Project Management System
Application Server Design Document

Charles Pascoe

April 6, 2017

Contents

1 Introduction

This document makes a number of references to the requirements document by referencing individual requirements - for example, SR-5.2 refers to Security Requirement 5.2. A copy of the requirements document is recommended for cross referencing.

1.1 Document Purpose

- Give a high-level view of the architecture of the solution
- Describe the security mechanisms that will protect the system from harm
- Detail the design of the solution
- State the justify technologies that will be used
- Outline testing that will be undertaken

2 Glossary Of Terms

- **Client application** - an application that interacts with the application server on behalf of the user, typically a graphical application
- **Web application client** - the HTML/JavaScript client application that will be designed and built in parallel to the application server
- **Project Management Server** - the program or process (depending on the context in which the term is used) that processes requests according to defined business logic rules and interacts with the database
- **Application server** - a synonym of Project Management Server
- **System** - the arrangement of all components, including but not limited to the application server and web application client, that make up the project management system
- **HTTP** - Hypertext Transfer Protocol
- **URL** - Universal Resource Locator

3 Recommended System Setup

Figure 3.1 shows a recommended setup of a highly scalable deployment of the project management system. This section on the recommended setup of the system will be included in the documentation given to the customer to assist their setup of the system, and it is just one of the possible configurations of the system.

3.1 Transport Layer Security Termination

Transport Layer Security (TLS) must be used to encrypt all traffic between the client and the server in order to maintain confidentiality and integrity as the data traverses the internet. In the recommended system setup, there should be a node within the DMZ, behind the first firewall, dedicated to handling TLS, which involves authentication of server via a digital certificate, key exchange, encryption, and integrity checks. The recommended TLS configuration is defined in the Security section.

Having a machine dedicated to TLS termination has a number of benefits; firstly, it frees up the application server from having to handle TLS, so that the application server can focus on authentication, authorisation, validation, and business logic. Secondly, it reduces the effort needed to change the digital certificate when it expires and maintain the list of secure key exchanges and cipher suites, as it required a single change to the TLS terminator rather than to each application server container. Thirdly, it allows for TLS session resumption, where a TLS key for a particular client is reused between multiple requests, reducing the cryptographic overhead on the client and server - which is especially important if the client is a low-powered portable device.

3.2 Load Balancer

The load balancer distributes traffic evenly between application server containers, if the customer requires more that one instance of the application server. Since multiple requests from a client would likely end up going to different application server containers, it is very important that the application server is build in a stateless manner, in order to maintain consistent behaviour between multiple requests. Instead, the database will serve as the system's state.

3.3 Application Server Package

It is required that the application server be packaged in a way that simplifies deployment in a number of different configurations; therefore, the application server will be distributed to customers as a Docker image that contains all of the dependencies for the application server to run, which will greatly simplify deployment. The Docker image will be used to create containers that execute in their own isolated environment, which can be quickly created and destroyed to meet demand. Docker was chosen over alternative containerisation solutions due to its support by many hosting solutions such as Amazon's web services; this gives the customer many hosting options as alternatives to hosting on their own hardware. Moreover, it is very difficult to break out of a container to the host machine - if an attacker can somehow remotely execute arbitrary code via

an instance of the application server, then the impact is limited to within the stripped-down container environment.

By default, the application server serves the web application, but it could also be hosted by a content distribution network (CDN), which would reduce the network load on the application server and minimise web application loading time, especially when accessing the web application from another part of the world.

3.4 Database

The database should be hosted separately from application server, outside the DMZ in a secure zone, with a firewall to control access between these zones. For large setups, the customer may require multiple database servers to meet demand - they would need to make sure that the data is synchronised to maintain data integrity. However, many third-party hosting solution include scalable relational database solutions, which is beyond the scope of this document.

