SOFTWARE REQUIREMENTS SPECIFICATION

for

TIME TRACKING AND MANAGEMENT SYSTEM

Version 1.0

Prepared by: Jean Jacques Avril
Master Student, Computer Science
Technische Hochschule Rosenheim

Module: Concepts of Programming Languages

November 10, 2024

Contents

1 Introduction

1.1 Purpose

The purpose of this Software Requirements Specification (SRS) document is to provide a detailed overview of the requirements for the Time Tracking and Management System. This document outlines the system's functionality, design goals, and intended use to ensure a comprehensive understanding of its scope and potential. The system aims to facilitate accurate tracking and calculation of work hours while maintaining flexibility for future expansions. The backend will demonstrate the benefits of a functional programming approach, supporting maintainability and extensibility.

1.2 Intended Audience and Reading Suggestions

This document is intended for various stakeholders who have an interest in the development and use of the system:

- **Project Stakeholders**: For a thorough understanding of the system's features and constraints.
- **Developers**: To understand the functional and non-functional requirements, allowing for seamless implementation.
- **Quality Assurance Teams**: To ensure that testing aligns with the specified requirements.
- **Future Contributors**: To grasp the system's architecture and contribute to its expansion effectively.

Readers are advised to first review the functional requirements section to understand the core system functionality and then move to the non-functional requirements for insights into performance and security aspects. Those involved in architecture and backend development should focus on the system architecture and technology sections to appreciate how the functional programming style enhances the backend.

1.3 Project Scope

The Time Tracking and Management System is designed to provide an efficient, user-friendly platform for tracking work hours, recording breaks, and generating reports. This software was conceptualized due to a personal interest in self-hosting and experimenting with homelab setups. Existing open-source solutions did not meet the specific needs for flexibility, modularity, and architectural robustness.

The initial scope of the project includes core functionalities such as user management, time tracking, project assignments, and report generation. However, the software's architecture is designed to support future modular extensions. Potential future modules could include features like work desk planning, allowing users to plan and allocate their work tasks in advance.

The backend of this system is built with a focus on functional programming principles, enabling cleaner, more predictable code and easier scalability. This approach also ensures that additional functionalities can be seamlessly integrated as the software evolves.

The combination of a robust, extendable architecture and a functional programming backend makes this system a unique and effective solution for personal and professional time management needs.

2 Requirements

2.1 Functional Requirements

FA-1: User Registration

- **Description**: The user must be able to register in the system.
- **Details**:
  - Input of username and password.
  - Check for existing username.
  - Return a confirmation message upon successful registration.
- **Validations**: Unique username, password must meet security requirements.

FA-2: User Login

- **Description**: The user can log into the system.
- **Details**:
  - Input of username and password.
  - Return an authentication token upon successful verification.
- **Validations**: Correct input of username and password.

FA-3: Password Recovery

- **Description**: The user can reset their password.
- **Details**:
  - Request a recovery link via email.
  - Set a new password.
- **Validations**: Valid email, new password must meet security requirements.

FA-4: User Profile Management

- **Description**: The user can manage their profile data.
- **Details**: Display and edit profile information.
- **Validations**: Unique email when modified.

FA-5: Automatic Time Tracking

- **Description**: Time tracking with start/stop buttons.
- **Details**: Store start and end times for calculating work duration.
- **Validations**: Cannot start tracking if already active.

FA-6: Manual Time Tracking

- **Description**: Manual entry of work times.
- **Details**: Input start and end times, including optional breaks.

FA-7: Break Times

- **Description**: Add break times during a work period.
- **Details**: Breaks are subtracted from total work time.
- **Validations**: Breaks must be within the start and end times.

FA-8: Project Assignment

- **Description**: Assign recorded times to projects.
- **Details**: Projects can be created and managed in the system.
- **Validations**: Valid project required for time entry.

FA-9: Task Description

- **Description**: Add task descriptions to time entries.
- **Details**: Input descriptions or select from templates.

FA-10: Daily and Weekly Reports

- **Description**: Generate reports of work times.
- **Details**: Includes total work time, break time, and overtime.
- **Validations**: Reports can only be generated for registered users.

FA-11: Export Function

- **Description**: Export reports as PDF or CSV.
- **Details**: Select projects and date ranges for export.

FA-12: Reminder Function

- **Description**: The system reminds the user to start or stop time tracking.
- **Details**: Configurable notifications in the user profile.

