, the application must allow the user to try and login again, and if the user needs to register before logging in, the application must provide access to the user to navigate to the register view. If the server fails to respond to a user request for the list of projects or project details, the application must notify this error to the user and allow the user to try and fetch the list of projects or project details from the remote server. Lastly, after computing the time that needs to be logged for a project, the client must send this data to the server and update the project records. If the server is unable to update the new logged time, the application needs to notify this error to the user and allow the user to try and log the computed time again.

4.1.4 Determine data flow between components

ProjectTimeLogger requires that users register with their details before they can login. The users need to login after creating an account using their email address and password. Once the login is successful, the UserID of the user is stored as a session variable. Whenever new information is requested from the server, the application checks for the session variable and if it is available, information is provided to the user but, if the session variable is not available, the user is asked to login to the application again. The application resets the session variable when the user signs out of the application or when the application reaches an unknown state. This ensures that users are only able to see data pertaining to them and that users without an account cannot modify existing data in the model. When a user successfully logs in, they can see a list of projects that they have added to the application and an option to add more projects. When a user adds a project, the user can view the new project in the list of projects. When the user selects a project from the list of projects, the corresponding ProjectID is stored as a part of the session variable. Thus, the inputs for features - login, show list of projects, add a project, and log time are all dependent on the successful registration feature for a user; and the features - show list of projects, add a project, and log time are dependent on the successful login feature for a user. This can be viewed as a containment relationship depicted by figure 4.2 below.

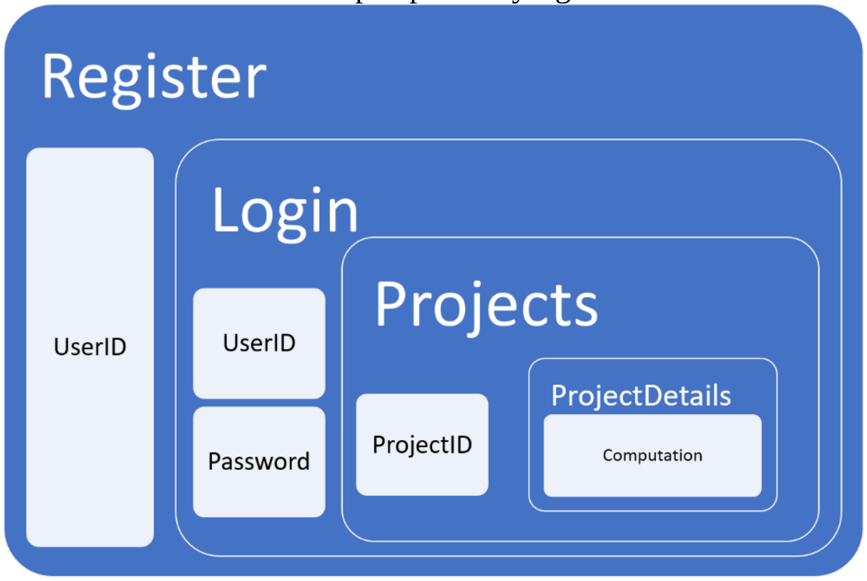
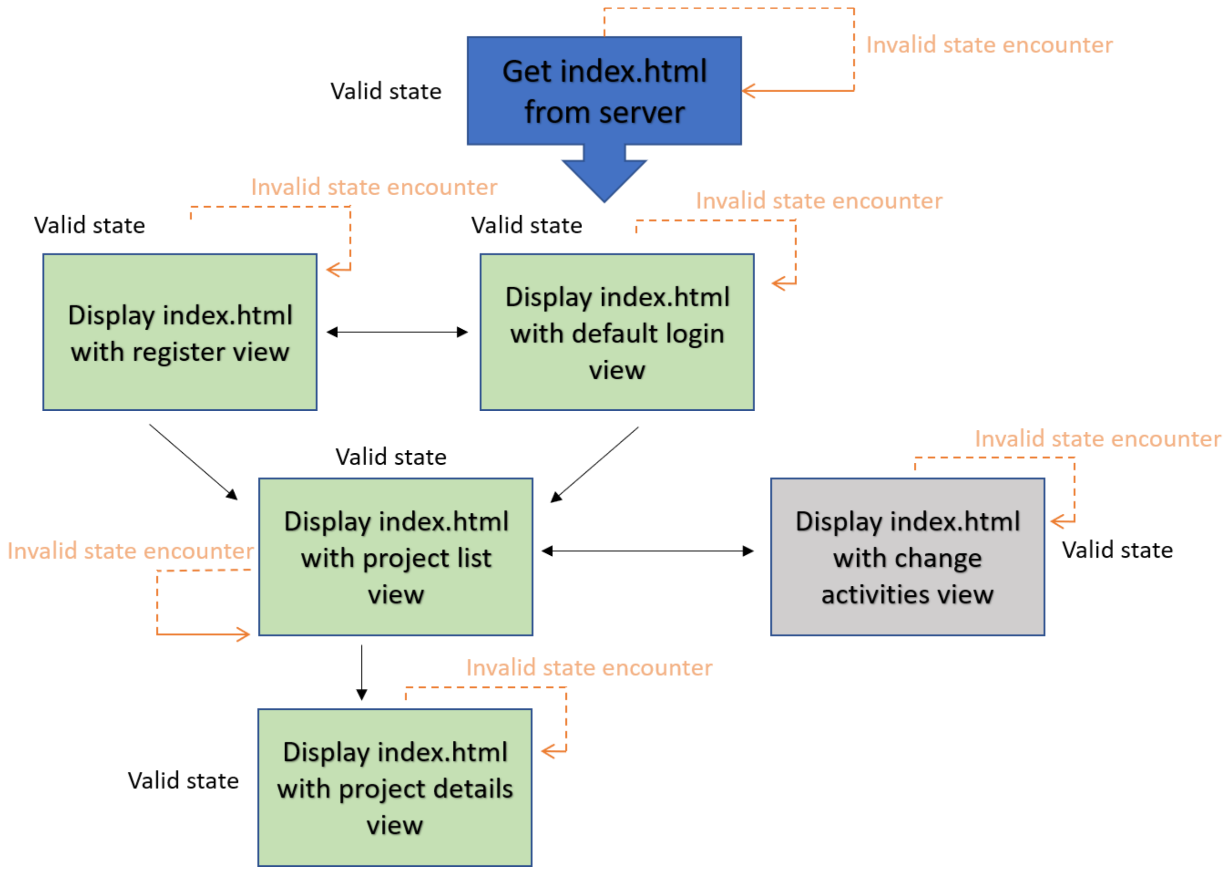


Figure 4.2: Component Relationship

4.1.5 Define the states of the application

Figure 4.3 shows the states of ProjectTimeLogger. The block in blue is requested from the server and it must be displayed to the users throughout the lifespan of the application. The blocks in green are high priority components and the block in grey is a low priority component.



Get index.html from server
Valid state
Invalid state encounter
Display index.html with register view
Display index.html with default login view
Display index.html with project list view
Display index.html with change activities view
Display index.html with project details view

Figure 4.3: Workflow of ProjectTimeLogger

4.2 Framework

4.2.1 Determine the architectural pattern

In Section 4.1.1, the application data storage was distributed between the client and the server. The majority of the data is stored in a remote server that can be accessed by any client-side machine. This data is the model of the application. The index.html page and the dynamic templates are the views of the application. The third aspect of this application's pattern needs to compute which view is shown to the user at any given time, make sure that the data shown to the user is the right data from the model, compute the session variables that ensure the application is always in a valid state, and keep a consistent look while the user navigates from one view to another.

Developing the application requires the application pattern to offer 1) routing between views to create a single page application, 2) the ability to use a data source of choice and data binding to update the time spent on each activity as required both, in the view and in the model and 3) easy to use APIs for routing and binding. AngularJS provides developers with a powerful controller that not only handles routing and data binding, but it also provides a user service that can be used as local storage to handle session data.

4.2.2 Determine the file structure

The data of ProjectTimeLogger will be available in a remote server, as determined from Section 4.1.1. The size of the files does not play a role in determining the file structure as the application code that consists of HTML, CSS, and JavaScript files along with code that accesses data from the model will be stored in a single local web server. ProjectTimeLogger can have multiple features added to it in the future such as: 1) multiple user access, 2) creating tasks in each project and assigning these tasks to individuals, and 3) creating reports for each project, etc. The file structure must allow the developers to add new features to the application easily. ProjectTimeLogger needs to be a single page application, that contains one html page and this html page needs to let the browser download the required CSS and JavaScript files, which in turn allow for “navigation” in the application. Storing all the CSS files together and the JavaScript files together would make development easier. Thus, the files of ProjectTimeLogger are structured based on the type of files.

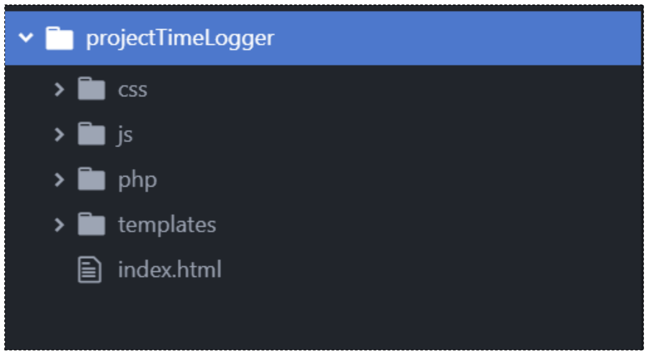


Figure 4.4: Files structured based on file type

4.3 Model

The model of the application contains user data and project data stored in a remote server. ProjectTimeLogger requires a data storage that will allow the application to add multiple users and the users to add multiple projects. The application does not have a limit to the number of projects that can be added by a user. When a user logs time in a project for an activity, the application must store the logged time as a new entry instead of updating existing project information, so that the application can display information about how much time a user has spent in an activity in a project. The data of the application is stored in the form of tables in MariaDB, provided by XAMPP web server package.

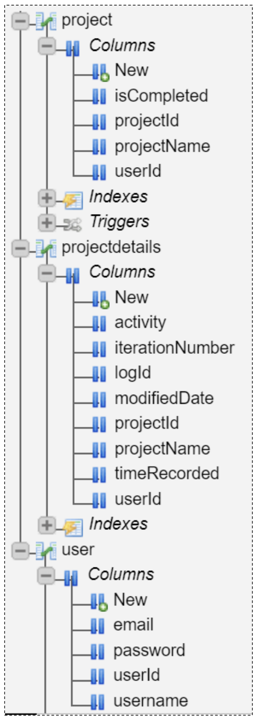


Figure 4.5: Model in the form of tables in a remote database

As shown in figure 4.5, the user details are stored in a ‘user’ table, the list of projects for the users is stored in a ‘project’ table, and the details of the projects are stored in a ‘projectdetails’ table. The three tables are in a database named ‘projecttimelogger’.

