Project Management System
Design Document

Charles Pascoe

April 6, 2017

Contents

1 Introduction

1.1 Document Purpose

This design document serves a number of purposes:

- To define principles that will guide decisions made in design and implementation process
- To give a high-level architecture of the solution
- Provide detail the design of the solution
- State the technologies that will be used, and justify their use
- Outline development process and testing that will be undertaken

1.2 Acknowledgements

To James Thomas, for his honest feedback during the development of the UI designs, and for providing his knowledge of Jira - an existing project management system - to help refine the application.

To Brad Evans, for his assistance in guiding key decisions in the structure of the UI, and who contributed his wealth of experience in formulating website styles to help refine the colours and styles used throughout the application.

1.3 Glossary of Terms

- **The Client, The Client Application** - the HTML/JavaScript client application created specifically for the project management system
- **Project Management Server** - the hosted daemon that processes requests according to defined business logic rules and interacts with the database
- **Application Server** - a synonym of Project Management Server
- **View** - a single screen within the client application
- **Page** - a HTML web page
- **The System** - the arrangement of all components, including but not limited to the application server and the client application, that make up the Project Management System

2 User Interface Design

2.1 User Interface Design Principles

Having a set of UI design principles to guide the design of the UI results in an application which is consistent, and subsequently easy to learn to use.

- UIDP-1 - Tasks that the user does often should be as quick as possible
  - This means that a commonly used features should be quick to access and simple to use, to make sure that it isn't a burden on the user - after all, this tool is supposed to assist them in their work, not add more work.
- UIDP-2 - If the user does something wrong, give feedback to the user as soon as possible
- UIDP-3 - Don't interrupt the user unless necessary
- UIDP-4 - Ask for user's confirmation before doing something serious or irreversible
- UIDP-5 - The view content should only have the minimum content required for that view; details and advanced options should only appear when required by the user
  - This will make the view itself clear and uncluttered, which in turn will make the user interface easier to understand for new users.
- UIDP-6 - There should be platform-independent visual cues to indicate actions a user can do
  - For example, some desktop applications may change the cursor to indicate what will happen when the user clicks on the element, such as a pointing hand to indicate an action - such visual cues would not work on mobile devices that don't have cursors. Instead, the design and style of the elements of the UI should indicate functionality.

2.2 Colour Scheme

2.2.1 Primary Colours

The primary colours define the theme that runs throughout the application - for this project, a bright blue #2D59FF has been selected. Additionally, variants of this colour will used to indicate hierarchy - figure 2.2.1.1 shows the colours chosen, including an off-white #FCFCFC for the background.

2.2.2 Status Colours

These bright, bold colours will be used to visually indicate the context of information given to the user - for example, green colours are typically used to indicate that something has been successful, whereas reds are used to indicate that something has gone wrong. In addition to the text, they will convey the nature of the information at a glance. Figure 2.2.2.1 shows the colours that have been chosen.

Figure 2.2.1.1: The Application's primary colours and their hex codes

Figure 2.2.2.1: Status colours and their hex codes

2.3 Common Components

Alerts and Notifications are small boxes of content that inform the user. They will use one of the status colours as a background colour and a simple icon to indicate the nature of the information - for example, it could have a success background colour with a tick to indicate a user has been successfully created. Alerts will appear within the content of the view, whereas notifications pop up from the bottom of the screen over the content, with a slight shadow to indicate height above the view. Figure 2.3.1 shows some examples of alerts, and figure 2.3.2 includes an example notification.

Panels will be used to divide content into logical sections, and will be collapsible on smaller screens, using CSS3 media queries and JavaScript to control visibility. Figure 2.3.2 has some examples of panels - note that the headers use the primary colour.

The basic **View Structure** includes a navigation bar fixed at the top of the screen - even as the user scrolls - for easy access, a header bar to give the context of the current view to the user, followed by the main content. Figures 2.3.2 and 2.3.4 show example screens on mobile and desktop devices, respectively.

Dialogues are like small windows that appear over the view content. Their purpose is to 1) get user confirmation when doing something serious or irreversible (e.g. to confirm deleting a user) as described by UIDP-4, and 2) to move certain functionality off the view to keep it clear and uncluttered, as described in UIDP-5. Figure 2.3.3 has an example dialogue with an input form, and demonstrates how the user will be informed of validation.

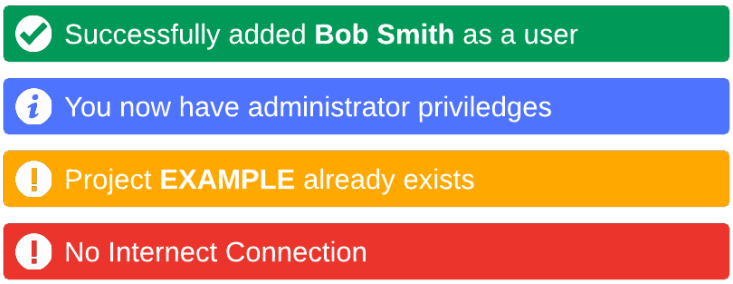

Figure 2.3.1: Some example alerts that will be used in the application

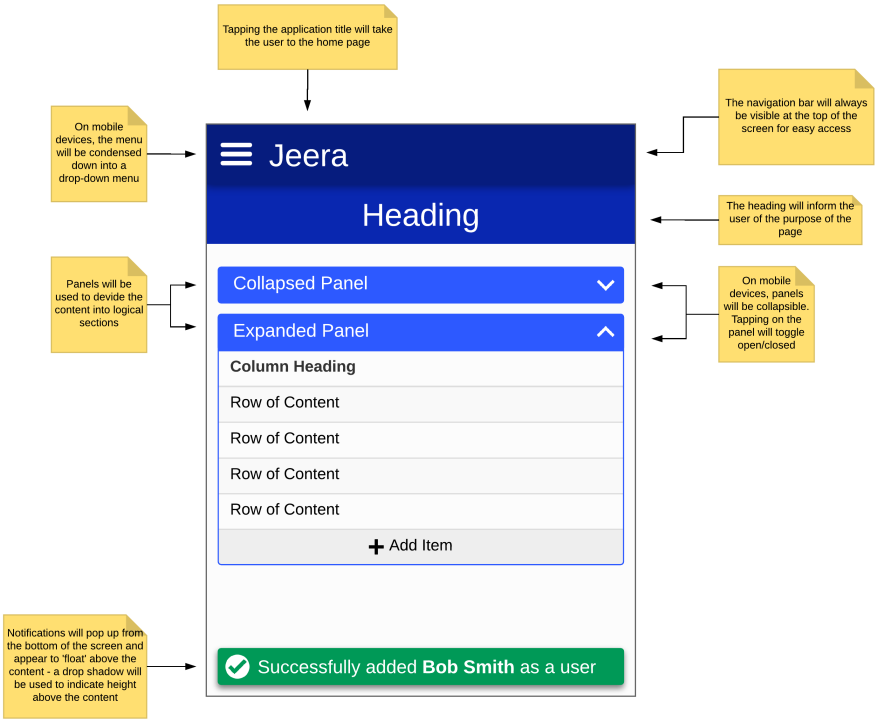

Figure 2.3.2: An example view of the application on a mobile device, with some of the common components annotated