2.2 Non-functional Requirements

NFA-1: User Data Security

- **Description**: User data must be securely stored and transmitted.
- **Details**: Passwords stored as hashes, TLS for data transmission.

NFA-2: Performance

- **Description**: Response time should be under 200 ms.

NFA-3: Scalability

- **Description**: Support a large number of concurrent users.

NFA-4: Usability

- **Description**: The interface should be intuitive and responsive.

NFA-5: Data Integrity and Security

- **Description**: Ensure data integrity and restricted access.
- **Details**: Authentication and data encryption required.

NFA-6: Performance Optimization

- **Description**: Optimization through caching techniques.

3 Technical Specification

3.1 Overall Architecture

The system is designed as a full-stack web application, comprising the following main components:

- **Frontend**: Implemented using React, connecting to the backend via WebSockets or HTTP (gRPC/REST).
- **Backend**: Two separate implementations using Go and Dart, managing requests, business logic, and database interactions.
- **Databases**: PostgreSQL for structured data persistence and Redis for caching and real-time data management.
- **Communication Protocols**: WebSockets for real-time updates and gRPC for efficient communication, with REST as a fallback.

3.2 Technology Stack

3.2.1 Frontend

- **React**: Used for building an interactive and responsive user interface.
- **WebSockets**: Integrated for real-time updates from the server.

3.2.2 Backend

- **Go**:
  - **ORM**: GORM for database interactions with PostgreSQL.
  - **Redis Integration**: Using `go-redis` for caching and real-time operations.
  - **WebSockets**: Implemented using the `gorilla/websocket` package.
  - **gRPC/REST**: Using gRPC for communication, with `net/http` and `gorilla/mux` as a REST fallback.
- **Dart**:
  - **ORM**: Using `drift` or `aqueduct` for PostgreSQL integration.
  - **Redis Client**: `redis_client` for interacting with Redis.
  - **WebSockets**: Implemented with `shelf_web_socket` or `web_socket_channel`.
  - **gRPC/REST**: Using gRPC Dart with `shelf` as a REST fallback.

3.2.3 Databases

- **PostgreSQL**: Used for storing structured data such as user and session information.
- **Redis**: Utilized for caching, managing active sessions, and real-time data handling.

3.3 Key Specifications

3.3.1 Frontend Design

The frontend of the Time Tracking and Management System is implemented using React and is designed to be interactive, user-friendly, and responsive. It acts as the primary interface for user interaction, connecting seamlessly with the backend for data retrieval and real-time updates.

Key Features

- **Interactive UI Components**: The frontend consists of reusable and modular components, such as forms for user input, dashboards for time tracking, and reports.
- **Real-Time Updates**: WebSockets are used to push real-time updates to the UI, allowing users to see changes instantly (e.g., active work session status).
- **Responsive Design**: The frontend is built to adapt to various screen sizes, ensuring usability across desktop and mobile devices.
- **Data Visualization**: Charts and tables are used to present work sessions, time tracking, and reports in an easy-to-understand manner.

Technologies and Libraries Used

- **React**: The main framework for building the UI.
- **Axios** or `Fetch API`: Used for making HTTP requests to the backend (REST/gRPC endpoints).
- **Socket.IO** or native WebSocket API: For handling real-time communication with the backend.
- **Styled Components/SCSS**: For flexible and maintainable CSS styling.
- **Chart.js** or `Recharts`: For data visualization in reports and dashboards.

Frontend Architecture The frontend is organized into a modular architecture to ensure scalability and maintainability:

- **Components Layer**: Contains reusable UI components (e.g., input forms, buttons, modals).
- **Containers**: Components that handle state and pass props to presentational components.
- **Services Layer**: Manages API calls and WebSocket connections.
- **State Management**: Utilizes React's built-in state hooks, or optionally, a library such as Redux for more complex state management.
- **Routing**: Implemented with `react-router-dom` for navigation between pages and maintaining a single-page application (SPA) experience.

User Experience (UX) Design The design prioritizes intuitive navigation and user interaction. Key considerations include:

- **Easy-to-Use Forms**: Simplified forms for user registration, login, and time entry.
- **Dashboards**: Overview pages that display user statistics, current projects, and active work sessions.
- **Notifications**: In-app notifications to remind users to start or stop work sessions or to alert them about upcoming deadlines.