4.4 User interface

The big difference in creating a single page contemporary application from creating a traditional one is how the “pages” are set up and how navigation occurs in the application. There is only one HTML page that contains a section for parts of the application that need to be visible to the users at all times and a section reserved for dynamically generated content.

4.4.1 Index.html

The index.html file contains the parts of the application that need to be visible to the user at all times. It contains one section where dynamically generated content is added or replaced as necessary.

The index.html page of ProjectTimeLogger has a navbar that displays the name of the application and a sign out button when a user has logged in to the application; and has a container for dynamic views. Figure 4.6 shows the index.html page with no dynamic views added to the container. This layout makes the size of index.html page small.

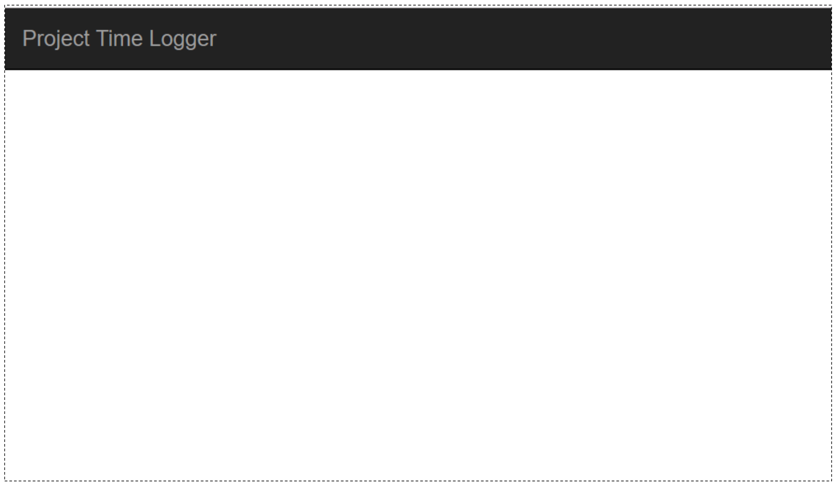
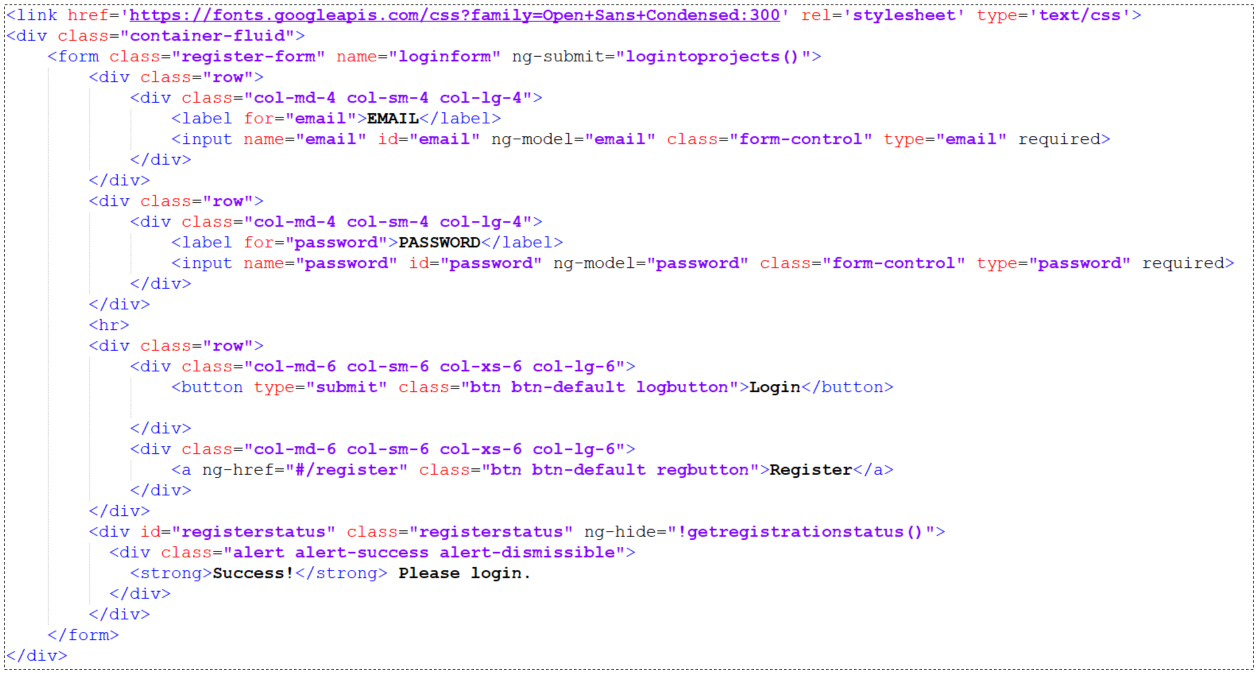


Figure 4.6: Index.html page without a dynamic view

4.4.2 Dynamic views

The dynamic views, also known as templates, can be created using HTML snippets or by using JavaScript code. The templates for ProjectTimeLogger were created with HTML snippets because the layout of the application is static and unchanging. The templates do not require <html>, <head>, or <body> tags in them because they are placed in a container in the index.html page during run time. Figure 4.7 shows the template for the login view of ProjectTimeLogger. The <link> tag on line 1 adds a remote stylesheet to the view. This makes it easy for developers to create views quickly by using existing frameworks like Bootstrap.

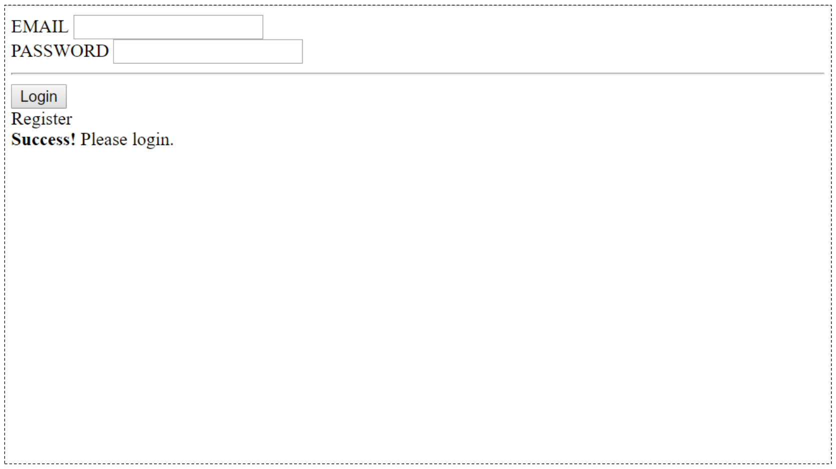


```
<link href='https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300' rel='stylesheet' type='text/css'>
<div class="container-fluid">
    <form class="register-form" name="loginform" ng-submit="logintoprojects()">
        <div class="row">
            <div class="col-md-4 col-sm-4 col-lg-4">
                <label for="email">EMAIL</label>
                <input name="email" id="email" ng-model="email" class="form-control" type="email" required>
            </div>
        </div>
        <div class="row">
            <div class="col-md-4 col-sm-4 col-lg-4">
                <label for="password">PASSWORD</label>
                <input name="password" id="password" ng-model="password" class="form-control" type="password" required>
            </div>
        </div>
        <hr>
        <div class="row">
            <div class="col-md-6 col-sm-6 col-xs-6 col-lg-6">
                <button type="submit" class="btn btn-default logbutton">Login</button>

            </div>
            <div class="col-md-6 col-sm-6 col-xs-6 col-lg-6">
                <a ng-href="#/register" class="btn btn-default regbutton">Register</a>
            </div>
        </div>
        <div id="registerstatus" class="registerstatus" ng-hide="!getregistrationstatus()">
          <div class="alert alert-success alert-dismissible">
            <strong>Success!</strong> Please login.
          </div>
        </div>
    </form>
</div>
```

Figure 4.7: Login template in ProjectTimeLogger

Figure 4.8 shows how the login view would appear to users if it was not placed in the index.html file. The index.html page must contain references to stylesheets to give the application a consistent look. The figure also shows a message in the bottom of the page for success and error. The JavaScript code controls the visibility of these messages based on the success or failure of the login feature of the application. The next section shows how JavaScript is used to control what aspects of the application the user can view at any given time.



EMAIL
PASSWORD

Register
Success! Please login.

Figure 4.8: Login template without adding it to index.html page

4.5 Weaving JavaScript and CSS into the HTML page and views

ProjectTimeLogger has two types of JavaScript controller files. The first type of controller is used to control the entire application, monitor the state of the application and maintain the session variables. The second type of controllers are used to control the individual views of the application.

4.5.1 Monitoring the state of the application

Single page applications do not refresh the page when the user navigates from one view to another, causing the URL in the address bar of the browsers to remain constant. This might seem confusing to the users because they will not be able to bookmark pages in the browser for future references. ProjectTimeLogger addresses this issue by using the feature provided by AngularJS to change a part of the URL when a new view is added to the index.html page. Figure 4.9 shows how app.js handles URLs and session data in ProjectTimeLogger to ensure the application always shows a valid view to its users.

```
var ptlMod = angular.module('ptlapp', ['ngRoute']);
ptlMod.config(function ($routeProvider, $locationProvider) {
  $routeProvider
    .when('/login', { templateUrl: 'templates/login.html', controller: 'loginCtrl' })
    .when('/register', { templateUrl: 'templates/register.html', controller: 'registerCtrl' })
    .when('/projectlist', { templateUrl: 'templates/projectlist.html', controller: 'projectlistCtrl' })
    .when('/addnewproject', { templateUrl: 'templates/addnewproject.html', controller: 'addnewprojectCtrl' })
    .when('/projectdetails', { templateUrl: 'templates/projectdetails.html', controller: 'projectdetailsCtrl' })
    .otherwise({ redirectTo: "/login" });
});

ptlMod.controller('ptlappCtrl', function ($scope,$window,$http,userServices){
  $scope.getsignoutoption = function(){
    return (Boolean(userServices.getloginstatus()));
  };
  $scope.loguserout = function(){
    userServices.setloginstatus(false);
    userServices.setregisterstatus(false);
    userServices.setuserid(null);
    userServices.setusername("");
    $window.location="#/login";
  };
});
```

Figure 4.9: JavaScript file (app.js) that controls the application

4.5.2 Navigation between views

As shown in figure 4.9, the $routeProvider lets the developer set up all the template pages, its controllers, and the URL that the users can see on the browser when they navigate to that view. This, in turn, makes it easy for the controller to handle the back and forward buttons in the browser. When the URL contains /login, for example, the $routeProvider lets loginctrl.js act as the controller for that view. This allows computation to be distributed and developers don't end up with one bulky JavaScript file.