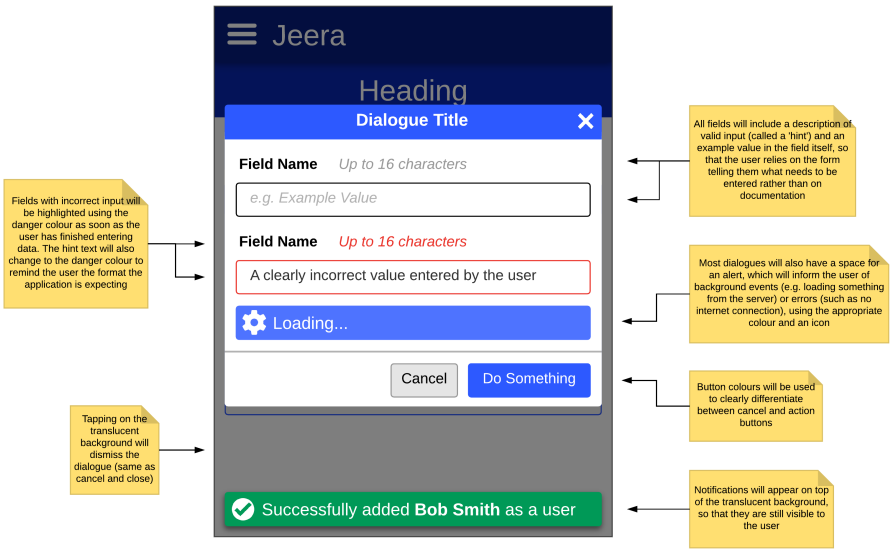

Figure 2.3.3: An example view of a dialogue with a form - including notes on validation behaviour

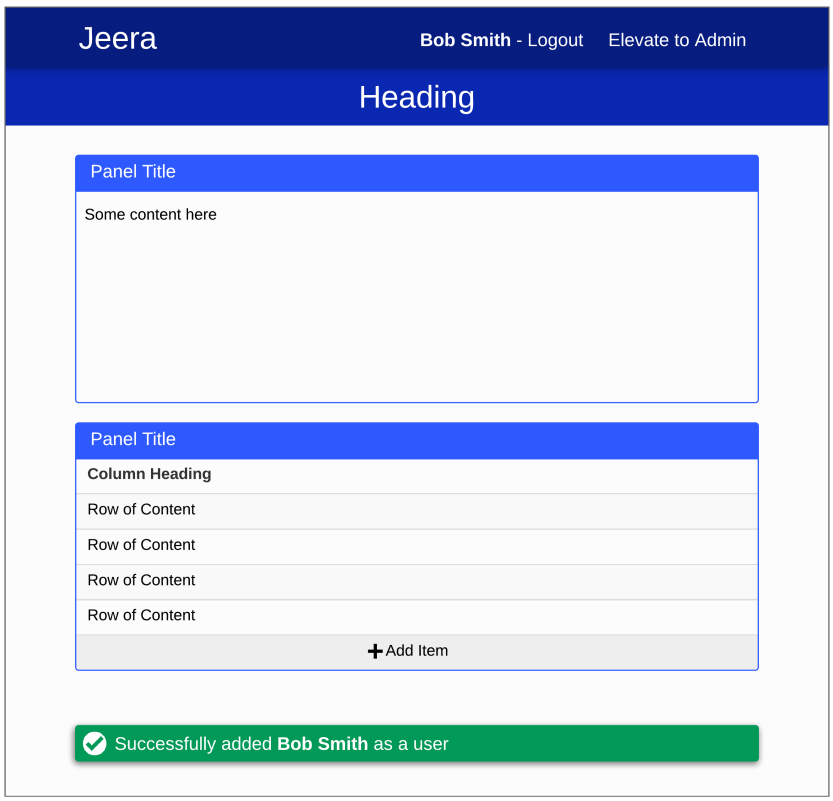

Figure 2.3.4: An example view of the application on a desktop device

2.4 View Responsibility and Navigation

This section looks at the responsibilities of each view and navigation between views. The responsibilities of a view describe what the user can do on that view - if a view has too many responsibilities, it will be overcrowded and cluttered. The navigation between views describes direct navigation between views - e.g. clicking a link or a button.

General Navigation

- All views go home if the user clicks on the application title in the navigation bar
- All views can go to the administration view from the navigation bar if the user is elevated to system administrator
- Back navigation is not described in the following description, as it is implicit. Only forward navigation described.
- Each time the user goes to the home view, back navigation is reset.

Starting the Application

When the application starts:

- If there is a password reset token in the URL, then go to the Set Password view
- If the user is logged in with valid details, the go to the Home view
- Otherwise, go to the Login view

Login View

Responsibilities

- Allows the user to log into the system

Navigation

- Forgot Password view when the user clicks 'Forgot Password?'
- Home view when the user successfully logs in

Forgot Password View

Responsibilities

- User requests a password reset (forgotten password) using this view

Navigation

- There is no navigation from this view; a link in the email they receive will go to the Set Password view

Set Password View

Responsibilities

- There is no navigation from this view; a link in the email they receive will go to the Set Password view

Navigation

- Home view after successfully setting their password

Home View

Responsibilities

- Shows all projects the user is assigned to, for quick access

Navigation

- View Project view when the user selects a project

User Details View

Responsibilities

- Allows the user to change their details (name etc.)
- Allows the user to change their password

There is no direct navigation from this page - only back or going home.

Administration View

Responsibilities

- Add a user to the system
- Remove a user from the system
- Add a project to the system

Navigation

- Manage Project view when the user selects a project

Manage Project View

Responsibilities

- Add a user to the list of project members
- Remove a member from the list of project members
- Change a member's role in the project
- Mark the project as complete

Navigation

- View Project view for the current project

View Project View

Responsibilities

- Shows all tasks in the project, broken into 4 sections:
  - Tasks assigned to the current user
  - Unassigned tasks
  - Other tasks
  - Completed tasks

Navigation

- Add Task view, if the user is allowed to add tasks to the project
- View Task view
- Manage Project view, if the user is allowed to manage the project

Add Task View

Responsibilities

- Create a new task within a project

Navigation

- View Task view if the task is created successfully
- View Project view if the user cancels creating a task

Edit Task View

Responsibilities

- Edit certain details on a task

Navigation

- View Task view

View Task View

Responsibilities

- View details of the task
- View the work log of the task
- Add a work log entry to the task, if the user is allowed to add a log entry
- Delete a work log entry of the task, if the user is allowed to delete a work log entry

Navigation

- Edit Task view, if the user is allowed to edit tasks within the project

2.5 Wireframes

The wireframes have been created based upon the view responsibilities and navigation described above. It is a mobile-first design; the appearance on desktops or laptops would be slightly different, but mobile is priority during first phase of design and development. Additional wireframes for different devices will be created after the first round of usability testing, where potential users will give feedback which could result in changes to the user interface design.