Security Considerations

- **Secure Data Handling**: All user data passed between the frontend and backend is encrypted using TLS.
- **Authentication**: JSON Web Tokens (JWT) are used for user authentication and to secure API endpoints.
- **Input Validation**: Client-side validation ensures that user inputs meet required formats before sending them to the backend.

3.3.2 Backend Implementation

Go Backend

- **ORM**: GORM for database interaction with PostgreSQL.
- **Redis Integration**: `go-redis` for handling real-time operations.
- **WebSockets**: Implemented with `gorilla/websocket`.
- **Communication Protocols**: Initial focus on gRPC, with REST using `net/http` as a fallback.

Dart Backend

- **ORM**: `drift` or `aqueduct` for database operations.
- **Redis Integration**: `redis_client` for caching and real-time functions.
- **WebSockets**: Implemented using `shelf_web_socket` or `web_socket_channel`.
- **Communication Protocols**: gRPC for internal service communication, with `shelf` for REST as a fallback.

3.4 Entities

3.4.1 User

The **User** entity represents an individual who interacts with the system.

- **Attributes**:
  - **id**: UUID, unique identifier for the user.
  - **username**: String, must be unique.
  - **password_hash**: String, stores the hashed password.
  - **email**: String, must be unique.
  - **created_at**: Timestamp, when the user was created.
  - **updated_at**: Timestamp, when the user was last updated.

3.4.2 WorkSession

The **WorkSession** entity tracks a specific period of work for a user.

- **Attributes**:
  - **id**: UUID, unique identifier for the work session.
  - **user_id**: UUID, foreign key referencing the **User**.
  - **start_time**: Timestamp, when the session starts.
  - **end_time**: Timestamp, when the session ends.
  - **breaks**: JSON array representing break intervals.
  - **project_id**: UUID, optional, foreign key referencing **Project**.
  - **description**: Text, optional, describing the session.

3.4.3 Project

The **Project** entity represents a project to which work sessions can be assigned.

- **Attributes**:
  - **id**: UUID, unique identifier for the project.
  - **name**: String, name of the project.
  - **description**: Text, optional, describing the project.
  - **created_at**: Timestamp, when the project was created.
  - **updated_at**: Timestamp, when the project was last updated.
  - **owner_id**: UUID, foreign key referencing the **User** who owns the project.

3.4.4 BreakInterval

The **BreakInterval** entity represents a break within a work session.

- **Attributes**:
  - **start_time**: Timestamp, when the break starts.
  - **end_time**: Timestamp, when the break ends.
- **Note**: This can be stored as a JSON array or in a separate table linked to **WorkSession**.

3.4.5 NotificationSetting

The **NotificationSetting** entity holds user-specific notification preferences.

- **Attributes**:
  - **id**: UUID, unique identifier for the setting.
  - **user_id**: UUID, foreign key referencing the **User**.
  - **notification_type**: Enum (e.g., Email, Push).
  - **enabled**: Boolean, whether the notification is enabled.

3.4.6 Report

The **Report** entity represents generated reports of work sessions.

- **Attributes**:
  - **id**: UUID, unique identifier for the report.
  - **user_id**: UUID, foreign key referencing the **User**.
  - **generated_at**: Timestamp, when the report was generated.
  - **type**: Enum (e.g., Daily, Weekly).
  - **content**: JSON, containing the report summary.

3.4.7 Role

The **Role** entity defines the different roles that users can have in the system.

- **Attributes**:
  - **id**: UUID, unique identifier for the role.
  - **name**: String, name of the role (e.g., Admin, User).
  - **description**: Text, optional, describing the role.

3.4.8 Permission

The **Permission** entity specifies the actions that can be performed in the system.

- **Attributes**:
  - **id**: UUID, unique identifier for the permission.
  - **name**: String, name of the permission (e.g., CREATE_PROJECT, VIEW_REPORT).
  - **description**: Text, optional, describing the permission.

3.4.9 RolePermission

The **RolePermission** linking table associates roles with permissions.

- **Attributes**:
  - **role_id**: UUID, foreign key referencing **Role**.
  - **permission_id**: UUID, foreign key referencing **Permission**.

3.4.10 UserRole

The **UserRole** linking table associates users with their roles.

- **Attributes**:
  - **user_id**: UUID, foreign key referencing **User**.
  - **role_id**: UUID, foreign key referencing **Role**.