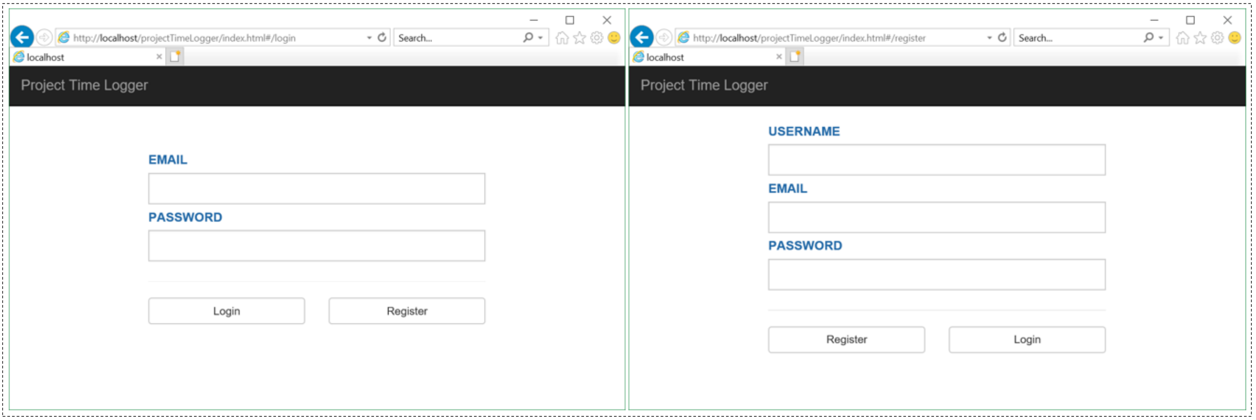
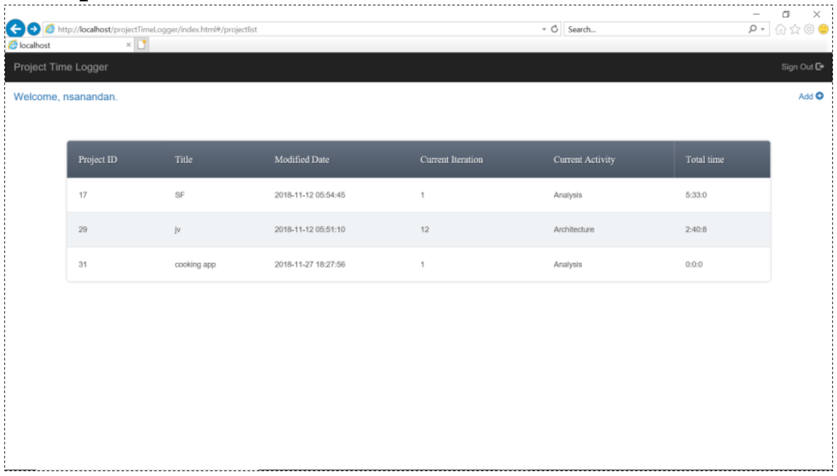


Figure 4.10: Index.html page with a) the login view and b) the register view

Figure 4.10 shows the index.html page with the login template and the register template. A page load does not occur when the user clicks on the register button on the login page. The app.js uses the $routeProvider to change the URL in the address bar of the browser and the control is transferred from loginCtrl to registerCtrl, which is then responsible for any computation on the register view.

4.5.3 Client-side computation



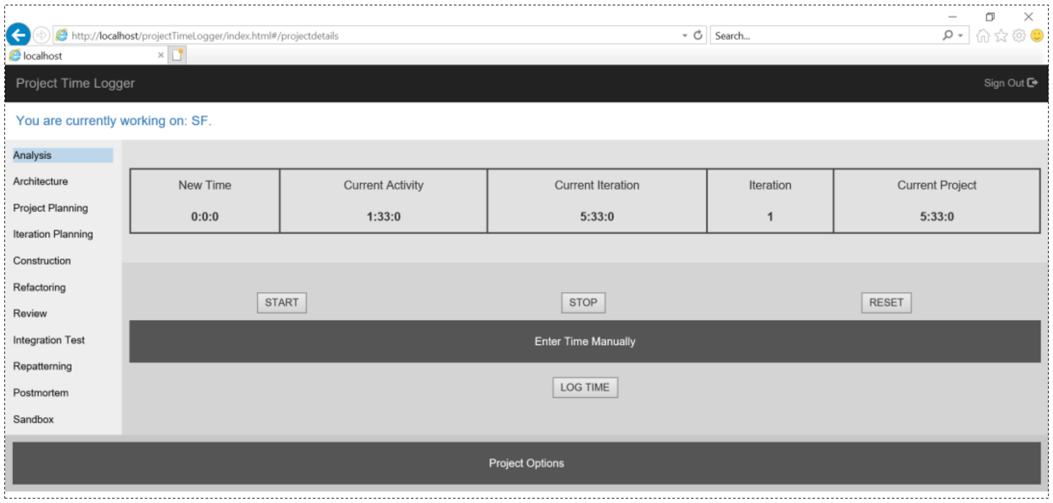
Project ID	Title	Modified Date	Current Iteration	Current Activity	Total time
17	SF	2018-11-12 05:54:45	1	Analysis	5:33:0
29	jv	2018-11-12 05:51:10	12	Architecture	2:40:8
31	cooking app	2018-11-27 18:27:56	1	Analysis	0:0:0

Figure 4.11: Index.html page with projectlist view

As shown in figure 4.11, when a user logs in to ProjectTimeLogger, the application displays a list of projects that the user has added to the database. The $routeProvider changes the URL in the address bar and transfers control from loginCtrl to projectlistCtrl. projectlistCtrl is responsible for requesting project data from the database in the model and mapping the response into the necessary fields in the view. Figure 4.12 shows the projectlistCtrl JavaScript code that retrieves project details from the database using a http post method, processes the information if there was no error in data retrieval, and stores the project details into a variable called ‘rowdata’, which is then pushed to the HTML <tr> tag using data binding provided by AngularJS.

```
for(var i=0; i<$scope.projectids.length; i++)
{
  //console.log(projectids[i].projectId);
  var eachprojectid = $scope.projectids[i].projectId;
  $http.post("php/getprojectinfo.php",
      {
          'userid': usernumber,
          'projectid': eachprojectid
      })
  .then(function(result){
    var projectdetails = result.data;
    console.log(projectdetails);
    var rowdata = [];
    rowdata[0] = result.data.info.projectId;
    rowdata[1] = result.data.info.projectName;
          rowdata[2] = result.data.info.modifiedDate;
          rowdata[3] = result.data.info.iterationNumber;
          rowdata[4] = result.data.info.activity;
    var timerecorded = [];
          timerecorded[0] = result.data.time.hours;
          timerecorded[1] = result.data.time.minutes;
          timerecorded[2] = result.data.time.seconds;
          rowdata[5] = userServices.gettimeinHMS(timerecorded);
    $scope.row.push(rowdata);
  },
  function(error){
          console.error(error);
      });
```

Figure 4.12: projectlistCtrl code to retrieve, process and display project details



You are currently working on: SF.

New Time	Current Activity	Current Iteration	Iteration	Current Project
0:0:0	1:33:0	5:33:0	1	5:33:0

Figure 4.13: Index.html page with the projectdetails view for project ‘SF’

When the user selects a project from the list of projects, the application displays more information about the project and allows users to log time by either using the timer or entering time manually, as shown in figure 4.13.

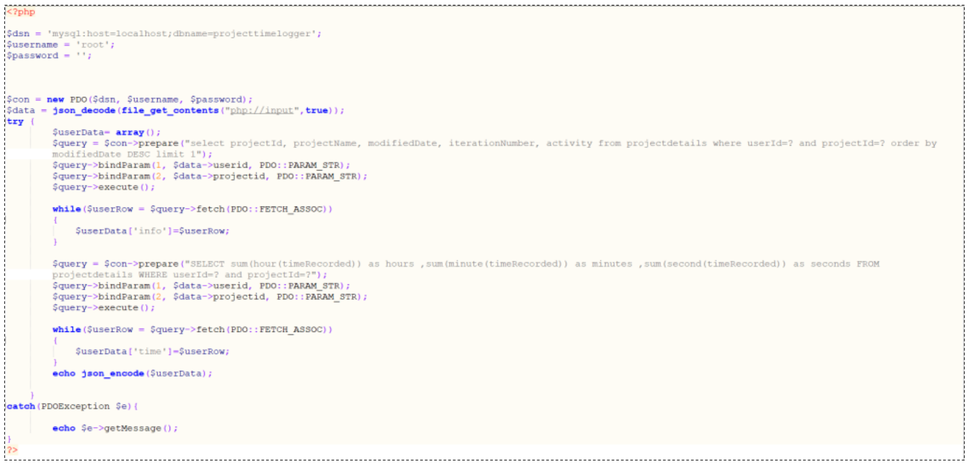
The first timer indicates the new time that is going to be logged for the selected project and the highlighted activity. The second timer displays the total time spent on the highlighted activity for that project, the third timer displays the time spent in the current iteration of the project, and the fourth timer displays the total time spent on the selected project. When the user clicks on the start button, the four timers start to count time: the first timer starts from zero and the other three timers increment time in one-second intervals. When the user clicks on the stop button, all the timers stop running and the user can click on the ‘log time’ button to log the time that is displayed on the screen. The application also provides a ‘reset’ button to reset the timers to the last logged values for the highlighted activity. If the user decides to enter time manually, the ‘enter time manually’ button is clicked, which allows the user to enter time in HH:MM format. The user can then click on the ‘log time’ button to log the manual time. All these computations are done by the projectdetailsCtrl JavaScript code on the client-side machine. Only when the user clicks on the ‘log time’ button, the controller sends a request to the server to add a new entry to the projectdetails table in the remote database.

4.5.5 Adding styles to HTML page and views

ProjectTimeLogger uses Bootstrap JavaScript package elements for the majority of the application. Thus, the CSS is added to the index.html page for the individual elements. The CSS that binds these individual elements together to give the application a consistent user experience is in the ptlappstyle.css as shown in appendix A.