Wireframes, as the name suggests, lack style - instead, they are just the outline of the elements, perhaps with some simple styling to indicate function (for example, buttons are a light grey colour).

Each view is annotated to explain the purpose and function of certain elements. Some pages appear multiple times, but with dynamic elements such as dialogues; the most logical approach to viewing them is to open the main view (e.g. `View Task.png`) followed by additional wireframes (e.g. `View Task - Log Work dialogue.png`).

3 Solution Architecture

The architecture is an important aspect of the design, especially of a single-page application, where significant functionality must be moved from the server to the client. The architecture provides a framework that governs the structure of the application - this is especially useful for teams of developers as it results in consistency within the application, which means it that once a developer understands the architecture, they understand the purpose of different classes and why the classes were structured in a particular way, even though the code was written by another developer. This reduces the need for unnecessary documentation, which can be challenging to keep up-to-date.

The application will be built following the 3-layer architecture of an application - the user interface layer, the business logic layer, and the data access layer.

3.1 User Interface Layer

The User Interface logic will follow the MVVM architecture (Model, View, View Model), initially developed by Microsoft (2012). Essentially, the user interface is divided into stateless views, which describe what the UI looks like, and viewmodels, which hold the state of the view and contains the view's logic and behaviour. The view is 'bound' to properties that the viewmodel exposes, meaning that any changes to the viewmodel properties propagate to the view, and changes made to view causes the viewmodel properties to be updated accordingly. The viewmodel interacts with the 'model', which is a term used to represent business logic and data classes. Figure 3.1.1 is a simple rendition of the architecture.

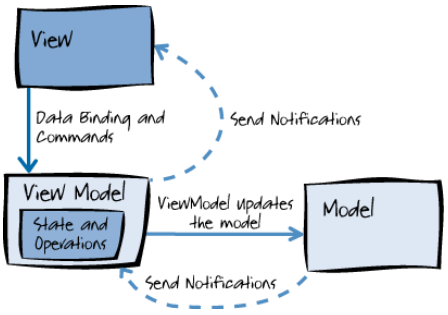

Figure 3.1.1: A high-level view of the MVVM architecture (Microsoft, 2012)

This separation of concerns - that is, separating the appearance from the state and behaviour - simplifies the logic needed to control the interface, as the viewmodel is only mutating simple properties, rather than referencing UI elements. Additionally, the inversion of component coupling, where the view knows of the viewmodel and the viewmodel has no knowledge of the view, has many advantages over alternative architectures such as MVP (Model, View, Presenter). In MVP, the view has state and the presenter is responsible for updating the view's state as changes occur, which results in the view and presenter being tightly coupled; MVVM, on the other hand, allows reuse of view behaviour between views because the viewmodel is independent of the view.

Each View and Viewmodel will be a class; most of the time, there will be one viewmodel class per view. Complex views may have sub-viewmodels for individual components with state, such as lists of items. Take the view project view as an example; a task has additional options that appear beneath the task - whether or not the details section is open is state, and should therefore be

stored in a viewmodel, so the list must be a list of some type of viewmodel. However, this raises issues when updating the list - simply replacing all items in the list would cause all state to be lost; this lead to the development of an algorithm to preserve the viewmodel's state, as described in Appendix 1.

3.1.1 Navigation

An important aspect of a single page web application is navigation - a responsibility typically handled by the browser in a 'traditional' website. The architecture of the navigation system was designed based on the requirements of a number of other MVVM applications that the developer has worked on, and thus the architecture is intended to be reusable and independent of any particular application.

The navigation system assumes that the application is formed of a number of independent views where only one view is visible at a time, similar to that of the pages of a website. Each view has a unique identifier, to simplify referencing each view.

At the core of the navigation is the router, which is responsible for executing the logic for navigating between views; it allows for logic to be executed before a view is loaded and when the view is 'done' (that is, when the view has successfully finished its purpose). In its most simple form, the navigation logic for entering or leaving a particular view will simply change the current view to a another predefined view, but it could also show dialogues or even initiate navigation to multiple different views depending on a number of conditions.

The navigation logic is not defined in the router, but in a number of classes called navigators. Each navigator defines all of the navigation logic for a particular part of the application - for example, there will be a ProjectNavigator class that defines what to do when entering 'Manage Project' and 'View Project', and what to do when leaving these views. Separating the navigation logic into individual classes prevents the situation of a single, large 'Navigator' class, which would result in unmanageable code.

When a view is navigated to, the view's viewmodel is given the unique identifier of the view and any additional data that the viewmodel needs to complete its purpose. Navigation away from the view is initiated from the viewmodel in one of two ways, either by informing the router that they are 'done', or by calling a method on a navigator. For example, when the 'Login' view has completed its purpose (when the user has successfully logged in), the 'Login' view's viewmodel will inform the router that it is done, giving the router the unique identifier of the view. The router will then lookup and execute the appropriate 'done' handler for that view, which was declared by one of the navigators when the application started, which in the case of the login view is navigate to the home view. Another example is the 'View Project' view, where tapping on a task needs to go to the 'View Task' view; the viewmodel will simply call a method on the appropriate navigator, given the task data model as an argument, and the navigator will pass both the unique identifier for the 'View Task' view and the task data model to the router, which will invoke the logic to navigate to the view.

As a result, the viewmodels can be made in such a way that they are almost completely independent of the view and the navigation logic, and are only loosely coupled to the interfaces of the navigators; this means that navigators

that implement the same methods could be substituted for each other, allowing viewmodels to be reused between multiple views. For example, in a previous application, there was a requirement to have a setup process when the user logs in for the first time, and a number of the views were functionally identical to some of the settings views. Rather than duplicating the logic, the viewmodels were reused between the setup process and the settings, which was only possible because the viewmodels were independent of the navigation logic. Although there is no such requirement in this application, the architecture was still used in order to continue refining it.

3.2 Business Logic Layer

The business logic layer will consist of the business logic classes (as the name suggests). The logic in these classes will be invoked from the viewmodels, and they will interact with the data access layer to communicate with the server and to store the data for offline use.

3.3 Data Access Layer

3.3.1 Data Models

Data Model classes will represent the data, and will also include logic for interacting with other related data models; for example, there will be a data model class to represent a Project, which will have the logic necessary for retrieving a project's tasks via appropriate API classes, which are described in the next section.

