SOFTWARE REQUIREMENTS SPECIFICATION

for

TIME TRACKING AND MANAGEMENT SYSTEM

Version 1.2

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

Module: Concepts of Programming Languages

January 5, 2025

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 **ActaTempus Time Tracking and Management System** is a full-stack web application with the following main components:

- **Frontend**: Built with Next.js and React, utilizing HTTP for RESTful APIs and WebSockets for real-time communication.
- **Backend**: Implemented in Go, with support for scalable service layers, data management, and business logic.
- **Databases**: PostgreSQL serves as the primary relational data store for all persistent data.
- **Communication Protocols**: REST is the primary communication method, with WebSockets used for real-time updates where needed.

3.2 Technology Stack

3.2.1 Frontend

- **React/Next.js**: Frameworks for building interactive and responsive user interfaces.
- **RxJS**: Functional reactive programming for state management and event streams.
- **Axios**: For making HTTP requests to backend APIs.
- **WebSocket API**: Native API for bi-directional communication between frontend and backend.

3.2.2 Backend

- **Go Backend**:
  - **ORM**: Prisma Go and Dart adapter for PostgreSQL integration.
  - **Web Framework**: `gin-gonic` (Go) and `shelf` (Dart) for handling HTTP requests.
  - **Authentication**: JSON Web Tokens (JWT) for securing API endpoints.

3.2.3 Database

- **PostgreSQL**: A robust, open-source relational database for structured data storage.

3.3 Key Specifications

3.3.1 Frontend Design

The frontend is designed to provide an intuitive, responsive, and user-friendly interface.

Key Features

- **Modular Components**: Reusable React components for user input, dashboards, and notifications.
- **Real-Time Updates**: WebSocket integration ensures real-time synchronization between frontend and backend.
- **Responsive Design**: Ensures compatibility with desktops, tablets, and mobile devices.
- **Data Visualization**: Utilizes charts and tables for presenting reports and time tracking summaries.

Security Considerations

- **Secure Communication**: All data exchanges are secured using TLS encryption.
- **Authentication and Authorization**: User sessions are secured via JWTs.
- **Input Validation**: Ensures data integrity before submission to the backend.

3.3.2 Backend Implementation

The backend handles business logic, data storage, and secure communication.

Components

- **Service Layer**: Implements the business logic and coordinates with repositories.
- **Repository Layer**: Abstracts data access logic for easy maintenance.
- **DataSource Layer**: Handles direct interaction with PostgreSQL via Prisma adapters.

Key Features

- **Role-Based Access Control (RBAC)**: Restricts actions based on user roles and permissions.
- **Scalable Architecture**: Designed to handle growing user bases and data loads.
- **Real-Time Updates**: Implements WebSocket endpoints for real-time functionality.

3.4 Entities

3.4.1 User

Represents a user of the system.

- **id**: UUID, unique identifier.
- **name**: String, not null.
- **email**: String, unique.
- **password_hash**: String, stores the hashed password.
- **created_at**, **updated_at**: Timestamps.

3.4.2 TimeEntry

Tracks user work sessions.

- **id**: UUID, unique identifier.
- **user_id**: UUID, foreign key referencing **User**.
- **start_time**, **end_time**: Timestamps.
- **description**: Optional text description.
- **project_id**: UUID, foreign key referencing **Project**.

3.4.3 Project

Represents a project that users can be assigned to.

- **id**: UUID, unique identifier.
- **name**: String, not null.
- **description**: Optional text description.
- **created_at**, **updated_at**: Timestamps.
- **owner_id**: UUID, foreign key referencing **User**.

3.4.4 Role and Permission

Manages access control.

- **Role Table**: Defines roles.
- **Permission Table**: Lists permissions.
- **RolePermission Table**: Links roles and permissions.
- **UserRole Table**: Links users to roles.

4 System Design

4.1 Overview

The system design for the **ActaTempus Time Tracking and Management System** is rooted in principles of Domain-Driven Design (DDD) and Clean Architecture. This ensures a modular, maintainable, and scalable system, with clearly defined layers that simplify future enhancements while preserving the core logic.