4.6 Accessing model and server-side code

The model of ProjectTimeLogger is in a remote database and when users request for remote data in a view, the corresponding controller must fetch information from the server. For instance, when a user selects a project from the list of projects, the projectdetailsCtrl requests for project details from the model by using a http post method. This method, in turn, runs a php script to fetch the required details from the database. Figure 4.14 shows the php script to fetch project details from the database.



```php
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';


$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));
try {
        $userData= array();
        $query = $con->prepare("select projectId, projectName, modifiedDate, iterationNumber, activity from projectdetails where userId=? and projectId=? order by
        modifiedDate DESC limit 1");
        $query->bindParam(1, $data->userid, PDO::PARAM_STR);
        $query->bindParam(2, $data->projectid, PDO::PARAM_STR);
        $query->execute();

        while($userRow = $query->fetch(PDO::FETCH_ASSOC))
        {
            $userData['info']=$userRow;
        }

        $query = $con->prepare("SELECT sum(hour(timeRecorded)) as hours ,sum(minute(timeRecorded)) as minutes ,sum(second(timeRecorded)) as seconds FROM
        projectdetails WHERE userId=? and projectId=?");
        $query->bindParam(1, $data->userid, PDO::PARAM_STR);
        $query->bindParam(2, $data->projectid, PDO::PARAM_STR);
        $query->execute();

        while($userRow = $query->fetch(PDO::FETCH_ASSOC))
        {
            $userData['time']=$userRow;
        }
        echo json_encode($userData);

    }
catch(PDOException $e){

        echo $e->getMessage();
}
?>
```

Figure 4.14: php script to retrieve project details of selected project

At the end of this step, we can conclude that ProjectTimeLogger is a contemporary application which distributes the computation between the client and the server such that the application utilizes the client-side computational capabilities and makes requests to remote server only to get data that is not accessible to the client at all times. Figure 4.15 shows the distribution of computation between the client and the server for ProjectTimeLogger.

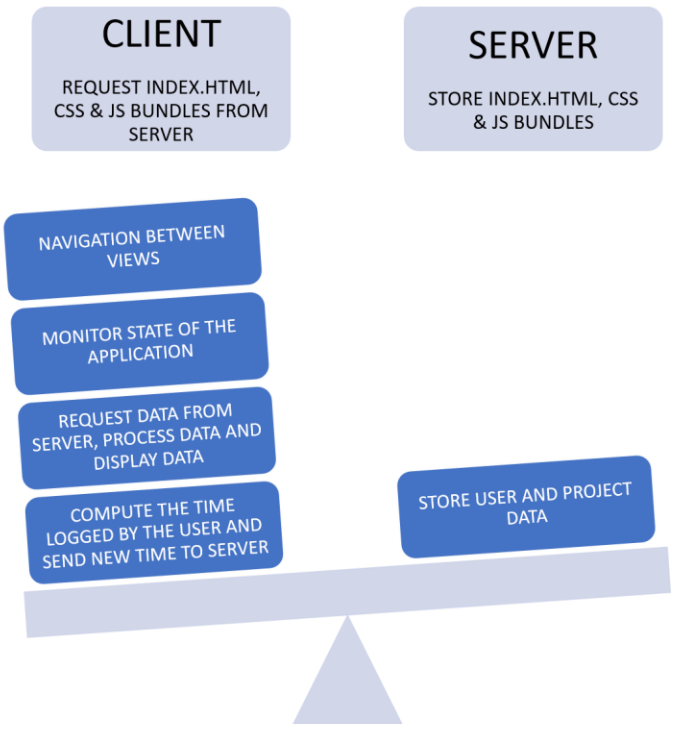

CLIENT
REQUEST INDEX.HTML, CSS & JS BUNDLES FROM SERVER
SERVER
STORE INDEX.HTML, CSS & JS BUNDLES
NAVIGATION BETWEEN VIEWS
MONITOR STATE OF THE APPLICATION
REQUEST DATA FROM SERVER, PROCESS DATA AND DISPLAY DATA
COMPUTE THE TIME LOGGED BY THE USER AND SEND NEW TIME TO SERVER
STORE USER AND PROJECT DATA

Figure 4.15: Distribution of Computation between Client and Server for ProjectTimeLogger

Future Efforts

Although the proposed design approach in this research can be used to build a single page application, there are multiple steps that can be added to the design process. The following are the important considerations to add as a step or a set of steps during the design process of building a single page application:

1. Distribution of dynamically rendered HTML content using JavaScript and using HTML snippets to determine if the apt distribution of presentation logic allow users to share more computational logic with the server.
2. An approach to find the invalid and valid states of the application, to reduce the possibilities of data leakage from the applications.
3. An approach to distribute JavaScript and CSS code in to multiple bundles, to reduce the initial page load time of the application.
4. Using session tokens with an expiring policy to add a layer of security to the application.
5. Client-side machines use the back, history and forward keys to help users with navigation in a traditional application. An approach to store the information about the visited views and keeping these views in memory for a certain period of time, so that when the back, forward or history buttons are used, the application handles the user request before trying to request data from the server.

Bibliography

Web development with Django notes Summer 2012, by Dr. David A. Umphress, Computer Science and Software Engineering Department, Auburn University.

Event driven engineering notes, by Dr. David A. Umphress, Computer Science and Software Engineering Department, Auburn University.

[Scott, 2016] "SPA Design and Architecture; Understanding Single Page Web Applications", by Emmit A. Scott, Jr., licensed to Dr. David A. Umphress, Auburn University.

[Mikowski, Powell, 2014] "Single Page Web Applications", by Michael S. Mikowski and Josh C. Powell, licensed to Dr. David A. Umphress, Auburn University.

[Takada, 2012] Single page apps in depth, http://www.icaretonometer.com/wp-content/uploads/2012/11/singlepageappbook.pdf

[Veeravalli, 2017] Measuring and Optimizing Performance of Single-Page Applications (SPA) Using RUM, https://engineering.linkedin.com/blog/2017/02/measuring-and-optimizing-performance-of-single-page-applications

[Burkholder, 2018] JavaScript SEO: Server Side Rendering vs. Client Side Rendering,

https://medium.com/@benjburkholder/javascript-seo-server-side-rendering-vs-client-side-rendering-bc06b8ca2383

[Sherman, 2018] How Single-Page Applications Work,

https://medium.com/@pshrmn/demystifying-single-page-applications-3068d0555d46

Appendix A: Source Code

1) index.html

```
<!DOCTYPE html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.5/angular.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.5/angular-
route.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

    <link rel="stylesheet" href="css/ptlappstyle.css" />
    <link rel="stylesheet" href="css/tablestyle.css" />
    <link rel="stylesheet" href="css/sidenavbar.css" />
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
 </head>

  <body ng-app="ptlapp" ng-controller="ptlappCtrl">
    <nav class="navbar navbar-inverse navbar-static-top navbarapp">
      <div class="container-fluid">
        <div class="navbar-header">
          <a class="navbar-brand">Project Time Logger</a>
        </div>
        <ul id="signoutoption" ng-hide="!getsignoutoption()" class="nav navbar-nav navbar-
right">
          <li><a href="#" ng-click="loguserout()">Sign Out <span class="glyphicon glyphicon-log-
out"></span></a></li>
        </ul>
      </div>
    </nav>

    <div ng-view></div>

    <script src="js/app.js"></script>
    <script src="js/loginCtrl.js"></script>
    <script src="js/registerCtrl.js"></script>
    <script src="js/projectlistCtrl.js"></script>
    <script src="js/addnewprojectCtrl.js"></script>
    <script src="js/projectdetailsCtrl.js"></script>
  </body>
</html>
```

2) login.html

```
<link href='https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300' rel='stylesheet'
type='text/css'>
<div class="container-fluid">
  <form class="register-form" name="loginform" ng-submit="logintoprojects()">
    <div class="row">
      <div class="col-md-4 col-sm-4 col-lg-4">
        <label for="email">EMAIL</label>
        <input name="email" id="email" ng-model="email" class="form-control"
type="email" required>
      </div>
    </div>
    <div class="row">
      <div class="col-md-4 col-sm-4 col-lg-4">
        <label for="password">PASSWORD</label>
        <input name="password" id="password" ng-model="password" class="form-control"
type="password" required>
      </div>
    </div>
    <hr>
    <div class="row">
      <div class="col-md-6 col-sm-6 col-xs-6 col-lg-6">
        <button type="submit" class="btn btn-default logbutton">Login</button>

      </div>
      <div class="col-md-6 col-sm-6 col-xs-6 col-lg-6">
        <a ng-href="#/register" class="btn btn-default regbutton">Register</a>
      </div>
    </div>
    <div id="registerstatus" class="registerstatus" ng-hide="!getregistrationstatus()">
     <div class="alert alert-success alert-dismissible">
      <strong>Success!</strong> Please login.
     </div>
    </div>
  </form>
</div>
```

3) register.html

```
<link href='https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300' rel='stylesheet'
type='text/css'>
<div class="container-fluid">
  <form class="register-form" name="registerform" ng-submit="registeruser()">
    <div class="row">
      <div class="col-md-4 col-sm-4 col-lg-4">
        <label for="username">USERNAME</label>
        <input name="username" id="username" ng-model="username"  class="form-control"
type="text" required>
      </div>
    </div>
    <div class="row">
      <div class="col-md-4 col-sm-4 col-lg-4">
        <label for="email">EMAIL</label>
        <input name="email" id="email" ng-model="email" class="form-control"
type="email" required>
      </div>
    </div>
    <div class="row">
      <div class="col-md-4 col-sm-4 col-lg-4">
        <label for="password">PASSWORD</label>
        <input name="password" id="password" ng-model="password" class="form-control"
type="password" required>
      </div>
    </div>
    <hr>
    <div class="row">
      <div class="col-md-6 col-sm-6 col-xs-6 col-lg-6">
        <button type="submit" class="btn btn-default regbutton">Register</button>
      </div>
      <div class="col-md-6 col-sm-6 col-xs-6 col-lg-6">
        <a ng-href="#/login" class="btn btn-default logbutton">Login</a>
      </div>
    </div>
    <div id="registerstatus" class="registerstatus" hidden="true">
      <div class="alert alert-danger alert-dismissable">
        <strong>Error!</strong> Please try again.
      </div>
    </div>
  </form>
</div>
```

4) projectlist.html

```
<div ng-hide="isloggedin()">
  <nav class="navbar navbar navbar-static-top navbarapp">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand">{{welcomeuser}}</a>
      </div>
      <ul class="nav navbar-nav navbar-right">
        <li><a href="#/addnewproject" ng-click="addnewproject()">Add <span class="glyphicon
glyphicon-plus-sign"></span></a></li>
      </ul>
    </div>
  </nav>
  <div ng-hide="!ifnewprojectadded()">
    <div class = "alert alert-success alert-dismissable">
      <button type="button" class="close" data-dismiss = "alert" aria-hidden =
"true">&times;</button>
      <strong>Success!</strong> Project added.
    </div>
  </div>
  <div class="wrapper">
    <table id="acrylic" style="width:85%">
      <thead>
        <tr>
          <th>Project ID</th>
          <th>Title</th>
          <th>Modified Date</th>
          <th>Current Iteration</th>
          <th>Current Activity</th>
          <th>Total time</th>
        </tr>
      </thead>
      <tbody>
        <tr ng-repeat="(key,value) in row">
          <td ng-repeat="(key,val) in value" ng-click="showprojectdetails(value)">{{val}}</td>
        </tr>
      </tbody>
    </table>
  </div>
</div>

<div class="alert alert-danger" ng-hide="!isloggedin()">
  <strong>Error!</strong> Please <a href="#">login</a>.
</div>
```