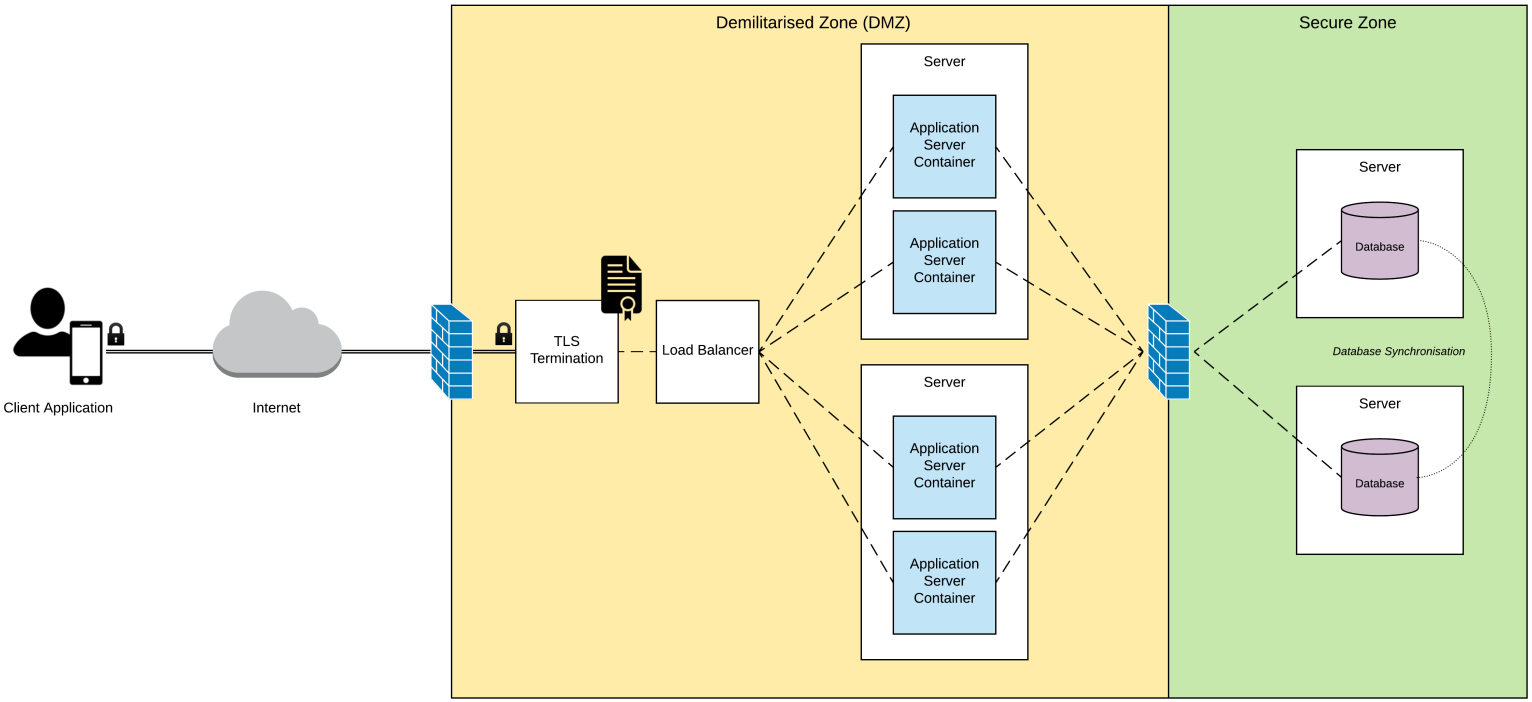

Figure 3.1: A diagram of the recommended system setup

4 Application Server Architecture

The application server is divided into 3 key sections - the interface layer, the business logic layer, and the data access layer. Data will move up through the layers when retrieved by a client application and down through the layers when changed, all controlled and regulated by the business logic layer in the middle.

4.1 Interface Layer

The interface layer contains all of the logic necessary to expose the business logic and the data in the system for a client application to use. It will be a REST (Representational State Transfer) interface, which consists of entities and actions that act upon those entities to change their state. This system will use HTTP (Hypertext Transfer Protocol) as the protocol for implementing the REST interface - the URL will indicate the entity being acted upon, and the HTTP method will indicate the action - typically `POST` to create, `GET` to retrieve, `PUT` to update, and `DELETE` to delete. REST is naturally stateless in terms of implementation, as the entities that represent the state of the system are stored in the database, thus allowing the system to scale horizontally, as described in the recommended system setup.

The interface layer of the application server will be responsible for HTTP request processing, such as parsing the URL and data sent to the server. For each entity, and for each action on that entity, there will be a method on a controller class (described below) to handle the request according to various business logic rules. The separation of HTTP routing and the business logic processing in the controller classes will simplify unit testing.

Authentication will occur in the interface layer, so that the request can be authenticated before being processed by any further logic, subsequently stopping any unauthenticated requests from reaching the controllers. There will be a controller dedicated to invoking authentication logic and deciding what to do with each request.

4.2 Business Logic Layer

Controller classes contain the business logic that is first invoked when processing a request. Each controller will handle the processing of different types of entities - for example, there will be a controller dedicated to handling all requests that act upon a task, such as getting a task's details or updating them. The controllers will be responsible for verifying that the user that initiated the request is authorised to execute the requested action on the entity, and to validate the user's input. The controllers will not interact with the database directly, but instead through data models and database interface classes.

In addition to controllers, there will be business logic classes that abstract controllers away from certain business logic rules and behaviour, so that the logic can be reused across multiple controllers. There will not be many of these classes in the first iteration of the system, but as functionality is added, more of these classes will be created to prevent duplication of logic between controllers.