3.3.2 API Classes

API classes will provide abstraction from low-level network requests by mapping method calls onto various HTTP requests, and as a result, the class methods will closely follow the structure of the REST API exposed by the server. Additionally, their role is to abstract higher components of the application from HTTP status codes by translating the HTTP status codes into instances of various classes, collectively known as Statuses. This allows complex errors (such as a combination of the HTTP status code with an application-specific error code within the response body) to be represented by a single object, and it means that new error responses that could occur as more features are added to the server are handled gracefully by the application.

For example, suppose a particular API call may return a 'BadRequestStatus' object when a 400 status code is returned from the server. The logic further up the stack - primarily the viewmodels - would check the type of the status object, and if it is an instance of the 'BadRequestStatus' class would show the appropriate message to the user. Later in the development of the server, suppose some additional data is added to the response body to indicate why there was a bad request (400) response; in the API method, it would create the appropriate instance of a new status class - let's call it the 'InvalidValueStatus' class - to represent this data, which would extend from the 'BadRequestStatus' class. This means that any existing code higher in the application, not aware of the new 'InvalidValueStatus' class, would still function as expected, since that code is looking for an instance of the 'BadRequestStatus' class or anything that

extends from it. This use of the type system to produce a tree of status classes allows for code that is highly adaptable to change.

3.3.3 Data Model Caching

Data model caching is a design pattern created specifically for this project, and differs slightly from the typical use of the term 'caching', such as caching data to reduce the number of network requests. Essentially, different data model caches maintain references to all instances of a particular class of data model, such as Tasks or Projects. When data is retrieved from the platform, the API instances will interact with the Cache instances to either find an existing instance of a data model and update its attributes, or create a new data model and add it to the cache - data models will be deleted from cache when getting complete set of data. This simple design pattern has a number of benefits:

- API calls will be (on average) faster, because once a data model exists, only a reference needs to be passed, rather than invoking the constructor of the class.
- There will be a lower memory footprint, because fewer data models exist in memory, as they are referenced rather than repeatedly created.
- Since each entity will only be represented by one data model which gets updated by each network request, all views that have a reference to a particular data model will automatically get updated if some attributes have changed (either by the user or when retrieving data from the server), meaning nothing gets 'out of sync'.
- This technique means that not all API calls have to return all details about the data they reference; for example, project members have roles and roles have a set of permissions, but it would waste bandwidth to send the role's permissions with each project member, because there may be many members with the same role. However, the proposed caching technique means that two separate API calls (e.g. one to get the details and permissions of the roles, and another to get a project's members) are automatically merged together in the data access layer, which greatly simplifies logic in the interface and business logic layers, and reduces the network bandwidth of the requests to the server.

4 Security

Security is primarily enforced by the server, as it is an exposed endpoint, meaning anyone with an internet connection could launch an attack against the server; thus, the server needs to be protected from anything that could affect the confidentiality, integrity, and accessibility of the data stored in the system.

The server implements security mechanisms and enforces correct business logic rules, but the client is aware of these mechanisms and rules in order to change the appearance of the application, such as to remove buttons if the current user doesn't have the required permissions. However, there are some elements of security which are directly relevant to the technical design of the application, which will be briefly discussed here.

Appendix 2 includes the server API design for reference; it is discussed in further detail in the server design document. Further details on the security of the system are discussed in the design document for the server; subjects include (but not limited to) HTTPS termination and configuration of cipher suites, authentication and authorisation mechanisms, and auditing.

4.1 Authentication

Authentication is achieved using a variant of OAuth2. When the user logs in, the client will receive two unique tokens from the server - an access token and a refresh token - which are used to authenticate the client. The access token is used to authenticate most requests made to the server, whereas the refresh token is used solely for the purpose of getting a new access and refresh token pair.

Both the access token and refresh token expire after a period of time; the access token will expire after a few hours, whereas the refresh token can last days or weeks. When the access token expires, all requests that require authentication will be rejected by the server with an 'Unauthorised' status code; the client application will automatically retrieve the new tokens using the refresh token and retry the request with the new access token. If the refresh token expires, the user must log in with their username and password; this is all in accordance with SR-5.1 and SR-5.2 from the requirements.

The API requires that the access and refresh tokens are sent using the HTTP 'Authorization' header, rather than as a cookie; therefore, the tokens will be stored in local storage, since they should not be sent to the server with each request.

4.2 Authorisation

System Administrator

A system administrator is a special user who can, when elevated (see SR-5.3 in the requirements document), perform actions not possible by other users. If the user is a system administrator, an 'Elevate to Admin' option will be visible in the navigation bar, which will open the 'Admin Elevation' dialogue. After confirming their password, the client will use the refresh token to obtain a new token pair, and then request elevation from the server. The reason the client requests a new token pair is because if the access token had somehow been

obtained by a malicious actor, when the user requests elevation, the malicious actor could use the access token - which is now represents an system administrator elevated session - to make serious changes to the system, and prevent legitimate users from accessing the system. This is also the reason why the elevation only applies to the current login session, as described by SR-5.5. The application will be aware of elevation in order to allow the user to access the Administrator view.

4.3 Offline Storage

It is a long-term goal is to have offline functionality using local storage; this present potential security vulnerabilities if the device is lost or stolen, as described by Threat 5 in the requirements document.

One possible solution is that the user could set a PIN after logging in, which is used to encrypt data stored locally; the user would enter this PIN when the app is started to decrypt the data. This would minimise the business impact of losing device, but it could also have a negative performance impact, especially since the application is designed to work on mobile devices, which do not have significant computational resources available.

5 Technical Design - Class Responsibility Collaborator

This is a high-level design of the classes that will be used by the client, documented using the Class Responsibility Collaborator (CRC) technique with a slight difference, in that the responsibility of each class has been broken into what the class 'knows' (the data it holds) and what it 'does' (the behaviour and functionality of the class). If a class extends another class, it will extend that class in the code, and as a result, inherit the responsibility of the class it extends from.

The classes have been divided into related sections, such as data models and viewmodels. View classes are not included in this design, as their sole purpose is to render HTML based upon the state of their viewmodel. Additionally, navigator classes have not been included for the same reason.