5) projectlist.html

```
<div ng-hide="isloggedin()">
  <nav class="navbar navbar navbar-static-top navbarapp">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand">{{welcomeuser}}</a>
      </div>
      <ul class="nav navbar-nav navbar-right">
        <li><a href="#/addnewproject" ng-click="addnewproject()">Add <span class="glyphicon
glyphicon-plus-sign"></span></a></li>
      </ul>
    </div>
  </nav>
  <div ng-hide="!ifnewprojectadded()">
    <div class = "alert alert-success alert-dismissable">
      <button type="button" class="close" data-dismiss = "alert" aria-hidden =
"true">&times;</button>
      <strong>Success!</strong> Project added.
    </div>
  </div>
  <div class="wrapper">
    <table id="acrylic" style="width:85%">
      <thead>
        <tr>
          <th>Project ID</th>
          <th>Title</th>
          <th>Modified Date</th>
          <th>Current Iteration</th>
          <th>Current Activity</th>
          <th>Total time</th>
        </tr>
      </thead>
      <tbody>
        <tr ng-repeat="(key,value) in row">
          <td ng-repeat="(key,val) in value" ng-click="showprojectdetails(value)">{{val}}</td>
        </tr>
      </tbody>
    </table>
  </div>
</div>

<div class="alert alert-danger" ng-hide="!isloggedin()">
  <strong>Error!</strong> Please <a href="#">login</a>.
</div>
```

6) addnewproject.html

```
<link href='https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300' rel='stylesheet'
type='text/css'>
<div class="container-fluid">
    <form class="register-form" name="addnewproj" ng-submit="addnewproject()">
        <div class="row">
            <div class="col-md-4 col-sm-4 col-lg-4" style="width:100%;">
                <label  for="email">PROJECT NAME</label>
                <input name="pname" id="pname" ng-model="pname" class="form-control"
type="text" required>
            </div>
        </div>
        <hr>
        <div class="row">
            <div class="col-md-6 col-sm-6 col-xs-6 col-lg-6">
                <button type="submit" class="btn btn-default logbutton">Add</button>
            </div>
            <div class="col-md-6 col-sm-6 col-xs-6 col-lg-6">
                <a ng-href="#/projectlist" class="btn btn-default regbutton">Cancel</a>
            </div>
        </div>
        <div class="row" style="padding: 20px 12px;">
          <div class="alert alert-danger alert-dismissable" ng-hide="ifnewprojectnotadded()">
            <strong>Error!</strong> Please try again.
          </div>
        </div>
    </form>
</div>
```

7) projectdetails.html

```
<div ng-hide="isloggedin()">
  <nav class="navbar navbar navbar-static-top navbarapp">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand">{{projectname}}</a>
      </div>
    </div>
  </nav>
  <div>
    <table style="width:100%; height:100%;">
      <tr>
        <td rowspan="2" style="width:25%;margin:2%;background-color: #EEEEEE;">
          <div>
            <ul class="ulsnb" style="padding:0px;width:max-content;margin:2%;">
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Analysis" ng-click="changeActivity('Analysis');">Analysis</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Architecture" ng-
click="changeActivity('Architecture');">Architecture</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Project Planning" ng-click="changeActivity('Project
Planning');">Project Planning</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Iteration Planning" ng-click="changeActivity('Iteration
Planning');">Iteration Planning</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Construction" ng-
click="changeActivity('Construction');">Construction</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Refactoring" ng-
click="changeActivity('Refactoring');">Refactoring</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Review" ng-click="changeActivity('Review');">Review</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Integration Test" ng-click="changeActivity('Integration
Test');">Integration Test</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Repatterning" ng-
click="changeActivity('Repatterning');">Repatterning</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Postmortem" ng-
click="changeActivity('Postmortem');">Postmortem</a></li>
              <li class="lisnb liaactivesnb lia:hover:not(.active)"><a class="asnb liaactivesnb
lia:hover:not(.active)" id="Sandbox" ng-click="changeActivity('Sandbox');">Sandbox</a></li>
```

```
      </ul>
     </div>
    </td>
    <td style="padding:10px;margin:2%;background-color: #e1e1e1;">
     <div>
      <table style="width:100%;">
       <tr>
        <td>
         <div>
          <table style="width:100%;">
           <tr>
            <td style="border: 2px solid #474747;">
             <span>
              <div class="panelheading">
               <p style="padding:10px;margin:0px;">New Time</p>
               <p ng-bind="newtime" id="newtimer" style="font-
weight:bold;padding:10px;margin:0px;">{{newtime}}</p>
              </div>
             </span>
            </td>
            <td style="border: 2px solid #474747;">
             <span>
              <div class="panelheading">
               <p style="padding: 10px;margin:0px;">Current Activity</p>
               <p ng-bind="thisAct" id="thisAct" style="font-
weight:bold;padding:10px;margin:0px;">{{thisAct}}</p>
              </div>
             </span>
            </td>
            <td style="border: 2px solid #474747;">
             <span>
              <div class="panelheading">
               <p style="padding: 10px;margin:0px;">Current Iteration</p>
               <p ng-bind="actThisIter" id="actThisIter" style="font-
weight:bold;padding:10px;margin:0px;">{{actThisIter}}</p>
              </div>
             </span>
            </td>
            <td style="border: 2px solid #474747;">
             <span>
              <div class="panelheading">
               <p style="padding: 10px;margin:0px;">Iteration</p>
               <p ng-bind="iteration" id="iteration" style="font-
weight:bold;padding:10px;margin:0px;">{{iteration}}</p>
              </div>
             </span>
            </td>
            <td style="border: 2px solid #474747;">
              <span>
                <div class="panelheading">
                  <p style="padding: 10px;margin:0px;">Current Project</p>
                  <p ng-bind="totalTime" id="totalTime" style="font-
weight:bold;padding:10px;margin:0px;">{{totalTime}}</p>
                </div>
              </span>
            </td>
          </tr>
        </table>
      </div>
    </td>
  </tr>
</table>
</div>
</td>
</tr>
<tr>
  <td style="padding:0px;background-color: #d4d4d4;">
    <div>
      <table style="width:100%;">
        <tr>
          <td align="center">
            <input type="button" id="starttimerbutton"  value="START" ng-
click="startTimers();" />
          </td>
          <td align="center">
            <input type="button" id="starttimerbutton"  value="STOP" ng-click="stopTimers();"
/>
          </td>
          <td align="center">
            <input type="button" id="starttimerbutton"  value="RESET" ng-
click="resetTimer();" />
          </td>
        </tr>
        <tr>
          <td align="center" colspan="3" style="padding:10px;">
            <div>
              <button class="collapsible">Enter Time Manually</button>
              <div class="content" style="height:45px">
                <input style="margin-top: 10px;margin-bottom: 10px;" type="text"
id="manualTime" placeholder="HH:MM" />
              </div>
            </div>
```

```
                </td>
              </tr>
              <tr>
                <td align="center" colspan="3" style="padding:10px;">
                  <div>
                    <input type="button" id="starttimerbutton"  value="LOG TIME" ng-
click="logTime();" />
                  </div>
                </td>
              </tr>
            </table>
          </div>
        </td>
      </tr>
      <tr>
        <td colspan="2" align="center" style="padding:10px;background-color: #c7c7c7;">
          <div>
            <button class="collapsibleproject">Project Options</button>
            <div class="contentproject" style="height:45px">
              <input type="button" id="iterationcomplete"  value="ITERATION COMPLETE" ng-
click="iterationcomplete();" />
              <input type="button" id="projectcomplete"  value="PROJECT COMPLETE" ng-
click="projectcomplete();" />
            </div>
          </div>
        </td>
      </tr>
    </table>
  </div>
  <!-- <div id="stripH" class="stripH">
      <div ng-bind="newtime" id="timerlabel" class="timerlabel">{{newtime}}</div>
  </div> -->

</div>
<div class="alert alert-danger" ng-hide="!isloggedin()">
  <strong>Error!</strong> Please <a href="#">login</a>.
</div>
```

8) ptlappstyle.css

```
body, html{
   font-family: 'Open Sans Condensed', sans-serif;
    font-size: 18px;
}

.register-form{
    font-size: 16px;
   left: 50%;
    top: 50%;
    position: absolute;
    -webkit-transform: translate3d(-50%, -50%, 0);
    -moz-transform: translate3d(-50%, -50%, 0);
    transform: translate3d(-50%, -50%, 0);
}

.register-form input{
    margin-bottom: 5px;
    width: 430px;
    height: 40px;
    border-radius: 0px;
}

.register-form label{
    color: #2269a7;
}

.regbutton{
  width: 200px;
}
.logbutton{
  width: 200px;
}

.registerstatus{
  padding-top: 40px;
  text-align: center;
}

.navbarapp{
  margin-bottom: 0px !important;
}

.stripH{
  height: 100px;
  background-color: rgba(0,0,0,.065);
}
.timerlabel{
  position: absolute;
  left: 30px;
  line-height: 1.928571;
  font-size: -webkit-xxx-large;
}

.stripV{
  position:absolute;
  width: 150px;
  height: 200%;
  background-color: rgba(0,0,0,.065);
  left: 100px;
  bottom: 0;
}

.newTimer{
  font-size: -webkit-xxx-large;
}

.content {
    padding: 0 18px;
    display: none;
    overflow: hidden;
    background-color: #f1f1f1;
}

.active, .collapsible:hover {
    background-color: #777;
}

.collapsible {
    background-color: #555;
    color: white;
    cursor: pointer;
    padding: 18px;
    width: 100%;
    border: none;
    text-align: center;
    outline: none;
    font-size: 15px;
}
```

```
.contentproject {
   padding: 0 18px;
   display: none;
   overflow: hidden;
   background-color: #f1f1f1;
}

.active, .collapsibleproject:hover {
   background-color: #777;
}

.collapsibleproject {
   background-color: #555;
   color: white;
   cursor: pointer;
   padding: 18px;
   width: 100%;
   border: none;
   text-align: center;
   outline: none;
   font-size: 15px;
}

.panelheading{
   text-align:center;
   font-size: 16px;
}
```

9) sidenavbar.css

```
body {
   margin: 0;
}

.ulsnb {
   list-style-type: none;
   margin: 0;
   padding: 0;
   width: 15%;
   height: max-content;
   overflow: auto;
}

.asnb {
   display: block;
   color: #000;
   padding: 0px;
   text-decoration: none;
}

.lisnb {
   display: block;
   color: #000;
   padding: 8px;
   text-decoration: none;
}

.liaactivesnb {
   text-decoration: none;
}

.lia:hover:not(.active) {
   background-color: #555;
   color: white;
   text-decoration: none;
}
```