4.3 Data Access Layer

The entities referred to in the interface layer will be represented in the application server as data model classes. The data models will follow the active record design pattern, where an instance of a data model class will represent a row in the data model's corresponding table, and the data model's properties will match the columns. Methods on the data model will operate on the data in the database, which results in a database interface which is simple to comprehend - for example, a work log entry data model could have a 'delete' method to remove the row from the table. Each data model will know how to retrieve data models related to it, such as an instance of a Project data model will know how to get the project's tasks and members. There will also be database interface classes, which will be used to get the 'top-level' data models such as projects and users.

Each class of data model will have a schema associated with it, which defines how the columns of the table map to the properties of the data models, and it will also include validation for each field, so that validation logic is not duplicated in the controllers. When loading data from the database, the constructor of each data model will use the schema to parse the raw data from the database, and similarly it will use the schema to build the queries used to save changes to the database. Data models will also be responsible for transforming the data into a format that the client can consume, and so data models are key for interfacing between the business logic and the database, as well as between the business logic and the REST interface.

5 Security

Security is a critical aspect of the application server, as it is accessible to anyone with access to the internet. A sound approach to security is defence in depth, where multiple security mechanisms are used to maintain confidentiality, integrity, and availability; if one mechanism fails, there is another after it. The recommended system setup covers some of these, such as the use of a demilitarised zone and TLS to protect data in transit; this section will focus mainly on the security mechanisms in the application server. The application server will be built under the assumption that the request being processed is malicious.

5.1 Clarification on the use of HTTP Status Codes for Authentication and Authorisation

The HTTP status code `401 Unauthorised` is typically used to indicate that the requested entity requires authentication, as described in section 10 of RFC-2616 (Fielding *et al.*, 1999). The status code's misleading name likely dates back to simpler web servers where authentication *was* authorisation, in that if a user could provide a correct username and password pair, the user would be allowed access to the entity. For this system, there is a need for differentiation between authentication and authorisation; therefore, `403 Forbidden` will be used when the request is correctly authenticated, but the user does not have the authorisation to access the requested entity.

5.2 Recommended TLS Configuration

This section covers the recommended configuration of TLS termination, and as such this section will be included in the system setup documentation. The predecessor to TLS, Secure Sockets Layer (SSL) version 3 was deprecated in RFC-7568, so only TLS should be used - ideally version 1.1 and above, as 1.0 is technically vulnerable to the BEAST attack when using a block cipher in cipher block chaining mode.

Server Certificate: The certificate should use either RSA (2048 bit or above) or ECDSA (any NIST approved curve), and have an expiry of no more than 1 year.

Key Exchange: Ephemeral Elliptic Curve Diffie-Hellman (ECDHE), using any of the standard NIST elliptic curves. As a fallback for TLS implementations that do not support elliptic curves is Ephemeral Diffie-Hellman (DHE) with at least a 4096 bit prime, which should changed at least once a year to mitigate the Logjam attack.

Cipher: Ideally AES Galois Counter Mode (GCM) using either a 256 or 128 bit key, with AES Cipher Block Chaining (CBC) using either a 256 or 128 bit key as a fallback for TLS implementations that do not support GCM.

Message Authentication Code (MAC): Either SHA-512 or SHA-256 for CBC mode, as GCM verifies its own integrity.

5.3 Authentication

Authentication is the first line defence against malicious activity, as it will thwart attacks made by malicious actors without valid authentication credentials. This section describes the various processes and mechanisms that will protect the system from unauthenticated attackers.

5.3.1 New Users

The new user process can only be started by system administrators. It uses out-of-band authentication for authenticating the new user - in this case, via email - and all client-to-server communication will be encrypted.

1. System administrator user enters the details for the new user into a client application
2. Client application sends new user details to the application server
3. Application Server generates a random password reset token and hashes it
4. Application Server saves the new user details and the password reset token hash to the database
5. Application Server sends an email to the new user with a link, which has the password reset token encoded in the URL
6. When the new user opens the link in the email, they will be taken to a view within the web application where they can set their new password
7. Web application sends the password and password reset token (parsed from the URL) to the application server
8. Application Server loads the user's details from the database, and hashes the given password reset token to compare it with the stored password reset hash - if the hashes do not match, reject the request and stop
9. Application Server computes the hash of the given password using a slow hashing algorithm (described in 'Logging In')
10. Application Server clears the password reset token hash from the database to prevent it being used again, and saves the password hash to the database

In step 1, the system administrator can specify whether or not the new user will also be a system administrator, meaning there can be more than 1 system administrator.

First User

The first user of the system will be created in a slightly different manner. The details for the first user (i.e. their name and email) will be stored in the application server's configuration file; when the application server is started, if there are no users in the database, it will create the first user as a system administrator, effectively starting from step 2 of the New User process. Since

the configuration is controlled by the system administrator, the first user is authenticated by the fact that the user has direct access to the application server's configuration - therefore, the server that is hosting the application server must be configured correctly to prevent unauthorised access to or modification of the application server's configuration. The first user will always be a system administrator, so that they can then add other users via a client application.