5.1 Data Models

User

Knows

- User ID
- Email
- First Name
- Other Names
- Whether or not the user is a sysadmin
- When the user's sysadmin eleavation expires

Does

- Updates the user's details

Collaborators

- Response
- UsersApi

Project

Knows

- Project ID
- Project Name
- Project Icon URL
- Date of Completion
- The project's members

- The permissions the current user has in the project

Does

- Updates the project details
- Gets the project's members from the platform
- Adds a member to the project
- Gets the current user's project permissions
- Gets project tasks
- Adds a task

Collaborators

- UserManager
- MembersApi
- ProjectsApi
- TasksApi
- Response

Role

Knows

- Role ID
- Role Name
- Role's Permissions

Member

Knows

- The project
- The user
- Their role

Does

- Removes the member from the project
- Changes the member's role

Collaborators

- Project
- User

- Role
- ProjectsCache
- UsersCache
- RolesCache
- MembersApi
- Response

Task

Knows

- The task's project
- Task ID
- Summary
- Description
- Creation Date
- Target Completion Date
- The task's state (Open, In Progress, and Completed)
- When the task was completed
- Priority (1-5)
- Estimated Effort
- The assigned user ID

Does

- Updates the task details
- Gets the task's work log
- Adds a work log entry

Collaborators

- Project
- ProjectsCache
- User
- UsersCache
- TasksApi
- WorkLogApi
- Response

WorkLogEntry

Knows

- The task this work log entry belongs to
- Work log entry ID
- The user ID the log entry belongs to
- Log description
- Log effort
- Log timestamp

Does

- Deletes the work log entry

Collaborators

- Task
- User
- UsersCache
- WorkLogApi
- Response

5.2 Data Access Classes

Status Classes

There are a special set of Status classes, extending from a base 'Status' class, that will be used to represent various response statuses. If a class collaborates with the 'Status' class, it means that it will be using any of the subclasses of 'Status' - see Solution Architecture for more information on the purpose of status classes.

RestResponse

Knows

- The status code of the response
- The parsed JSON object of the response

RestClient

Knows

- The base URL for all requests
- Headers to attach to all requests

Does

- Make requests to different URLs
- Serialise/Deserialise data (just JSON, for now)

Collaborators

- RestResponse

AuthenticationApi

Knows

- The base URL for all authentication requests

Does

- Handle login requests
- Handle logout requests
- Handle token refresh requests
- Handles requests to trigger a password reset for a user (not the current user)
- Handles system administrator elevation requests

Collaborators

- RestClient
- RestResponse
- Status

Response

Knows

- The status object that represents the nature of the response
- The data, if any, returned from the server

AuthenticationClient

Knows

- OAuth tokens

Does

- Saves/loads OAuth tokens to storage
- Appends authorisation headers to all requests made through it
- Automatically refreshes tokens when the access token expires, and retries the request

Collaborators

- AuthenticationApi
- Response
- RestClient
- RestResponse
- Status

UsersCache

Knows

- A map of user IDs onto User models

Does

- Finds a User model or creates a new one

Collaborators

- User

UsersApi

Knows

- The base URL for all requests

Does

- Handles requests to create a user
- Handles requests to deactivate a user
- Handles requests to get all users
- Handles requests to get a specific user's details (names, email, etc.)
- Handles requests to get all projects a user is assigned to
- Handles requests to change user information (except sysadmin status)

Collaborators

- AuthenticationClient
- Response
- RestResponse
- Status
- User
- UsersCache

ProjectsCache

Knows

- A map of project IDs onto Project models

Does

- Finds a Project model or creates a new one

Collaborators

- Project

ProjectsApi

Does

- Handles requests to get all projects
- Handles requests to get a project's details
- Handles requests to get all members in a project
- Handles requests to create a project
- Handles requests to mark a project as complete

Collaborators

- AuthenticationClient
- ProjectsCache
- Response
- RestResponse
- Status

MembersCache

Knows

- A map of project ID and user ID pairs onto Member models

Does

- Finds a Member model or creates a new one

Collaborators

- Member

MembersApi

Does

- Handles requests to get all project members
- Handles requests to add a member
- Handles requests to update a member's role
- Handles requests to remove a member

Collaborators

- AuthenticationClient
- MembersCache
- Response
- RestResponse
- Status

TasksCache

Knows

- A map of project ID and task ID pairs onto Task models

Does

- Finds a Task model or creates a new one

Collaborators

- Task

TasksApi

Does

- Handles requests to get a project's tasks
- Handles requests to get a specific task's details
- Handles requests to create a task
- Handles requests to update a task
- Handles requests to get all work log entries for a task
- Handles requests to add a work log entry to a task
- Handles requests to delete a work log entry

Collaborators

- AuthenticationClient
- Response
- RestResponse
- Status
- TasksCache

5.3 Business Logic Classes

UserManager

Knows

- The currently logged in user

Does

- Logs in
- Logs out
- Handles elevation (requesting/dropping)
- Handles login expiry

Collaborators

- AuthenticationClient
- UsersApi
- AuthenticationApi
- NotificationQueue

UsersManager

Knows

- A cached list of users

Does

- Gets all system users
- Stores users in/loads users from local storage for offline functionality
- Creates a user
- Deletes a user

Collaborators

- UserManager
- UsersApi

ProjectsManager

Knows

- A cached list of projects

Does

- Get all projects
- Stores projects in/loads projects from local storage for offline functionality
- Get all project members
- Stores projects members in/loads projects members from local storage for offline functionality
- Loads project members
- Creates a project
- Updates a project
- Marks a project as complete

Collaborators

- ProjectsApi
- Response

5.4 Viewmodels

Viewmodel

Knows

- All 'changed' event handlers

Does

- Fires all 'changed' event handlers when the viewmodel changes

LoginViewmodel (Extends Viewmodel)

Knows

- The entered username
- The entered password
- Whether or not to remember the user
- Whether or not the input is valid
- Whether or not the client is contacting the server
- The error message

Does

- Initiates a login and informs the user of the result

Collaborators

- Response
- Status
- UserManager

5.5 Admin Viewmodels

UserViewmodel (Extends Viewmodel)

Knows

- The User data model
- Whether or not the client is contacting the server to delete the user
- The 'delete' event handler

Does

- Initiates deleting the system user
- Informs the user of the result via a notification
- Fires 'delete' event

Collaborators

- NotificationQueue
- User
- UserManager

ProjectViewmodel (Extends Viewmodel)

Knows

- The Project data model

Does

- Initiates navigation to the View Project view

Collaborators

- Project

AdminViewmodel (Extends Viewmodel)

Knows

- The list of UserViewmodels that represents all system users
- The list of ProjectViewmodels that represents all projects
- Whether or not the client is communicating with the server to get users/projects

Does

- Initiates loading users/projects from the server
- Initiates adding a user via the Add User dialogue
- Initiates adding a project via the Create Project dialogue

Collaborators

- ProjectViewmodel
- UserViewmodels

5.6 Manage Project View Viewmodels

MemberViewmodel (Extends Viewmodel)

Knows

- The member's user ID
- Member's name
- A list of all roles the member can have
- The user's role
- Whether or not the member is the current user

Does

- Removes the member from the project (after showing confirmation dialogue)
- Changes the member's role in the project

Collaborators

- Member
- NotificationQueue
- Response

ManageProjectViewmodel (Extends Viewmodel)

Knows

- The project's name
- The project's ID
- A list of the project's members
- Whether or not it is loading the project members from the server