4 System Design

4.1 Overview

The system design for the Time Tracking and Management System follows the principles of Domain-Driven Design (DDD) and Clean Architecture. This approach ensures that the system is modular, maintainable, and scalable. The system is divided into well-defined layers, separating concerns and allowing for future enhancements with minimal changes to the core logic.

4.2 Architectural Design

- **Domain Layer**: Contains the core business logic and entities, including **User**, **WorkSession**, **Project**, **Role**, and **Permission**.
- **Application Layer**: Hosts use cases and application services, such as **UserService**, **TimeTrackingService**, and **RoleManagementService**, that coordinate business logic.
- **Infrastructure Layer**: Implements the database interaction using ORMs (e.g., GORM for Go, drift/aqueduct for Dart), caching with Redis, and integration with PostgreSQL.
- **Interface Adapters Layer**: Contains controllers, WebSocket handlers, and REST/gRPC endpoints that interact with the frontend and external services.
- **Frontend (React)**: Serves as the interface between the user and the backend, handling user input and displaying data in a user-friendly manner.

4.3 Module Design

Each component of the system is broken down into modules to promote modularity and reusability.

4.3.1 User Management Module

- **Description**: Handles user registration, authentication, profile management, and password recovery.
- **Components**:
  - **UserController**: Exposes API endpoints for user-related operations.
  - **UserService**: Contains business logic for user management.
  - **UserRepository**: Provides data access to the **User** table.

4.3.2 Time Tracking Module

- **Description**: Manages work sessions, including starting, stopping, and manual entry of time.
- **Components**:
  - **TimeTrackingController**: Handles API requests for time tracking.
  - **TimeTrackingService**: Implements logic for managing work sessions.
  - **WorkSessionRepository**: Manages database interactions for work sessions.

4.3.3 Project Management Module

- **Description**: Facilitates the creation and management of projects that users can be assigned to.
- **Components**:
  - **ProjectController**: Provides endpoints for project-related operations.
  - **ProjectService**: Contains business logic for managing projects.
  - **ProjectRepository**: Accesses data in the **Project** table.

4.3.4 Role and Permission Module

- **Description**: Manages role-based access control (RBAC) and permissions for users.
- **Components**:
  - **RoleController**: Handles role-related requests.
  - **PermissionService**: Manages permissions and role-permission mappings.
  - **RoleRepository** and **PermissionRepository**: Provide data access for roles and permissions.

4.4 Data Flow

The system follows a clear data flow to ensure separation of concerns:

1. **Frontend Request**: The user interacts with the React frontend, which sends requests (e.g., via WebSockets or HTTP).
2. **Controller Layer**: The request is received by a controller (e.g., **UserController**), which forwards it to the appropriate service.
3. **Application Layer**: The service (e.g., **UserService**) processes the request, applies business logic, and interacts with repositories.
4. **Infrastructure Layer**: The repository interacts with the database or caching layer (PostgreSQL or Redis) to retrieve or modify data.
5. **Response**: The result is sent back to the controller and then to the frontend for display to the user.

4.5 Technologies Used

- **Backend (Go)**:
  - **ORM**: GORM for database interaction.
  - **Redis Client**: `go-redis` for caching.
  - **WebSocket and HTTP Handling**: `gorilla/websocket` and `gorilla/mux`.
- **Backend (Dart)**:
  - **ORM**: `drift` or `aqueduct`.
  - **Redis Client**: `redis_client`.
  - **WebSocket Handling**: `shelf_web_socket`.
- **Frontend (React)**:
  - **HTTP Requests**: `Axios` or Fetch API.
  - **WebSocket Library**: `Socket.IO` or native WebSocket API.

4.6 Database Objects (DBOs)

4.6.1 User Table

The **User Table** stores information about users in the system.

- **Columns**:
  - **id**: UUID, primary key.
  - **username**: String, unique.
  - **password_hash**: String.
  - **email**: String, unique.
  - **created_at**: Timestamp, not null.
  - **updated_at**: Timestamp, not null.

4.6.2 WorkSession Table

The **WorkSession Table** keeps track of individual work sessions for each user.

- **Columns**:
  - **id**: UUID, primary key.
  - **user_id**: UUID, foreign key referencing **User**.
  - **start_time**: Timestamp, not null.
  - **end_time**: Timestamp, nullable.
  - **breaks**: JSON, represents break intervals.
  - **project_id**: UUID, foreign key referencing **Project**, nullable.
  - **description**: Text, optional.