5.3.2 Authenticating Requests

Requests to the application server's API will be authenticated using a variant of OAuth2. When the user logs in, the client application will receive two unique tokens from the server - an access token and a refresh token, together referred to as an authentication token pair - which are used to authenticate the client application, and are unique to that login session (as required by SR-5.1). 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. The tokens will both be 32 bytes (256 bits) long, and generated using a cryptographic random number generator, using system sources of entropy as the seed. This will result in high-entropy tokens which are difficult to predict.

Both the access token and refresh token expire after a set period of time, as required by SR-5.2; for this system, the access token will expire after 1 hour, whereas the refresh token will either expire after 6 hours by default, or after 30 days if the client application includes a flag in the query string of the URL (see API design for details). When the access token expires, all requests that require authentication will be rejected by the server with an `401 Unauthorised` status code; the client application will need to retrieve the new tokens using the refresh token and retry the request with the new access token. If the refresh token expires, the server will return `401 Unauthorised` when attempting to get a new token pair, and the user must then log in with their username and password.

Due to SR-4.3, a hash of each token will be stored in the database instead of the cleartext tokens, so that if any part of the system somehow leaks the stored authentication information, it cannot be used to legitimately authenticate and interact with the system - unless the attacker has some means of efficiently reversing cryptographic hashes of high-entropy data, which is highly improbable. SHA-256 will be used to hash the tokens, due to its speed and the fact that it has had significant cryptographic analysis with no known significant weaknesses. Its speed is important, as it will be run on every authenticated request except password login.

5.3.3 Logging In

When the user attempts to log in, their email will be used to lookup the user's details, and the password will be verified against the stored password hash using a slow hashing algorithm, as required by SR-4.4. Argon2 is the chosen slow hashing algorithm, as it is the winner of the recent Password Hashing Competition (2015). It has a number of parameters that control cost of computing a hash in terms of CPU cycles, memory, and parallelisation, all with the aim of rapidly exhausting an attacker's computational resources when attempting to break a password hash, in the unlikely event an attacker obtains the list of

password hashes. All passwords will be hashed with a 16 byte cryptographically generated random salt to mitigate rainbow and dictionary attacks.

5.3.4 System Administration Elevation

Although a user may be marked as a system administrator, they will not be permitted to make privileged requests to the server until the client application's login session has been temporarily elevated, as required by SR-5.3, SR-5.4, and SR-5.5. In the database, all authentication token pairs will have a timestamp field associated with them, indicating when the system administration elevation expires for that session. When the system administration elevation expires, it will not expire the token pair, but rather deny any administrative actions requested using the access token. Refreshing a token pair will not clear or reset the expiry timestamp.

To gain system administrator elevation, the client application must prompt the user for their password, which gets sent to the server when requesting elevation, in order to prove that the user really is the currently logged in user. The request must be accompanied by a valid and in-date access token. If the access token is valid and the password is correct, then the system administration elevation expiry for that token will be set to 15 minutes.

It is recommended that client applications should refresh the tokens prior to requesting elevation; this is because in the event a malicious actor had somehow obtained the access token for the client, it would be invalidated before elevation, preventing the malicious actor from making authenticated privileged requests to the application server.

5.3.5 Password Resets

It is a certainty that at some point, a user will forget their password, and as a result, they will be unable to authenticate with the application server. Therefore, the application server will have a password reset process to allow users to prove their identity using their email as an out-of-band authentication mechanism.

The process is essentially the same as the new user process with the exception of steps 1 and 2 - instead, the user will open the 'Forgot Password' view within the client application and enter their email, which is sent to application server, and then the process carries on from step 3. The application server will provide a CAPTCHA test - used to prove that the request was initiated by a human user - to the client application, in order to prevent automated bots from using the application server to spam the users with password reset emails.

5.4 Authorisation

Authorisation is the second line of defence against malicious activity. After authentication, it is necessary to verify the client is permitted to do the requested action - such as getting an entity or changing it. The application server will defined a set of permissions, which will be mirrored in the database in order to allow them to be linked to project member roles. There will be a dedicated authorisor class, whose sole purpose is to check that a given user is permitted to do the requested action.

5.4.1 Authorising System Administrator Requests

The user must be a system administrator (associated with their account data) and the access token used for the request must be elevated and in-date. Some permissions which are specific to projects - such as the ability to manage a project's members - can be executed by a system administrator, even if they are not a member of the project or their role does not permit them; in effect, system administrators have the highest authority over the system.

5.4.2 Authorising Project-Specific Requests

Certain requests that affect a particular project require that the user is a member of the project, and their role within the project has the required permission - the API design lists the required permissions for each action on each entity. The separation of roles and permissions allows new roles to be added easily, and the permissions of existing roles to be changed. Although such functionality has not been included in this iteration of the design, it would be relatively simple to incorporate into the existing data structures and authorisation mechanisms.