Does

- Load the project members from the server
- Triggers the Add Member dialogue

Collaborators

- MemberViewmodel
- NotificationQueue
- Project
- Response
- Status

5.7 View Project View Viewmodels

TaskViewmodel (Extends Viewmodel)

Knows

- A task
- The task's identifier (project ID + task ID)
- The task's summary
- Target completion date
- Task state
- The assigned user, if any
- Estimated Effort
- Logged Effort

- Remaining Effort
- When the task was completed (if it has been completed)
- Whether or not the details panel is open
- Whether or not it is contacting the server

Does

- Triggers the Log Work dialogue
- Assign the task to the current user (if it hasn't already been assigned)
- Change the task's state
- Navigates to the Edit Task view
- Navigates to the View Task view
- Toggles the details panel

Collaborators

- Response
- Status
- Task
- User

ViewProjectViewmodel (Extends Viewmodel)

Knows

- The project
- The project's name
- The project's ID
- A list of tasks assigned to the current user
- A list of unassigned tasks
- A list of completed tasks
- A list of all other tasks

Does

- Navigates to the Add Task view
- Navigates to the Manage Project view

Collaborators

- PermissionsManager
- TaskViewmodel

5.8 Dialogue Viewmodels

DialogueViewmodel (Extends Viewmodel)

Knows

- The 'dismiss' event handler

Does

- Fires the 'dismiss' event handler

AddUserDialogueViewmodel (Extends DialogueViewmodel)

Knows

- New user's first name
- Whether or not the first name is valid
- New user's other names
- Whether or not the other names are valid
- Email
- Whether or not the email is valid
- Whether or not the app is contacting the server
- An error message

Does

- Initiates the process of adding a new user

Collaborators

- UsersManager
- User
- Response
- Status

AddProjectViewmodel (Extends DialogueViewmodel)

Knows

- New project's ID
- Whether or not the ID is valid
- New project's name
- Whether or not the name is valid
- New project's icon URL

- Whether or not the URL is valid
- Whether or not the app is contacting the server
- An error message

Does

- Initiates the process of creating a project

Collaborators

- ProjectsManager
- Project
- Response
- Status

AppProjectMemberDialogueViewmodel (Extends DialogueViewmodel)

Knows

- A list of users that aren't already project members
- A list of all roles on the system
- The selected user
- The selected role
- Whether or not the app is contacting the server
- An error message

Does

- Initiates the process of adding a member

Collaborators

- Project
- Response
- Status

RequestElevationDialogueViewmodel (Extends DialogueViewmodel)

Knows

- The entered password
- Whether or not the app is contacting the server

Does

- Initiates the process of requeting system administrator elevation

Collaborators

- UserManager
- Response
- Status
- An error message

6 Implementation

6.1 Technologies

6.1.1 React

React.js is a client-side JavaScript library for building the UI of HTML web applications, and it is developed by Facebook. The main reason React will be used is because React.js views are declarative, and are rendered as a function of state; this means it will work well with the MVVM design pattern, where the viewmodel has the state and views are stateless. The views are written in JSX, an extension of JavaScript which includes a mix of standard JavaScript code and HTML elements, which gets transpiled down to standard JavaScript by the React.js build tool.

React.js batches changes to the Document Object Model (DOM), which prevents what is known as 'layout thrashing', which is where a large number of changes to the DOM occur in rapid succession, and each one triggers page reflow - which is where the positions of all the elements are recalculated - resulting in a sluggish and unresponsive application. Since React.js handles batching, this doesn't have to be done in the application code.

Additionally, React.js offers highly reusable elements, meaning that common components (such as dialogues or alerts) can be written once and used many times throughout the application.

React.js has good browser support - all major browsers (Chrome, Firefox, etc.), except Internet Explorer 8 and below. This could pose compatibility issues for businesses that heavily use Internet Explorer; however, the client application is designed primarily for mobile devices, which means it will likely be running in Chrome, Firefox, or Safari.

6.1.2 ES6 and Babel

ECMA Script 6 (or ES6 for short) is the 6th official standard for JavaScript, and includes a number of improvements and additional features to the language. It will be used primarily for the addition of formal classes, lambdas, and the new style of importing modules; while these improvements to the syntax are not essential, they result in much clearer code that is easier to understand. For example, it is possible to have classes and inheritance in ES5, but it involves manipulation of prototypes, which add unnecessary clutter to the code and make the chain of inheritance harder to see - ES6 classes, on the other hand, take an approach more like Java's classes by defining class methods and properties within a block. While the internally they still use prototypes, it is much easier to understand the purpose of the class and its internal workings, and code that is easier to understand is easier to maintain.

However, most browsers currently do not fully support the ES6 standard, so Babel - a JavaScript transpiler - will transpile the ES6 code down to ES5, which is supported by virtually all browsers.

6.1.3 Async-Await

Async-await is a fairly new syntax for writing asynchronous code, with the aim of producing clearer and more understandable asynchronous code. Since all

interactions with the server are asynchronous (if they weren't, the whole application would have a very poor user experience, as each request would lock the UI thread), async-await will be used to control execution flow of asynchronous code. It was originally created for C#, but has since been adopted by other languages, including JavaScript in the ES7 standard.

The main benefit of async-await is the fact that it results in asynchronous code that resembles synchronous code. Consider a basic example of asynchronous code to retrieve a project and its tasks using the callbacks, where functions are passed to asynchronous methods to be executed when the asynchronous action is complete.

```
function getProject(projId, callback) {
  projectsApi.getProjectDetails(projId, function projCallback(err, project) {
    if (err != null) {
      callback(err);
      return;
    }

    if (project == null) {
      // Project doesn't exist
      callback(null, null);
      return;
    }

    tasksApi.getProjectTasks(projId, function tasksCallback(err, tasks) {
      if (err != null) {
        callback(err);
        return;
      }

      project.tasks = tasks;

      callback(null, project);
    });
  });
}
```

Here, there are two asynchronous calls - one to get the project details, and another to get the project's tasks. In other, more complex scenarios, there could be more asynchronous methods, each invoked in the callback of the previous. This results in long code that is significantly indented and very hard to read, especially when there are conditional calls to asynchronous methods. On the other hand, this same example code with async-await is much shorter - in fact, it is only 5 lines, as opposed to the previous example's 18 lines.

```
async function getProject(projId) {
  var project = await projectsApi.getProjectDetails(projId);

  if (project != null) {
    project.tasks = await tasksApi.getTasks(projId);
  }
```

```
  return project;
}
```

In this example, the exception handling is almost identical to synchronous code, where the exception will be thrown up the call stack until there is a try-catch statement to handle it, instead of the manual handling of errors when using callbacks. This results in less complex code which is - in this developer's experience - easier to write, understand, debug, and maintain; hence, async-await was chosen for this project.