10) sidenavbar.css

```
table#acrylic {
            border-collapse: separate;
            background: #fff;
            -moz-border-radius: 5px;
            -webkit-border-radius: 5px;
            border-radius: 5px;
            margin: 50px auto;
            -moz-box-shadow: 0px 0px 5px rgba(0, 0, 0, 0.3);
            -webkit-box-shadow: 0px 0px 5px rgba(0, 0, 0, 0.3);
            box-shadow: 0px 0px 5px rgba(0, 0, 0, 0.3);
        }

        #acrylic thead {
            -moz-border-radius: 5px;
            -webkit-border-radius: 5px;
            border-radius: 5px;
            cursor: text;
        }

        #acrylic thead th {
            font-family: 'Roboto';
            font-size: 16px;
            font-weight: 400;
            color: #fff;
            text-shadow: 1px 1px 0px rgba(0, 0, 0, 0.5);
            text-align: left;
            padding: 20px;
            background-size: 100%;
            background-image: -webkit-gradient(linear, 50% 0%, 50% 100%, color-stop(0%,
#646f7f), color-stop(100%, #4a5564));
            background-image: -moz-linear-gradient(#646f7f, #4a5564);
            background-image: -webkit-linear-gradient(#646f7f, #4a5564);
            background-image: linear-gradient(#646f7f, #4a5564);
            border-top: 1px solid #858d99;
        }

        #acrylic thead th:first-child {
            -moz-border-top-right-radius: 5px;
            -webkit-border-top-left-radius: 5px;
            border-top-left-radius: 5px;
        }

        #acrylic thead th:last-child {
            -moz-border-top-right-radius: 5px;
```

```
    -webkit-border-top-right-radius: 5px;
    border-top-right-radius: 5px;
}

#acrylic tbody tr td {
    font-family: 'Open Sans', sans-serif;
    font-weight: 400;
    color: #5f6062;
    font-size: 13px;
    padding: 20px 20px 20px 20px;
    border-bottom: 1px solid #e0e0e0;
    cursor: pointer;
}

#acrylic tbody tr:nth-child(2n) {
    background: #f0f3f5;
}

#acrylic tbody tr:last-child td {
    border-bottom: none;
}

#acrylic tbody tr:last-child td:first-child {
    -moz-border-bottom-right-radius: 5px;
    -webkit-border-bottom-left-radius: 5px;
    border-bottom-left-radius: 5px;
}

#acrylic tbody tr:last-child td:last-child {
    -moz-border-bottom-right-radius: 5px;
    -webkit-border-bottom-right-radius: 5px;
    border-bottom-right-radius: 5px;
}

#acrylic tbody:hover > tr td {
    filter: progid: DXImageTransform.Microsoft.Alpha(Opacity=50);
    opacity: 0.5;
}

#acrylic tbody:hover > tr:hover td {
    text-shadow: none;
    color: #2d2d2d;
    filter: progid: DXImageTransform.Microsoft.Alpha(enabled=false);
    opacity: 1;
    transition: 0.2s all;
}
```

11) app.js

```
var ptlMod = angular.module('ptlapp', ['ngRoute']);
ptlMod.config(function ($routeProvider, $locationProvider) {
  $routeProvider
    .when('/login', { templateUrl: 'templates/login.html', controller: 'loginCtrl' })
    .when('/register', { templateUrl: 'templates/register.html', controller: 'registerCtrl' })
    .when('/projectlist', { templateUrl: 'templates/projectlist.html', controller: 'projectlistCtrl' })
    .when('/addnewproject', { templateUrl: 'templates/addnewproject.html', controller:
'addnewprojectCtrl' })
    .when('/projectdetails', { templateUrl: 'templates/projectdetails.html', controller:
'projectdetailsCtrl' })
    .otherwise({ redirectTo: "/login" });
});

ptlMod.controller('ptlappCtrl', function ($scope,$window,$http,userServices){
  $scope.getsignoutoption = function(){
    return (Boolean(userServices.getloginstatus()));
  };
  $scope.loguserout = function(){
    userServices.setloginstatus(false);
    userServices.setregisterstatus(false);
    userServices.setuserid(null);
    userServices.setusername("");
    $window.location="#/login";
  };
});

ptlMod.service('userServices',function(){
  this.setregisterstatus=function(status){
        this.registerstatus = status;
    };
            this.getregisterstatus=function(){
        return this.registerstatus;
    };
  this.setloginstatus=function(status){
        this.loginstatus = status;
    };
            this.getloginstatus=function(){
        return this.loginstatus;
    };
  this.setuserid = function(userid){
    this.userid = userid;
  };
  this.getuserid = function(){
    return this.userid;
```

```
};
this.setusername = function(username){
  this.username = username;
};
this.getusername = function(){
  return this.username;
};
this.setNewProjectAdded = function(status){
  this.NewProjectAdded = status;
};
this.getNewProjectAdded = function(){
  return this.NewProjectAdded;
};
this.gettimeinHMS = function(timerecorded){
  if(timerecorded[2]>59){
                                var carryover = timerecorded[2]/60;
                                timerecorded[2] = timerecorded[2]%60;
                                carryover = String(carryover).split(".");
                                timerecorded[1] = +timerecorded[1] + +carryover[0];
                        }
                        if(timerecorded[1]>59){
                                var carryover = timerecorded[1]/60;
                                timerecorded[1] = timerecorded[1]%60;
                                carryover = String(carryover).split(".");
                                timerecorded[0] = +timerecorded[0] + +carryover[0];
                        }
  this.timeinHMS = timerecorded[0] +":"+ timerecorded[1] +":"+ timerecorded[2];
  return this.timeinHMS;
};
this.settotaltimerecorded = function(timerecorded){

};
this.setprojectname = function(projectname){
  this.projectname = projectname;
};
this.getprojectname = function(){
  return this.projectname;
};
this.setprojectid = function(projectid){
  this.projectid = projectid;
};
this.getprojectid = function(){
  return this.projectid;
};
this.settotaltime = function(totaltime){
  this.totaltime = totaltime;
  };
  this.gettotaltime = function(){
    return this.totaltime;
  };
  this.setiteration = function(iteration){
    this.iteration = iteration;
  };
  this.getiteration = function(){
    return this.iteration;
  };
  this.setactivity = function(activity){
    this.activity = activity;
  };
  this.getactivity = function(){
    return this.activity;
  };
});
```

12) loginCtrl.js

```
ptlMod.controller('loginCtrl', function ($scope,$window,$http,userServices){
  $scope.getregistrationstatus = function(){
    return (Boolean(userServices.getregisterstatus()));
  };
  $scope.logintoprojects = function(){
			if(($scope.loginform.email.$valid)&&($scope.loginform.password.$valid))
			{
				$http.post("php/login.php",
				{
					'email': $scope.email,
					'password': $scope.password
				})
				.then(function(result){
					if(result.data.username===undefined)
					{
						alert("Wrong Username or Password, Please try again!");
						$window.location="#!/login";
					}
					else
					{
						userServices.setloginstatus('true');
	userServices.setuserid(result.data.userId);
	userServices.setusername(result.data.username);
						$window.location="#/projectlist";
					}
```

```
                        },
                                function(error){
        userServices.setloginstatus('false');
        alert("Please try again!");
                                 console.error(error);
                        });
                }
          };
});
```

13) registerCtrl.js

```
ptlMod.controller('registerCtrl', function ($scope,$window,$http,userServices){
  $scope.registeruser = function(){

if(($scope.registerform.email.$valid)&&($scope.registerform.username.$valid)&&($scope.regist
erform.password.$valid)){
                                    $http.post("php/register.php",
                                    {
                                                'username': $scope.username,
                                                'email': $scope.email,
                                                'password': $scope.password
                                    })
                                    .then(function () {
                                                console.log("User registered.");
        userServices.setregisterstatus(true);
                                                $window.location="#/login";
                                    },
            function(error){
        userServices.setregisterstatus(false);
                        console.error(error);
            });
                        }
  };
});
```

14) projectlistCtrl.js

```
ptlMod.controller('projectlistCtrl', function ($scope,$window,$http,userServices){
  var usernumber = userServices.getuserid();
  $scope.row = [];
  $scope.isloggedin = function(){
    if(usernumber == null){
      userServices.setNewProjectAdded(false);
      return true;
    }
    else{
      var username = userServices.getusername();
      $scope.welcomeuser = "Welcome, " + username +".";
      return false;
    }
  };
  $scope.addnewproject = function(){
    $window.location="#/addnewproject";
  };
  $scope.ifnewprojectadded = function(){
    return (Boolean(userServices.getNewProjectAdded()));
  };

  $scope.showprojectdetails = function(value){
    //console.log(value);
    //console.log(value[0]);
    userServices.setprojectname(value[1]);
    userServices.setprojectid(value[0]);
    userServices.settotaltime(value[5]);
    userServices.setiteration(value[3]);
    userServices.setactivity(value[4]);
    $window.location="#/projectdetails";
  };

  $scope.projectids =[];
  $http.post("php/getprojectids.php",
          {
                    'userid': usernumber
          })
          .then(function(result){
                    $scope.projectids = result.data;
                    console.log($scope.projectids);
      for(var i=0; i<$scope.projectids.length; i++)
      {
        //console.log(projectids[i].projectId);
        var eachprojectid = $scope.projectids[i].projectId;
        $http.post("php/getprojectinfo.php",
                        {
                            'userid': usernumber,
                            'projectid': eachprojectid
                        })
        .then(function(result){
          var projectdetails = result.data;
          console.log(projectdetails);
          var rowdata = [];
          rowdata[0] = result.data.info.projectId;
          rowdata[1] = result.data.info.projectName;
                            rowdata[2] = result.data.info.modifiedDate;
                            rowdata[3] = result.data.info.iterationNumber;
                            rowdata[4] = result.data.info.activity;
          var timerecorded = [];
                            timerecorded[0] = result.data.time.hours;
                            timerecorded[1] = result.data.time.minutes;
                            timerecorded[2] = result.data.time.seconds;
                            rowdata[5] = userServices.gettimeinHMS(timerecorded);
          $scope.row.push(rowdata);
        },
        function(error){
                            console.error(error);
                        });
      }
          },
          function(error){
                console.error(error);
          });

    });
```

15) addnewprojectCtrl.js

```
ptlMod.controller('addnewprojectCtrl', function ($scope,$window,$http,userServices){
  console.log(userServices.getNewProjectAdded());
  $scope.addnewproject = function(){
    var userid = userServices.getuserid();
    $http.post("php/addnewproject.php",
                        {
                                    'pname': $scope.pname,
                                    'userid': userid
                        })
                        .then(function(result){
                                    console.log("Project Added.");
      userServices.setNewProjectAdded(true);
                                    $window.location="#/projectlist";
                        },
                        function(error){
      console.log("error function");
      userServices.setNewProjectAdded(false);
      $scope.noerrorvariable = false;
                                    console.error(error);
                        });
  };
  $scope.noerrorvariable = true;
  $scope.ifnewprojectnotadded = function(){
    return $scope.noerrorvariable;
  };
});
```