5.5 Validation

Validation is the third line of defence against malicious activity. After authorisation, all inputs need to be checked to verify that they are acceptable and will not result in undesired behaviour of the system. The controllers will be responsible for invoking data validation and selecting the appropriate response code and message. Data model schemas will define the validation rules, so that it is not duplicated across controllers.

5.6 Accountability

SR-5.8 requires that the application server will produce sufficient information on the activity of the users, so that in the event a legitimate user deliberately causes harm to the system - such as deleting a project or removing users unnecessarily - then the logs will allow that user to be held accountable to their actions.

5.7 Responding to Potentially Malicious Behaviour

The application server will include other mechanisms which will try to protect the system from potentially malicious behaviour, such as timing requests and repeated unauthenticated requests.

5.7.1 Timing of Responses

One possible means of gleaning information from the system is by timing responses, particularly error responses, which would reveal information on how the request had been processed. Therefore, in accordance with SR-3.1, the responses to bad requests or requests that result in an internal server error should always take a consistent amount of time to respond, under the assumption that the request was malicious. Successful requests (`2XX` response codes), or `Bad Request` status codes which are expected through normal use of the system

(such as `409 Conflict` when adding a duplicate project ID), should have no delay, to prevent any negative impact on user experience for legitimate users.

This will be achieved by marking each request with a timestamp; in the event of an unexpected bad request or internal server error, the server will wait long enough to make the response appear to take some constant amount of time. The constant response time for the system will be 1 second, because it longer than all requests (based upon response timings of similar systems), and short enough that it wouldn't significantly affect the user's experience in the rare event that a legitimate request results in an unexpected bad request or server error status code.

For example, suppose a request takes 100ms to complete if some system flag was set to `true`, and 250ms to complete if the said flag was set to `false`. The constant response time logic would wait 900ms and 750ms respectively before responding, resulting in approximately 1000ms for both responses, meaning a malicious actor would be unable to determine the internal state or configuration of the system.

5.7.2 Repeated Unauthenticated Requests

Some entities in the API do not require authentication using an access token, such as when getting authentication tokens using the user's email and password. In accordance with SR-2.3, the application server will block repeated login requests, because it could be used to crack a user's password, and it is the most computationally expensive request due to the use of a slow hashing algorithm.

The application server will have a mechanism that will use an IP-based strategy to block repeated login requests. It consists of an table of public IP addresses (which will initially be in memory, before being moved to the database) along with number of failed login attempts originating from that address; when the number of failed attempts exceeds a set threshold, all further login requests are immediately blocked, but will continue to increment the attempt counter for that IP address. After a set period of time, known as the cooldown interval, all attempt counters will be decremented by 1. Once an attempt counter reaches 0, the entry will be removed from the table.

For this system, the cooldown period will be 1 minute, and the threshold for failed attempts will be 100, which far exceeds the behaviour of a legitimate user who simply forgot their password, and thus can be assumed to be malicious behaviour. This allows 100 failed attempts within the first minute of an attack, reduced to 1 a minute thereafter, which would thwart any attack against the application server via password login.

Drawbacks

However, this strategy has some drawbacks. For example, an attacker with access to a large IP pool could mount a large attack - one possible way is to use the plethora of VPS providers to create and destroy many virtual machines, each one with a new IP address taken from the provider's address range. Alternative mechanisms would need to be employed to prevent this form of attack, but due to the effort required to execute this attack, the IP-based strategy is considered acceptable for use within the application server.

Additionally, an attacker could use this mechanism for a form of Denial of Service (DoS) attack; if there are a number of users using a single public IP address via Network Address Translation (such as a large organisation), a malicious actor within that network could deliberately make many failed attempts to the server so that it would prevent other users from logging in. This is why this mechanism will only apply to password login, not token refresh or other authenticated requests, as it minimises impact on legitimate users in this scenario. Also, since the users of the system will likely be highly distributed and connecting from different networks (as described in the requirements document), it is unlikely that many users will access the system from a single public IP address, so this scenario is less of an issue.

6 Technical Design

6.1 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.