6.1.4 SASS and SCSS

SASS (Syntactically Awesome Style Sheets) is a transpiler which converts an extended form of CSS, called SCSS, down to CSS3. It add numerous features that simplify writing stylesheets, such as inheritance, selector hierarchy, variables, and mixins (which allows CSS to be generated as a function of a few parameters).

The primary uses of this technology will be variables and selector hierarchy. Variables will be used to hold colours and sizes that affect the entire application, so that if the colours need to change, only a single variable in a file needs to be edited. The selector hierarchy will be used to prevent side-effects of using the same selectors across multiple views. On a typical website, this wouldn't be a problem, as each HTML page could have its own stylesheet with styles that only apply to that page; however, with a single-page web application, all styles are in a single stylesheet, and so there needs to be a way of making sure that styles for selectors that are used across different views only apply to the appropriate view. To do this, the root element of each view will have a unique ID, and any CSS rules that must only apply to that view will have the view's ID in the selector.

Suppose the following SCSS is used to give some custom formatting to input elements on the login view:

```
#login-view {
  .form {
    max-width: 500px;

    input {
      margin-bottom: 10px;
    }
  }
}
```

SASS will convert this to CSS that the browser will understand:

```
#login-view .form {
  max-width: 500px;
}

#login-view .form input {
  margin-bottom: 10px;
}
```

The use of SASS and SCSS reduces the repetition of the view's ID, which will make the CSS easier to understand and maintain.

6.2 Accessibility

The application is required by law to include accessibility features to assist users with disabilities, and there are a number of tools available to support accessibility on the web. For example, in HTML5, semantic elements have been added, as well as the addition of the `role` attribute to indicate the purpose of the element; the application will use these elements (such as `<nav>`) and correct use of headings to produce HTML with a meaningful structure that can easily be interpreted by screen readers. There may be a need to utilise additional JavaScript to make sure that dynamic elements - such as notifications - are correctly announced by screen readers.

Additionally, there will be text that is not visible to standard users that will assist visually impaired users, such as `alt` text on images to describe the image and hidden text next to icons to indicate the icon's purpose. Also, by building a responsive website, the application should be able to handle large fonts or the browser's zoom feature by rearranging elements to fit on screen.

6.3 Development Schedule

Phase 1 - Core functionality

In this phase, the core functionality will be implemented prior to initial user feedback. This functionality will focus on the project management aspect of the application - e.g. creating projects, adding tasks, and managing members. This will likely take 4-6 weeks for a group of 3 developers to create the core of the server and client, including tests.

Phase 2 - Usability and Accessibility Testing

This phase involves inviting potential users to try using the application, including those with various disabilities, to get feedback on usability and accessibility. The system will be hosted on a private testing system, and will have some testing data in the database. Testing is discussed in further detail in the 'Testing' section. This phase shouldn't take longer than a week.

Phase 3 - Complete Initial System

The design will be refined based upon user feedback, and then development will continue, implementing changes and adding peripheral features - such as user management. This phase will likely take 6-8 weeks, as in addition to development, it will involve writing usage and configuration documentation for the client, as well as integration and performance testing.

Phase 4 - Beta Testing

Please see *Testing*.

Main Lifecycle

Once the first stable version of the system is completed, it will be licensed to customers. The main lifecycle will include bug fixes and adding additional features based on user feedback; it will be during this phase that offline functionality will be implemented and desktop improvements will be made.

6.4 Testing

6.4.1 Functional Requirements Testing

The application's functionality will be tested use a set of automated UI testing tools to verify the application functions correctly and has met the functional requirements, defined in the requirement document. The tests will be written in Gherkin, a language used for behaviour-driven development by the Cucumber testing framework, before the application is developed. At first, all tests will fail, as the application won't exist; as development progresses, the tests will provide immediate feedback, verifying that each feature is implemented correctly and meets the original requirements.

After the main development is completed, the automated tests will act as regression tests, to make sure that new features don't break existing functionality. The fact that they are automated will mean the application can be tested quickly and frequently, unlike manual (human) testing.

6.4.2 Accessibility Testing

In addition to testing the functionality of the application, there will be tests to see how easy the application is to use with accessibility tools such as screen readers. Since the application is intended primarily for use on mobile devices, this testing must be done on a range of physical mobile devices and varying platforms. Given their market share, Android and iOS will be a priority, both on smartphones and tablets. Additionally, Windows desktop OS (7 and above) should be tested, as Windows is the most common family of operating systems in the corporate environment.

A set of common use cases, based upon the requirements, will be created and used to test the application, only using each platform's accessibility tools. Key elements of the user interface will need to be tested, especially dialogues and notifications, which appear after the main page has loaded, and thus could cause issues with screen readers. Ideally, this testing will be undertaken by potential users will real disabilities, in order to gain the most accurate results.

6.4.3 Performance Testing

Performance testing, from the client's perspective, will focus on the network bandwidth requirements, and the time it takes to process the data and present it on the screen. There are numerous profiling tools in modern browsers, which will be used to capture this information.

The network tests will involve varying the quantity of data sent over the network by creating realistic data that different teams could generate - from small to large teams - to test how well the client performs. A test plan will be drawn up that will attempt to model real-life scenarios - such as over WiFi

or 3G - in order to capture as much information as possible about the most probable scenarios. This information will be used to identify the biggest use of network bandwidth and optimise the application.

Additionally, since a large quantity of data will be stored in memory, memory profiling will be undertaken to measure how effectively the device's memory is used. This will need to be tested on a number of devices with different amount of RAM, especially low-end mobile devices.

6.4.4 Usability Testing

During phase 2 of development, potential users will be invited to test an early version of the application to gain feedback on the functionality and ease of use, which will be used to refine the application before continuing development. They may provide feedback that results in significant changes to the structure of the application, so it would be wise to start usability testing as early on in the lifecycle of the application as possible.

Appendices

Appendix 1 - MVVM List Synchronisation Algorithm

Problem Definition: The algorithm was originally created for a previous project to resolve an issue with viewmodel state in lists - namely, the elements of a list may have changed, but the ones that remain may have particular state that would be lost if the whole list was simply cleared and replaced with the new data.

Aim of Algorithm: Given a new list A and an previous list B, update B such that all elements of B are the same as A only by adding or removing elements; thus, all elements of B that have not been removed retain their state.

Preconditions: Elements of A and B must have unique and orderable identifiers, and A and B must be sorted in ascending order (although this can easily be reversed if necessary).

Initialisation: Zero-based index i set to 0

Algorithm Invariant: All elements before the i^{th} element in both lists are the same; i starts at the beginning of each list (0), so this is true at the start of the algorithm.