16) projectdetails.js

```
//make the background of the timers "#d0d9de" colors
ptlMod.controller('projectdetailsCtrl', function
($scope,$interval,$filter,$window,$http,userServices){

  $scope.iteration=0;
  $scope.getinitialvalues = function(){
                    var activityname = userServices.getactivity();
                    var selectedact = document.getElementById(activityname);
                    selectedact.style.backgroundColor = "#bdd6ea";
                    iterationnumber = userServices.getiteration();
          };

  $scope.manualTime=0;
  var coll = document.getElementsByClassName("collapsible");
  var i;
  for (i = 0; i < coll.length; i++) {
    coll[i].addEventListener("click", function() {
      this.classList.toggle("active");
      var content = this.nextElementSibling;
      if (content.style.display === "block") {
        content.style.display = "none";
        $scope.manualTime=0;
      }
      else {
      content.style.display = "block";
      $scope.manualTime=1;
      }
    });
  }
  var coll = document.getElementsByClassName("collapsibleproject");
  var j;
  for (j = 0; j < coll.length; j++) {
    coll[j].addEventListener("click", function() {
      this.classList.toggle("active");
      var content = this.nextElementSibling;
      if (content.style.display === "block") {
        content.style.display = "none";
        $scope.manualTime=0;
      }
      else {
      content.style.display = "block";
      $scope.manualTime=1;
      }
    });
```

```
}
//var usernumber = userServices.getuserid();
$scope.isloggedin = function(){
  var usernumber = userServices.getuserid();
  if(usernumber == null){
    return true;
  }
  else{
    var project = userServices.getprojectname();
    //console.log(project);
    $scope.projectname = "You are currently working on: " + project +".";
    return false;
  }
};

$scope.newtime = 00 + ":" + 00 + ":" + 00;
var iterationnumber = userServices.getiteration();
$scope.iteration = iterationnumber;
var activityname = userServices.getactivity();
var selectedact = document.getElementById(activityname);
        selectedact.style.backgroundColor = "#bdd6ea";
var userid = userServices.getuserid();
var projectid = userServices.getprojectid();

$http.post("php/gettotalprojecttime.php",
        {
                'userid': userid,
                'projectid': projectid
        })
        .then(function(result){
                var timerec = [];
                timerec[0] = result.data[0].hours;
                timerec[1] = result.data[0].minutes;
                timerec[2] = result.data[0].seconds;
  console.log(timerec);
  $scope.totalTimeOld = timerec;
  tph = tph+ +timerec[0]; tpm= tpm+ +timerec[1]; tps= tps+ +timerec[2];
        $scope.totalTime = tph + ":" + tpm + ":" + tps;
        },
        function(error){
                console.error(error);
        });



$http.post("php/getcurrentiterationtime.php",
        {
                'iterationnumber': iterationnumber,
                'userid': userid,
                'projectid': projectid
        })
        .then(function(result){
                /* console.log(result);
                console.log(result.data[0]); */
                var timerec = [];
                timerec[0] = result.data[0].hours;
                timerec[1] = result.data[0].minutes;
                timerec[2] = result.data[0].seconds;
  console.log(timerec);
  $scope.actThisIterOld = timerec;
  ith = ith+ +timerec[0]; itm= itm+ +timerec[1]; its= its+ +timerec[2];
  $scope.actThisIter = ith + ":" + itm + ":" + its;
        },
        function(error){
                console.error(error);
        });

$scope.thisAct = 00 + ":" + 00 + ":" + 00;
$scope.thisActOld = 00 + ":" + 00 + ":" + 00;
$http.post("php/getcurrentactivitytime.php",
        {
                'activity': activityname,
                'userid': userid,
                'projectid': projectid
        })
        .then(function(result){
                /* console.log(result);
                console.log(result.data[0]); */
                var timerec = [];
                timerec[0] = result.data[0].hours;
                timerec[1] = result.data[0].minutes;
                timerec[2] = result.data[0].seconds;
  console.log(timerec);
  $scope.thisActOld = timerec;
  tah=tah+ +timerec[0]; tam=tam+ +timerec[1]; tas=tas+ +timerec[2];
  $scope.thisAct = tah + ":" + tam + ":" + tas;
  // $scope.thisActOld = $scope.thisAct;
        },
        function(error){
                console.error(error);
        });
```

```
$scope.changeActivity = function(activityname){
  var oldactivity = userServices.getactivity();
  var selectedact = document.getElementById(oldactivity);
          selectedact.style.backgroundColor = "transparent";
  selectedact = document.getElementById(activityname);
  selectedact.style.backgroundColor = "#bdd6ea";
  userServices.setactivity(activityname);
  var userid = userServices.getuserid();
  var projectid = userServices.getprojectid();
  var pname = userServices.getprojectname();
  var iterationnumber = userServices.getiteration();
  var timeRecorded = 0 + ":" + 0 + ":" + 0;
  $http.post("php/addactivitychange.php",
          {
    'projectid': projectid,
    'userid': userid,
    'pname': pname,
    'iterationnumber': iterationnumber,
    'activity': activityname,
    'timeRecorded': timeRecorded
          })
          .then(function(){
    console.log("addednewactivitychange");
    $scope.getinitialvalues();
          },
          function(error){
                  console.error(error);
          });

};

$scope.Timer1 = null;
          $scope.Timer2 = null;
          $scope.Timer3 = null;
          $scope.Timer4 = null;
          $scope.timers = false;
var newh=0; var newm=0; var news=0;
          var tah=0; var tam=0; var tas=0;
          var ith=0; var itm=0; var its=0;
          var tph=0; var tpm=0; var tps=0;


$scope.startTimers = function(){
  $scope.timers = true;
  $scope.startnewtime();
```

```
    $scope.startthisacttime();
    $scope.startthisitertime();
    $scope.startTotProjTime();
  };
  $scope.stopTimers = function(){
      if (angular.isDefined($scope.Timer1)) {
         $interval.cancel($scope.Timer1);
      }
          if (angular.isDefined($scope.Timer2)) {
         $interval.cancel($scope.Timer2);
      }
          if (angular.isDefined($scope.Timer3)) {
         $interval.cancel($scope.Timer3);
      }
          if (angular.isDefined($scope.Timer4)) {
         $interval.cancel($scope.Timer4);
      }
  };
  $scope.resetTimer = function(){
    if (angular.isDefined($scope.Timer1)) {
         $interval.cancel($scope.Timer1);
                        }
                        if (angular.isDefined($scope.Timer2)) {
                                $interval.cancel($scope.Timer2);
                        }
                        if (angular.isDefined($scope.Timer3)) {
                                $interval.cancel($scope.Timer3);
                        }
                        if (angular.isDefined($scope.Timer4)) {
                                $interval.cancel($scope.Timer4);
                        }
    newh = 0; newm = 0; news = 0;
    tah=0;  tam=0;  tas=0;
    tah=tah+ +$scope.thisActOld[0]; tam=tam+ +$scope.thisActOld[1]; tas=tas+
+$scope.thisActOld[2];
    ith=0;  itm=0;  its=0;
    ith = ith+ +$scope.actThisIterOld[0]; itm= itm+ +$scope.actThisIterOld[1]; its= its+
+$scope.actThisIterOld[2];
    tph=0;  tpm=0;  tps=0;
    tph = tph+ +$scope.totalTimeOld[0]; tpm= tpm+ +$scope.totalTimeOld[1]; tps= tps+
+$scope.totalTimeOld[2];
                        $scope.newtime = newh + ":" + newm + ":" + news;
    $scope.thisAct = tah + ":" + tam + ":" + tas;
    $scope.totalTime = tph + ":" + tpm + ":" + tps;
    $scope.actThisIter = ith + ":" + itm + ":" + its;
  };
```

```
$scope.startnewtime = function () {
      $scope.newtime = newh + ":" + newm + ":" + news;
      $scope.Timer1 = $interval(function () {
          $scope.newtime = newh + ":" + newm + ":" + news;
          if(news<59)
          {
              news= news + 1;
          }
          else
          {
              news = 0;
              if(newm<59)
              {
                  newm= newm + 1;
              }
              else
              {
                  newm = 0;
                  newh= newh + 1;
              }
          }
      }, 1000);
  };

  $scope.startthisacttime = function () {
      $scope.thisAct = tah + ":" + tam + ":" + tas;
      $scope.Timer2 = $interval(function () {
          $scope.thisAct = tah + ":" + tam + ":" + tas;
          if(tas<59)
          {
              tas= tas + 1;
          }
          else
          {
              tas = 0;
              if(tam<59)
              {
                  tam= tam + 1;
              }
              else
              {
                  tam = 0;
                  tah= tah + 1;
              }
          }
```

```
    }, 1000);
};

$scope.actThisIter = 00 + ":" + 00 + ":" + 00;
$scope.startthisitertime = function () {
    $scope.actThisIter = ith + ":" + itm + ":" + its;
    $scope.Timer3 = $interval(function () {
        $scope.actThisIter = ith + ":" + itm + ":" + its;
        if(its<59)
        {
            its= its + 1;
        }
        else
        {
            its = 0;
            if(itm<59)
            {
                itm= itm + 1;
            }
            else
            {
                itm = 0;
                ith= ith + 1;
            }
        }
    }, 1000);
};

$scope.totalTime = 00 + ":" + 00 + ":" + 00;
$scope.startTotProjTime = function () {
    $scope.totalTime = tph + ":" + tpm + ":" + tps;
    $scope.Timer4 = $interval(function () {
        $scope.totalTime = tph + ":" + tpm + ":" + tps;
        if(tps<59)
        {
            tps= tps + 1;
        }
        else
        {
            totprojtime[2] = 0;
            if(tpm<59)
            {
                tpm= tpm + 1;
            }
            else
            {
                tpm = 0;
                tph= tph + 1;
            }
        }
    }, 1000);
};

$scope.logTime = function(){
  var selectedtime=0;
  if($scope.manualTime==0){
    var thistime = $scope.newtime;
    thistime = $scope.newtime.split(":");
    selectedtime = thistime[0] + ":" +thistime[1] + ":" +thistime[2];
  }
  else {
    var timeInHHMM = document.getElementById("manualTime").value;
    if (timeInHHMM.length < 5) {
        window.alert("Enter time in HH:MM format");
    }
    else{
      var times = timeInHHMM.split(":");
      if (times.length != 2) {
          window.alert("Enter time in HH:MM format");
      }
      else{
        // if(!((typeof times[0] === "number")&&(typeof times[1] === "number"))){
        //   window.alert("Enter time in HH:MM format");
        // }
        // else
        {
          if (times[1] > 60){
            var carryover = (times[1] / 60);
            carryover = String(carryover).split(".");
            var minutes = times[1] % 60;
            times[0] = +times[0] + +carryover[0];
            times[1] = minutes;
            selectedtime = times[0] + ":" + times[1];
          }
        }
      }
    }
  }
  var activityname1 = userServices.getactivity;
  console.log(activityname1);
  var userid1 = userServices.getuserid;
  console.log(userid1);
```

```
   var projectid1 = userServices.getprojectid;
   console.log(projectid1);
   var projectname1 = userServices.getprojectname;
   console.log(projectname1);
   var iterationnumber1 = userServices.getiteration;
   console.log(iterationnumber1);
   console.log(selectedtime);
   $http.post("php/lognewtime.php",
   {
                    'activityname': activityname1,
                    'userid': userid1,
                    'projectid': projectid1,
                    'projectname': projectname1,
                    'iterationnumber': iterationnumber1,
                    'timerecorded': selectedtime
            })
            .then(function(){
                    console.log("logged new time");
                    //change to next activity
            });
  };

$scope.iterationcomplete = function(){
 var iterationold = userServices.getiteration();
 var iterationnew = 1 + (+ iterationold);
 userServices.setiteration(iterationnew);
 var userid = userServices.getuserid();
 var projectid = userServices.getprojectid();
 var pname = userServices.getprojectname();
 var iterationnumber = userServices.getiteration();
 var timeRecorded = 0 + ":" + 0 + ":" + 0;
 $http.post("php/additeration.php",
 {
   'projectid': projectid,
   'userid': userid,
   'pname': pname,
   'iterationnumber': iterationnumber,
   'activity': 'Analysis',
   'timeRecorded': timeRecorded
 })
 .then(function(){
   console.log("addednewiteration");
   $scope.getinitialvalues();
 },
 function(error){
   console.error(error);
 });
};

$scope.projectcomplete = function(){
 var userid = userServices.getuserid();
 var projectid = userServices.getprojectid();
 $http.post("php/markprojectcomplete.php",
 {
   'projectid': projectid,
   'userid': userid,
 })
 .then(function(){
   console.log("projectmarkedcomplete");
 },
 function(error){
   console.error(error);
 });
};

});
```