- `/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
    - `Basic` authentication (username and password, separated by a colon, encoded using Base 64) for password-based authentication
    - `Bearer` authentication for refresh token
- `/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
- `/auth/reset-password`
  - `GET` - Initiates a new password reset, returns details to render a CAPTCHA
  - `POST` - Initiates the password reset process
    - Data must include the CAPTCHA result and the user's email
- `/auth/set-password`
  - `POST` - Sets a password, either for a new user or from a password reset
    - Requires a valid password reset token and an acceptable password

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

6.2 Database Design

The database is based upon a design created for a similar project management system, but it did not have the same type of user-first requirements analysis, which has led to a number of changes - figure 6.2.1 shows a revised entity relationship diagram with the changes. The full database schema is defined in the `database/create-database.sql` file, but the summary of changes are as follows:

- `user.email` can now be `null`, which indicates that the account has been deactivated, and so `user.active` has been removed
- `user.sysadmin` field to user table to indicate that the user is a System Administrator - defaults to `false`
- The type of `project.project_id` has been changed from `INT(11)` to `VARCHAR(16)`, so tasks can be referenced as `EXAMPLE-1`
- `project.icon_url` field was added to allow each project to have a unique icon
- `project.active` has been removed and replaced by `project.completed`, which is a timestamp used to show when the project was completed (`null` indicates the project is still active)
- The permission columns have been removed from the `role` table, and replaced with the `permission` table, which is linked to the `role` table via the `role_permission` junction table. This offers a much more flexible solution to controlling the permissions each role has.
- `task.state` has been added, which is an enum of `OPEN`, `IN_PROGRESS`, and `COMPLETED` - defaults to `OPEN`.
- `task.assigned_user_id`, which replaces the `task_assignment` table as only one user can now be assigned to a task - this simplification of the system was the result of analysis of similar project management systems, such as Atlassian's Jira.

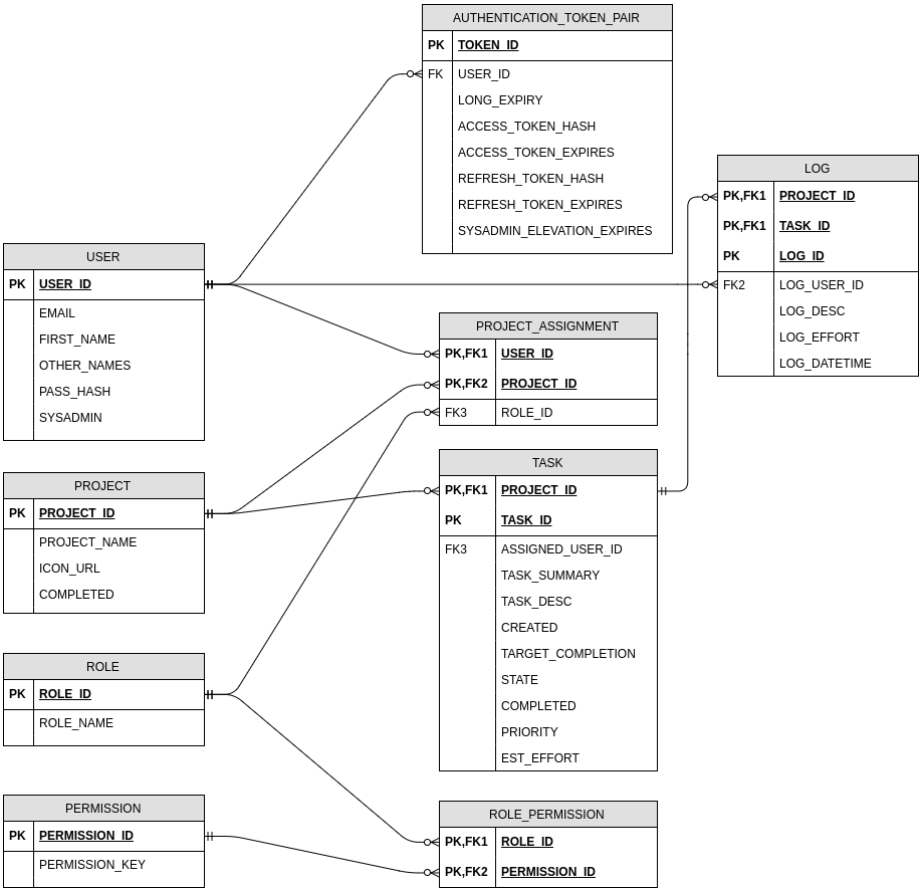

Figure 6.2.1: The latest iteration of the Entity Relationship Diagram of the database

6.3 Class Responsibility Collaborator Design

This is a high-level design of the classes that will be used by the application server, 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 controllers.

6.3.1 Data Models

Data models will be implemented in an Active Record style. Model will be the base class, and will save changes made to the entity (the entity being an instance of a class that extends Model, e.g. an instance of a the User class).