4.6.3 Project Table

The **Project Table** contains data related to projects that users can be assigned to.

- **Columns**:
  - **id**: UUID, primary key.
  - **name**: String, not null.
  - **description**: Text, optional.
  - **created_at**: Timestamp, not null.
  - **updated_at**: Timestamp, not null.
  - **owner_id**: UUID, foreign key referencing **User**.

4.6.4 BreakInterval Table

The **BreakInterval Table** represents breaks within a work session.

- **Columns**:
  - **id**: UUID, primary key.
  - **work_session_id**: UUID, foreign key referencing **WorkSession**.
  - **start_time**: Timestamp, not null.
  - **end_time**: Timestamp, not null.

4.6.5 NotificationSetting Table

The **NotificationSetting Table** holds user-specific notification preferences.

- **Columns**:
  - **id**: UUID, primary key.
  - **user_id**: UUID, foreign key referencing **User**.
  - **notification_type**: Enum, defines the type of notification (e.g., Email, Push).
  - **enabled**: Boolean, whether the notification is enabled or not.

4.6.6 Report Table

The **Report Table** stores generated reports related to user work sessions.

- **Columns**:
  - **id**: UUID, primary key.
  - **user_id**: UUID, foreign key referencing **User**.
  - **generated_at**: Timestamp, when the report was generated.
  - **type**: Enum, type of report (e.g., Daily, Weekly).
  - **content**: JSON, contains the report details.

4.6.7 Role Table

The **Role Table** defines different roles that can be assigned to users.

- **Columns**:
    - **id**: UUID, primary key.
    - **name**: String, unique.
    - **description**: Text, optional.

4.6.8 Permission Table

The **Permission Table** holds information about various actions that can be performed in the system.

- **Columns**:
    - **id**: UUID, primary key.
    - **name**: String, unique.
    - **description**: Text, optional.

4.6.9 RolePermission Table

The **RolePermission Table** is a linking table associating roles with their respective permissions.

- **Columns**:
    - **role_id**: UUID, foreign key referencing **Role**.
    - **permission_id**: UUID, foreign key referencing **Permission**.

4.6.10 UserRole Table

The **UserRole Table** associates users with their assigned roles.

- **Columns**:
    - **user_id**: UUID, foreign key referencing **User**.
    - **role_id**: UUID, foreign key referencing **Role**.

5 Detailed Design for the Go Backend

5.1 Project Structure for DDD and Clean Architecture in Go

The application of Domain-Driven Design (DDD) and Clean Architecture in Go promotes a clear separation of concerns, which facilitates maintainability and scalability. A proposed project structure is outlined below:

```
backend-go/
|-- cmd/                    # Entry points of the application
|   ‘-- your_app/
|       ‘-- main.go         # Main program
|-- internal/               # Non-exported code
|   |-- domain/             # Domain layer
|   |   |-- entities/       # Domain models
|   |   |-- repositories/   # Repository interfaces
|   |   ‘-- services/       # Domain services
|   |-- application/        # Application layer
|   |   |-- usecases/       # Use cases
|   |   ‘-- dto/            # Data Transfer Objects
|   |-- infrastructure/     # Infrastructure layer
|   |   |-- persistence/    # Repository implementations
|   |   |-- external/       # External services
|   |   ‘-- config/         # Configuration management
|   ‘-- interfaces/         # Interface layer
|       |-- http/           # HTTP handlers
|       ‘-- cli/            # CLI interfaces
|-- pkg/                    # Public packages
|   ‘-- logger/             # Example of a reusable package
|-- go.mod                  # Module definition
‘-- go.sum                  # Dependencies
```

5.1.1 Explanation of the Structure

- **cmd/**: Contains the main entry point of the application.
- **internal/domain/**: Defines the core domain, including entities and domain services.
- **internal/application/**: Implements application logic and orchestrates domain objects.
- **internal/infrastructure/**: Contains technical details such as database access and external API integrations.
- **internal/interfaces/**: Provides communication interfaces like HTTP controllers.
- **pkg/**: Public packages that can be reused by other projects.

5.1.2 Implementation Highlights

- **Entities**: Define ‘User‘, ‘WorkSession‘, ‘Project‘, and ‘Role‘ as Go structs.
- **Repositories**: Implement interfaces for data access in the ‘domain/repositories/‘ and actual implementations in ‘infrastructure/persistence/‘.
- **Services**: Create domain services in ‘domain/services/‘ to encapsulate business logic.
- **HTTP Handlers**: Define handlers in ‘interfaces/http/‘ to handle HTTP requests and map them to use cases.