17) login.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';



$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));

try {
        $query = $con->prepare("SELECT * FROM user WHERE email=:email");
        $query->bindParam(':email', $data->email, PDO::PARAM_STR);
        $query->execute();
        $userRow = $query->fetch(PDO::FETCH_ASSOC);
        if ($query->rowCount() > 0) {
            if (strcmp($data->password, $userRow['password'])==0) {
                echo json_encode($userRow);
            }

        }

    }
catch(PDOException $e){

        echo $e->getMessage();
}
?>
```

18) register.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';


$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));

$query = $con->prepare("INSERT INTO user(username,email,password) VALUES(?,?,?)");

$query->bindParam(1,$data->username,PDO::PARAM_STR);
$query->bindParam(2,$data->email,PDO::PARAM_STR);
$query->bindParam(3,$data->password,PDO::PARAM_STR);
$query->execute();

?>
```

19) getprojectids.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';



$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));
try {
			$userData= array();
			$query = $con->prepare("SELECT projectId FROM `project` WHERE 
userId=:userid");
			$query->bindParam(':userid', $data->userid, PDO::PARAM_STR);
			$query->execute();
			while($userRow = $query->fetch(PDO::FETCH_ASSOC))
			{
				$userData[]=$userRow;
			}
	echo json_encode($userData);
   }
catch(PDOException $e){

	echo $e->getMessage();
}
?>
```

20) getprojectinfo.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';



$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));
try {
			$userData= array();
			$query = $con->prepare("select projectId, projectName, modifiedDate,
iterationNumber, activity from projectdetails where userId=? and projectId=? order by
modifiedDate DESC limit 1");
			$query->bindParam(1, $data->userid, PDO::PARAM_STR);
			$query->bindParam(2, $data->projectid, PDO::PARAM_STR);
			$query->execute();

			while($userRow = $query->fetch(PDO::FETCH_ASSOC))
			{
				$userData['info']=$userRow;
			}

			$query = $con->prepare("SELECT sum(hour(timeRecorded)) as hours
,sum(minute(timeRecorded)) as minutes ,sum(second(timeRecorded)) as seconds FROM
projectdetails WHERE userId=? and projectId=?");
			$query->bindParam(1, $data->userid, PDO::PARAM_STR);
			$query->bindParam(2, $data->projectid, PDO::PARAM_STR);
			$query->execute();

			while($userRow = $query->fetch(PDO::FETCH_ASSOC))
			{
				$userData['time']=$userRow;
			}
        echo json_encode($userData);

    }
catch(PDOException $e){

        echo $e->getMessage();
}
?>
```

21) addnewproject.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';


$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));

$query = $con->prepare("INSERT INTO project(userId,projectId,projectName) VALUES(?,
NULL, ?)");
$query->bindParam(1,$data->userid,PDO::PARAM_STR);
$query->bindParam(2,$data->pname,PDO::PARAM_STR);
$query->execute();

?>
```

22) addnewprojectdetails.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';


$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));

$query = $con->prepare("INSERT INTO
projectdetails(logId,projectId,userId,projectName,modifiedDate,iterationNumber,activity,timeRe
corded) VALUES(NULL, NULL,?,?,SYSDATE(),1,?,?)");
$query->bindParam(1,$data->userid,PDO::PARAM_STR);
$query->bindParam(2,$data->pname,PDO::PARAM_STR);
$query->bindParam(3,$data->activity,PDO::PARAM_STR);
$query->bindParam(4,$data->timeRecorded,PDO::PARAM_STR);
$query->execute();

?>
```

23) gettotalprojecttime.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';



$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));
try {
			$userData= array();
			$query = $con->prepare("SELECT sum(hour(timeRecorded)) as hours
,sum(minute(timeRecorded)) as minutes ,sum(second(timeRecorded)) as seconds FROM
`projectdetails` where userId=? and projectId=?");
			$query->bindParam(1, $data->userid, PDO::PARAM_STR);
			$query->bindParam(2, $data->projectid, PDO::PARAM_STR);
			$query->execute();
			while($userRow = $query->fetch(PDO::FETCH_ASSOC))
			{
				$userData[]=$userRow;
			}
        echo json_encode($userData);
    }
catch(PDOException $e){

        echo $e->getMessage();
}
?>
```

24) gettotalprojecttime.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';



$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));
try {
                    $userData= array();
                    $query = $con->prepare("SELECT sum(hour(timeRecorded)) as hours
,sum(minute(timeRecorded)) as minutes ,sum(second(timeRecorded)) as seconds FROM
`projectdetails` where userId=? and projectId=? and iterationNumber=?");
                    $query->bindParam(1, $data->userid, PDO::PARAM_STR);
                    $query->bindParam(2, $data->projectid, PDO::PARAM_STR);
                    $query->bindParam(3, $data->iterationnumber, PDO::PARAM_STR);
                    $query->execute();
                    while($userRow = $query->fetch(PDO::FETCH_ASSOC))
                    {
                              $userData[]=$userRow;
                    }
      echo json_encode($userData);
   }
catch(PDOException $e){

      echo $e->getMessage();
}
?>
```

25) getcurrentiterationtime.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';



$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));
try {
			$userData= array();
			$query = $con->prepare("SELECT sum(hour(timeRecorded)) as hours
,sum(minute(timeRecorded)) as minutes ,sum(second(timeRecorded)) as seconds FROM
`projectdetails` where userId=? and projectId=? and iterationNumber=?");
			$query->bindParam(1, $data->userid, PDO::PARAM_STR);
			$query->bindParam(2, $data->projectid, PDO::PARAM_STR);
			$query->bindParam(3, $data->iterationnumber, PDO::PARAM_STR);
			$query->execute();
			while($userRow = $query->fetch(PDO::FETCH_ASSOC))
			{
				$userData[]=$userRow;
			}
	echo json_encode($userData);
   }
catch(PDOException $e){

	echo $e->getMessage();
}
?>
```

26) getcurrentactivitytime.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';



$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));
try {
			$userData= array();
			$query = $con->prepare("SELECT sum(hour(timeRecorded)) as hours
,sum(minute(timeRecorded)) as minutes ,sum(second(timeRecorded)) as seconds FROM
`projectdetails` where userId=? and projectId=? and activity=?");
			$query->bindParam(1, $data->userid, PDO::PARAM_STR);
			$query->bindParam(2, $data->projectid, PDO::PARAM_STR);
			$query->bindParam(3, $data->activity, PDO::PARAM_STR);
			$query->execute();
			while($userRow = $query->fetch(PDO::FETCH_ASSOC))
			{
				$userData[]=$userRow;
			}
        echo json_encode($userData);
   }
catch(PDOException $e){

        echo $e->getMessage();
}
?>
```

27) addactivitychange.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';


$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));

$query = $con->prepare("INSERT INTO
projectdetails(logId,projectId,userId,projectName,modifiedDate,iterationNumber,activity,timeRe
corded) VALUES(NULL,?,?,?,SYSDATE(),?,?,?)");
$query->bindParam(1,$data->projectid,PDO::PARAM_STR);
$query->bindParam(2,$data->userid,PDO::PARAM_STR);
$query->bindParam(3,$data->pname,PDO::PARAM_STR);
$query->bindParam(4,$data->iterationnumber,PDO::PARAM_STR);
$query->bindParam(5,$data->activity,PDO::PARAM_STR);
$query->bindParam(6,$data->timeRecorded,PDO::PARAM_STR);
$query->execute();

?>
```

28) addactivitychange.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';


$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));

$query = $con->prepare("INSERT INTO
projectdetails(logId,projectId,userId,projectName,modifiedDate,iterationNumber,activity,timeRe
corded) VALUES(NULL,?,?,?,SYSDATE(),?,?,?)");
$query->bindParam(1,$data->projectid,PDO::PARAM_STR);
$query->bindParam(2,$data->userid,PDO::PARAM_STR);
$query->bindParam(3,$data->pname,PDO::PARAM_STR);
```

```
$query->bindParam(4,$data->iterationnumber,PDO::PARAM_STR);
$query->bindParam(5,$data->activity,PDO::PARAM_STR);
$query->bindParam(6,$data->timeRecorded,PDO::PARAM_STR);
$query->execute();

?>
```

29) markprojectcomplete.php

```
<?php

$dsn = 'mysql:host=localhost;dbname=projecttimelogger';
$username = 'root';
$password = '';


$con = new PDO($dsn, $username, $password);
$data = json_decode(file_get_contents("php://input",true));

$query = $con->prepare("UPDATE project set isCompleted = 1 where userId =? and projectId =
?");
$query->bindParam(1,$data->userid,PDO::PARAM_STR);
$query->bindParam(2,$data->projectid,PDO::PARAM_STR);
$query->execute();

?>
```