Algorithm:
While i is less than the length of B

1. If i is equal to the length of A, there are no more elements, so remove all elements of B from and including the i^{th} element to the end of B, and break from the loop
2. If the ID of $A[i]$ is less than the identifier of $B[i]$, then $A[i]$ is a new element - insert $A[i]$ at position i in B, increment i and continue the loop
3. If the ID of $A[i]$ is equal to the identifier of $B[i]$, then this element is the same, so just increment i and continue the loop
4. If the ID of $A[i]$ is greater than the identifier of $B[i]$, then $B[i]$ has been removed from A, so remove the i^{th} element from B, leave i unchanged, and continue the loop

While i is less than the length of A

5. Insert element $A[i]$ at position i in B, and then increment i

(Note: in steps 2-4 where it says and continue the loop, it effectively means to start the next iteration of the while loop, rather like the continue statement in many languages - this means that for each iteration of the first while loop, only one of steps 1-4 will be run)

Proof of Correctness: All steps assume that the invariant is true before the condition is evaluated.

- Step 1 - all elements removed from B; now, i is equal to the length of A and B, but i has not changed - therefore, if the loop invariant was true before, it is still true, and since i is outside the list bounds for A and B, all elements of B must be the same as A
- Steps 2 and 5 - a new element is inserted into B at position i; when i is incremented, the elements before i in both lists include the element that was added to B, and so the algorithm invariant is still true
- Step 3 - the element is still present, so when i is incremented, the elements before i in both lists now include this element, so the algorithm invariant is still true
- Step 4 - the element has been removed, but i is not incremented, so the algorithm invariant is still true

Since all steps preserve the algorithm invariant, it must be true that after the algorithm has run, that list B is equal to list A - thus, the algorithm is correct. The number of steps taken is proportional to the number of elements of A, and so is $O(n)$ in time, under the assumption that inserting and deleting elements are constant-time operations.

Appendix 2 - Server API Design

Each route will have different levels of authorisation defined as permissions. This will include a description of the class of user (e.g. Sysadmin, or a project member), or it may be a permission key (e.g. `MANAGE_MEMBERS`) which is associated with the role that the user has in that particular project. This design is discussed in further detail in the server design document.

- `/auth/auth-token`
  - `GET` - Given some means of authentication (either username/password or refresh token) via the 'Authorization' header, it will return a new authentication token pair (access/refresh token) and a unique ID for the pair
- `/auth/auth-token/{Optional Token Pair ID}`
  - `DELETE` - Invalidates the specified authentication token pair (e.g. on logout)
    - If no ID is provided, the token used to make the request will be deleted
    - Requires a valid access token when making the request

  All other requests require a valid access token to succeed
- `/users`
  - `GET` - Returns a list of all users
    - Permission: Sysadmin
  - `POST` - Adds a new user

        * Permission: Sysadmin

- `/users/{User ID}`
  - `GET` - Gets detailed info on a specific user
    * Permission: The user or a Sysadmin
  - `PUT` - Updates the user's information
    * Permission: The user or a Sysadmin
  - `DELETE` - Deactivates the user
    * Permission: Sysadmin (can't deactivate themselves)
- `/users/{User ID}/assignments`
  - `GET` - Gets a list of all projects the user is assigned to
    * Permission: The user or a Sysadmin
- `/projects`
  - `GET` - Returns a list of all projects
    * Permission: Sysadmin
  - `POST` - Creates a new project
    * Permission: Sysadmin
    * The request body must contain all of the data needed to create the project
- `/projects/{Project ID}`
  - `GET` - Returns detailed information about a project
    * Permission: Project Member or Sysadmin
  - `PUT` - Updates certain information about the project (e.g. project name)
    * Permission: `UPDATE_PROJECT_DETAILS` or Sysadmin
  - `DELETE` - Marks the project as complete (archives the project)
    * Permission: `MARK_PROJECT_COMPLETE` or Sysadmin
- `/projects/{Project ID}/members`
  - `GET` - Returns a list of all members in a project and their roles
    * Permission: Project Member or Sysadmin
  - `POST` - Adds a member to the project
    * Permission: `MANAGE_MEMBERS` or Sysadmin
    * The body must include the user ID and the role ID of the role the user has in the project
- `/projects/{Project ID}/members/{user ID}`
  - `GET` - Gets the role and permissions a user has in a project
    * Permission: `MANAGE_MEMBERS`, the user, or a Sysadmin

  - `PUT` - Updates the role a user has in a project
    - Permission: `MANAGE_MEMBERS` or Sysadmin
  - `DELETE` - Removes a member from the project
    - Permission: `MANAGE_MEMBERS` or Sysadmin
- `/projects/{Project ID}/non-members`
  - `GET` - Gets a list of all users who are not in the current project
    - Permission: `MANAGE_MEMBERS` or Sysadmin
- `/projects/{Project ID}/tasks`
  - `GET` - Returns a list of all tasks in the project
    - Permission: Project Member
    - Includes basic information such as summary and time estimate
  - `POST` - Creates a new task in the project
    - Permission: `CREATE_TASK`
    - Must include all the information necessary to create a new task
- `/projects/{Project ID}/tasks/{Task ID}`
  - `GET` - Returns detailed information about a task, such as description
    - Permission: Project Member
  - `PUT` - Updates certain attributes of the task, including state changes
    - Permissions:
    - 1) Any user with `EDIT_TASK` for editing the fields
    - 2) The assigned user with `CHANGE_ASSIGNED_TASK_STATE` to change the state (e.g. to mark as completed)
    - 3) Any user with `CHANGE_ANY_TASK_STATE` to change the state
- `/projects/{Project ID}/tasks/{Task ID}/worklog`
  - `GET` - Gets list of work log entries
    - Permission: Project Member
  - `POST` - Adds a new work log entry
    - Permission: Must be the assigned user, and have the `LOG_WORK` permission
- `/projects/{Project ID}/tasks/{Task ID}/worklog/{Work Log ID}`
  - `DELETE` - Deletes a work log entry
    - Permissions:
    - 1) The assigned user with the `DELETE_WORK_LOG` permission
    - 2) Any user with the `DELETE_ANY_WORK_LOG` permission
- `/roles`
  - `GET` - Returns the list of roles each user can have

References

Microsoft (2012) *The MVVM Pattern.* Available at: `https://msdn.microsoft.com/en-gb/library/hh848246.aspx` (Accessed 28/11/2016)

Bibliography

Ambler, S.W. (1998) *CRC Modeling: Bridging the Communication Gap Between Developers and Users.* Available at: `http://bilalbajwa.pbworks.com/w/file/fetch/53054363/crcModeling.pdf` (Accessed 19/12/2016)

Gamma, E., Heml, R., Johnson, R., and Vlissdes, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston, MA: Addison-Wesley Longman Publishing Co.

Microsoft (2015) *Asynchronous Programming with async and await (C#).* Available at: `https://msdn.microsoft.com/en-gb/library/mt674882.aspx` (Accessed: 03/12/2016)