5.2 Best Practices and Principles

- **Dependency Rule**: Inner layers should not depend on outer layers.
- **Interface Definition**: Define repository interfaces in ‘domain/repositories/‘ and implement them in ‘infrastructure/persistence/‘.
- **Testability**: Write unit tests for each layer, ensuring high code coverage.

6 Detailed Design for the Dart Backend

6.1 Project Structure for DDD and Clean Architecture in Dart

The Dart backend, potentially built using the **Shelf** framework, can be organized similarly to Go for consistency and maintainability:

```
backend-dart/
|-- bin/
|   ‘-- main.dart           # Entry point of the application
|-- lib/
|   |-- domain/             # Domain layer
|   |   |-- entities/       # Domain models
|   |   |-- repositories/   # Repository interfaces
|   |   ‘-- services/       # Domain services
|   |-- application/        # Application layer
|   |   |-- usecases/       # Use cases
|   |   ‘-- dto/            # Data Transfer Objects
|   |-- infrastructure/     # Infrastructure layer
|   |   |-- persistence/    # Repository implementations
|   |   |-- external/       # External services
|   |   ‘-- config/         # Configuration management
|   ‘-- interfaces/         # Interface layer
|       |-- http/           # HTTP controllers
|       ‘-- cli/            # CLI interfaces
|-- pubspec.yaml            # Package configuration
‘-- test/                   # Tests
```

6.1.1 Explanation of the Structure

- **bin/**: Entry point of the application (‘main.dart‘).
- **lib/domain/**: Contains the core domain models and repository interfaces.
- **lib/application/**: Hosts use cases and orchestrates interactions between domain models.
- **lib/infrastructure/**: Implements repository interfaces and external service interactions.
- **lib/interfaces/**: Defines HTTP controllers and CLI interfaces.
- **test/**: Contains unit and integration tests for all modules.

6.1.2 Implementation Highlights

- **Entities**: Define classes like ‘User‘, ‘WorkSession‘, ‘Project‘, and ‘Role‘ in ‘domain/entities/‘.
- **Repository Interfaces**: Place in ‘domain/repositories/‘ and implement in ‘infrastructure/persistence/‘.
- **Services**: Implement business logic in ‘domain/services/‘.
- **HTTP Controllers**: Use the Shelf package to define routes and request handlers in ‘interfaces/http/‘.

6.2 Best Practices and Principles

- **Dependency Injection**: Ensure that dependencies are injected to keep layers decoupled.
- **Testing**: Write unit and integration tests to cover business logic and infrastructure components.
- **Functional Programming Practices**: Use ‘fpdart‘ for functional constructs to make the code more predictable and easier to test.

7 API Endpoints

7.1 Overview

The API for the Time Tracking and Management System is designed using RESTful principles to provide clear and efficient communication. The main entities covered include Users, Projects, Time Entries, Clients, Tasks, and Notifications. This document outlines the key endpoints, ensuring a consistent, secure, and scalable interface.

7.2 API Endpoints

7.2.1 1. Authentication and Authorization

- **Register a New User**
  - **Endpoint**: `POST /api/auth/register`
  - **Request Body**:

    ```
    {
      "name": "Max Mustermann",
      "email": "max@example.com",
      "password": "geheimespasswort"
    }
    ```

  - **Response**: `201 Created` on success, `400 Bad Request` on validation error.
- **User Login**
  - **Endpoint**: `POST /api/auth/login`
  - **Request Body**:

    ```
    {
      "email": "max@example.com",
      "password": "geheimespasswort"
    }
    ```

  - **Response**: `200 OK` with JWT token on success, `401 Unauthorized` on failure.
- **Logout User**
  - **Endpoint**: `POST /api/auth/logout`
  - **Response**: `200 OK` on success.
- **Get Current User**
  - **Endpoint**: `GET /api/auth/me`
  - **Response**: `200 OK` with user details.

7.2.2 2. User Management

- **Get All Users (Admin Only)**
  - **Endpoint**: `GET /api/users`
  - **Response**: `200 OK` with a list of users.
- **Get Single User**
  - **Endpoint**: `GET /api/users/{userId}`
  - **Response**: `200 OK` with user details, `404 Not Found` if user does not exist.
- **Update User**
  - **Endpoint**: `PUT /api/users/{userId}`
  - **Response**: `200 OK` on success, `400 Bad Request` on validation error.