Schema

- Knows
  - The properties of an entity and the columns that they map to
- Does
  - Map properties onto columns

Model

- Knows
  - The schema for this entity
- Does
  - Saves entity changes to the database
  - Deletes this entity from the database
- Collaborators
  - Database
  - Schema

AuthenticationTokenPair (Extends Model)

- Knows
  - The access token hash and its expiry
  - The refresh token hash and its expiry
  - When this session's system administrator priviledges expire
  - Formats the data to be sent to the client
- Collaborators
  - Database
  - Schema

User (Extends Model)

- Knows
  - Email
  - Names
  - Password Hash
  - User's authentication tokens
  - Whether or not the user is a System Administrator
  - The projects the user is assigned to
- Does
  - Saves an authentication token to the database
  - Deactivate the user
  - Formats the data to be sent to the client
- Collaborators
  - AuthenticationToken
  - Database
  - Schema

Role (Extends Model)

- Knows
  - Role Name
  - Permissions (application-defined string constants) that this role has
- Does
  - Formats the data to be sent to the client
- Collaborators
  - Database
  - Schema

ProjectAssignment (Extends Model)

- Knows
  - The user
  - The project
  - The role the user has in that project
- Does
  - Formats the data to be sent to the client
- Collaborators

  - User
  - Project
  - Role
  - Database
  - Schema

Project (Extends Model)

- Knows
  - Project ID
  - Project Name
  - Project Icon URL
- Does
  - Adds a task to the project
  - Loads project tasks from the database
  - Add a user as a project member
  - Loads assigned project members from the database
  - Removes a project member
  - Formats the data to be sent to the client
- Collaborators
  - Database
  - Schema
  - Task
  - Assignment

Task (Extends Model)

- Knows
  - The project this task belongs to
  - Task ID
  - Summary
  - Description
  - Estimated Effort
  - Created Timestamp
  - Target Completion Date
  - State
  - Completed Timestamp
  - The assigned user
- Does

  - Gets the work log associated with the task
  - Formats the data to be sent to the client
- Collaborators
  - Database
  - Schema
  - Project
  - WorkLogEntry

WorkLogEntry (Extend Model)

- Knows
  - The task this work log belongs to
  - Log ID
  - The user that created the log
  - Log description
  - Log effort
  - Log timestamp
- Does
  - Formats the data to be sent to the client
- Collaborators
  - Database
  - Schema
  - Task
  - User

6.3.2 Database Classes

Transaction

- Knows
  - Transaction ID
- Does
  - Queries the database
  - Commits/rolls back the changes made to the database

Database

- Knows
  - The next transaction ID
- Does
  - Queries the database
  - Initiates a transaction
- Collaborators
  - Transaction

Users

- Does
  - Gets all users from the database
  - Gets a user by email or ID from database
  - Adds a user to the database
- Collaborators
  - Database
  - User

AuthenticationTokens

- Does
  - Adds an authentication token pair to the database
- Collaborators
  - Database
  - AuthenticationTokenPair

Projects

- Does
  - Gets all projects in the database
  - Adds a projects to the database
- Collaborators
  - Database
  - Project

Roles

- Does
  - Gets a role by ID
  - Gets all roles
- Collaborators
  - Database
  - Role

Security Classes

PasswordHasher

- Knows
  - The current parameters to use when hashing new passwords
- Does
  - Hashes passwords
  - Verifies passwords and password hashes
  - Updates password hash if it uses old parameters

Authenticator

- Does
  - Authenticates user login requests
  - Authenticates refresh requests
  - Generates authentication token pairs
  - Elevates a system administrator's login session
  - Generates password reset tokens
  - Hashes new passwords
- Collaborators
  - Users
  - User
  - PasswordHasher

Authorisor

- Does
  - Verifies that a user is correctly elevated for a system administrator action
  - Verifies that a user has the correct permission for a given project
- Collaborators

  - User
  - ProjectAssignment
  - Role

6.3.3 Controller Classes

AuthenticationController

- Does
  - Handles requests to create a new authentication token pair
  - Handles requests to delete an authentication token pair
  - Handles requests to elevate a login session
  - Handles requests to drop system administrator elevation
  - Handles requests to change the user's password
- Collaborators
  - User
  - AuthenticationTokenPair
  - Authenticator
  - EmailDistributor
  - EmailTemplator

UsersController

- Does
  - Handles requests to add a user
  - Handles requests to get all users
  - Handles requests to get a specific user's details
  - Handles requests to get a user's assignments
  - Handles requests to edit a user's details
  - Handles requests to delete a user
- Collaborators
  - Authorisor
  - Users
  - User
  - Authenticator
  - ProjectAssignment
  - EmailDistributor
  - EmailTemplator

ProjectsController

- Does
  - Handles requests to get all projects
  - Handles requests to create a new project
  - Handles requests to edit a project's details
- Collaborators
  - Authorisor

MembersController

- Does
  - Handles requests to get all project members
  - Handles requests to get project non-members
  - Handles requests to add a user to a project
  - Handles requests to update a member's role in a project
  - Handles requests to remove a member from a project
- Collaborators
  - Projects
  - Project
  - User
  - Roles
  - Role
  - Authorisor

TasksController

- Does
  - Handles requests to get all tasks within a project
  - Handles requests to create a new task in a project
  - Handles requests to get a task's details (description etc.)
  - Handles requests to edit a task
- Collaborators
  - Project
  - Task
  - Authorisor

WorkLogController

- Does
  - Handles requests to create a new log entry for a task
  - Handles requests to get all work log entries for a task
  - Handles deleting log entries
- Collaborators
  - Authorisor
  - User
  - ProjectAssignment
  - Task
  - WorkLogEntry

RolesController

- Does
  - Handles requests to get all roles
- Collaborators
  - Roles
  - Role

6.3.4 Email Classes

EmailDistributor

- Knows
  - Email distribution server configuration
- Does
  - Sends a HTML email to a given email address

EmailTemplator

- Does
  - Generates the HTML for a password reset email, with the given user's name and password reset token
  - Generates the HTML for a welcome email, with the given user's name and password reset token

7 Implementation

7.1 Technologies

7.1.1 Node.js

Node.js is used to build JavaScript HTTP servers; since this is the same language as the web application, it allows developers to work on both the web application and the application server, which will make the whole system easier to maintain in the long term, as it reduces the number of languages developers needs to learn. Node.js has a large choice of open-source modules, which can also speed up development.

A Node.js server is, by default, single-threaded, which avoids a number of multi-threading issues such as deadlock or concurrent state mutation. Instead, a server is scaled by creating more independent processes, rather than more threads - this has a slight memory overhead, but it is minimal on Linux-based systems and is outweighed by the benefits of a single-threaded architecture. For this system, there will one process per Docker container, and to scale the system, there will be many independent Docker containers, which can be orchestrated by a number of systems such as Google's Kubernetes to dynamically meet demand.

7.1.2 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 database are asynchronous (if they were not, then the server would not be able to handle as many requests, as each database query would lock the server's 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 callbacks, where functions are passed to asynchronous methods to be executed when the asynchronous action is complete.

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

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

    project.getTasks(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 data model, 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 getProjectWithTasks(projId) {
  var project = await projectsDbInterface.getProject(projId);

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

  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.

7.1.3 MySQL

The system is highly dependent on relational data - projects have tasks, roles have permissions, and so on. MySQL will be used as the database, as SQL is a well-known query language and there are many Node.js modules to simplify interacting with the database. All queries with the database will use placeholders which get replaced by the escaped values, in order to prevent SQL injection, as required by SR-4.1. Most of these queries will be prepared statements, whereas a small number of them will be dynamically built by data models; while this should not be vulnerable to SQL injection, as all of the values will still be properly escaped, it may allow certain attributes to be changes when they should not be changed - hence, the controllers will need to be thoroughly tested to prevent this. The database user and password will be set in the application server configuration; it is recommended that this account should have limited

access to the system, as required by SR-4.2 - it will only require adding, editing, and removing of rows within the project management system database, not removing tables or the database itself.

8 Testing

8.1 Unit Testing

The system will be unit tested using a JavaScript unit testing library called Tape. Each class is considered a unit, and each method will have a number of independent tests run against them to check that they function as expected and are fit for their purpose. Dependency injection will be used to substitute real dependencies with testing dependencies, to isolate each unit under test and to replicate certain scenarios.

Key Areas To Focus On

- Database interface code
- Security-critical code
- API interface code, e.g. serialising the data to be sent to the client application

8.2 Performance Testing

A testing system will be set up on reference hardware and configuration, which will be used to stress test the application server in order to find out how many requests the system is capable of handling, using Apache's HTTP server benchmarking tool. In addition to measuring the number of requests per second, the CPU and Memory usage will be captured to predict the maximum number of Docker containers per server. The process will be repeated with a number of different configurations to measure performance, to see how well the system scales.

Key Areas To Focus On

- Login is the most computationally expensive, and also it would be a good test of the mechanism that blocks repeated failed logins
- Requesting large lists, such as getting all tasks of a project with a large number of tasks (e.g. 5000)
- Writing to the database, such as editing/adding a task

8.3 Penetration Testing

Penetration testing aims to find flaws in the security mechanisms built into the application server. The testing will be undertaken by an external team with experience in this field, given only the API design (which is effectively what any other malicious actor would have). There will be a mix of unauthenticated and authenticated testing - for the latter, they will be given legitimate but unprivileged login credentials, to see if they can break the authorisation mechanisms, as well as testing how well the system logs information for accountability.

Key Areas To Focus On

- Login
- System Administrator Elevation
- Attempting requests that require privileges that the user does not have

References

Password Hashing Competition (2015) Available at: `https://password-hashing.net/` (Accessed 15/12/2016)

Fielding, R., UC Irvine, Gettys, J., W3C, Mogul, J., Compaq, Frystyk, H., MIT, Masinter, L., Xerox, Leach, P., Microsoft, and Berners-Lee, T. (1999) *Hypertext Transfer Protocol – HTTP/1.1.* RFC-2616. Available at: `https://www.w3.org/Protocols/rfc2616/rfc2616.html` (Accessed: 17/12/2016)