- **Delete User**
  - **Endpoint**: `DELETE /api/users/{userId}`
  - **Response**: `200 OK` on success, `404 Not Found` if user does not exist.

7.2.3 3. Project Management

- **Get All Projects**
  - **Endpoint**: `GET /api/projects`
  - **Response**: `200 OK` with a list of projects.
- **Create New Project**
  - **Endpoint**: `POST /api/projects`
  - **Request Body**:

    ```
    {
      "name": "New Project",
      "clientId": "12345",
      "description": "Project description"
    }
    ```

  - **Response**: `201 Created` on success.
- **Get Single Project**
  - **Endpoint**: `GET /api/projects/{projectId}`
  - **Response**: `200 OK` with project details.
- **Update Project**
  - **Endpoint**: `PUT /api/projects/{projectId}`
  - **Response**: `200 OK` on success.
- **Delete Project**
  - **Endpoint**: `DELETE /api/projects/{projectId}`
  - **Response**: `200 OK` on success.

7.2.4 4. Time Entry Management

- **Get All Time Entries**
  - **Endpoint**: `GET /api/time-entries`
  - **Query Parameters**: `projectId, dateFrom, dateTo`
  - **Response**: `200 OK` with a list of time entries.
- **Create New Time Entry**
  - **Endpoint**: `POST /api/time-entries`
  - **Request Body**:

    ```
    {
      "projectId": "12345",
      "taskId": "67890",
      "startTime": "2023-10-01T08:00:00Z",
      "endTime": "2023-10-01T10:00:00Z",
      "description": "Work session description"
    }
    ```

  - **Response**: `201 Created` on success.
- **Update Time Entry**

  - **Endpoint**: `PUT /api/time-entries/{entryId}`
  - **Response**: `200 OK` on success.

- **Delete Time Entry**
  - **Endpoint**: `DELETE /api/time-entries/{entryId}`
  - **Response**: `200 OK` on success.

7.2.5 5. Client Management (Optional)

- **Get All Clients**
  - **Endpoint**: `GET /api/clients`
  - **Response**: `200 OK` with a list of clients.
- **Create New Client**
  - **Endpoint**: `POST /api/clients`
  - **Request Body**:

    ```
    {
      "name": "Client Name",
      "contactInfo": "Contact details"
    }
    ```

  - **Response**: `201 Created` on success.
- **Get Single Client**
  - **Endpoint**: `GET /api/clients/{clientId}`
  - **Response**: `200 OK` with client details.

7.2.6 6. Task Management (Optional)

- **Get All Tasks**
  - **Endpoint**: `GET /api/tasks`
  - **Response**: `200 OK` with a list of tasks.
- **Create New Task**
  - **Endpoint**: `POST /api/tasks`
  - **Request Body**:

    ```
    {
      "name": "Task Name",
      "description": "Task description"
    }
    ```

  - **Response**: `201 Created` on success.
- **Update Task**
  - **Endpoint**: `PUT /api/tasks/{taskId}`
  - **Response**: `200 OK` on success.

7.2.7 7. Reporting

- **Generate Report**
  - **Endpoint**: `GET /api/reports`
  - **Query Parameters**: `userIds, dateFrom, dateTo, projectId`
  - **Response**: `200 OK` with report data.

7.2.8 8. Notification Settings

- **Update Notification Settings (Bitmask)**
  - **Endpoint**: `PUT /api/notifications/settings`
  - **Request Body**:

    ```
    {
      "userId": "uuid",
      "notificationSettings": 5  // Example bitmask
    }
    ```

  - **Response**: `200 OK` on success.
- **Get Notification Settings**
  - **Endpoint**: `GET /api/notifications/settings/{userId}`
  - **Response**: `200 OK` with the current bitmask settings.

7.3 Security Considerations

- **Authentication**: All protected endpoints require a valid JWT token.
- **Role-Based Access Control**: Specific endpoints may require roles such as ‘Admin‘ or ‘Manager‘.
- **HTTPS**: All API calls should be made over HTTPS to ensure data security.

7.4 Error Handling

- **Consistent Error Responses**:

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input for 'email'.",
    "details": {
      "email": "This field cannot be empty."
    }
  }
}
```