4.2 Architectural Design

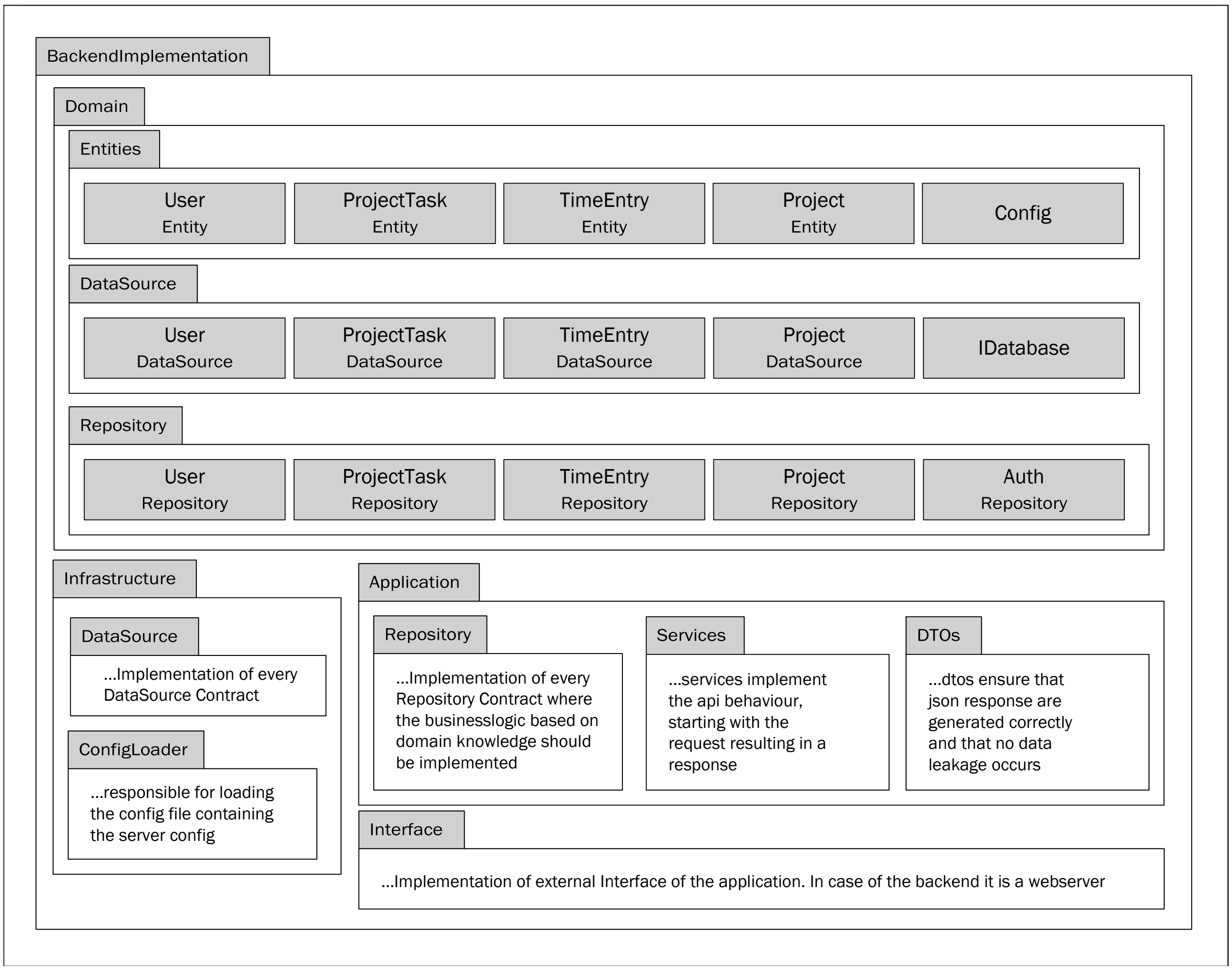
BackendImplementation
Domain
Entities
User Entity
ProjectTask Entity
TimeEntry Entity
Project Entity
Config
DataSource
User DataSource
ProjectTask DataSource
TimeEntry DataSource
Project DataSource
IDatabase
Repository
User Repository
ProjectTask Repository
TimeEntry Repository
Project Repository
Auth Repository
Infrastructure
DataSource
...Implementation of every DataSource Contract
ConfigLoader
...responsible for loading the config file containing the server config
Application
Repository
...Implementation of every Repository Contract where the businesslogic based on domain knowledge should be implemented
Services
...services implement the api behaviour, starting with the request resulting in a response
DTOs
...dtos ensure that json response are generated correctly and that no data leakage occurs
Interface
...Implementation of external Interface of the application. In case of the backend it is a webserver

Figure 1: Backend Architecture Concept

The architecture displayed in figure 1 can be described as following:

- **Domain Layer**: Encapsulates the core business logic and entities such as **User**, **TimeEntry**, **Project**, **Role**, and **Permission**.
- **Application Layer**: Contains use cases and services like **UserService**, **TimeEntryService**, and **RoleManagementService**, coordinating operations.
- **Infrastructure Layer**: Manages database interaction using Prisma adapters for PostgreSQL and provides additional abstractions for external systems.
- **Interface Adapters Layer**: Hosts server logic that mediate interactions between the backend, the client.
- **Frontend App**: Built with Next.js and React, utilizing `fp-ts` and RxJS to ensure a functional and responsive user experience.

4.3 Module Design

4.3.1 User Management Module

- **Description**: Manages user registration, authentication, profile updates, and password recovery.
- **Components**:
  - **UserService**: Implements business logic for user management.
  - **UserRepository**: Interfaces with the **UserDataSource** for database access.
  - **UserDataSource**: Provides database interaction for user data using Prisma.

4.3.2 Authentication Module

- **Description**: Handles user authentication and session management.
- **Components**:
  - **AuthService**: Manages token generation, validation, and revocation.
  - **AuthRepository**: Interfaces with the token storage system.

4.3.3 Time Entry Module

- **Description**: Tracks time entries, including start/stop actions and manual entry.
- **Components**:
  - **TimeEntryService**: Implements logic for time tracking operations.
  - **TimeEntryRepository**: Accesses **TimeEntryDataSource** for data persistence.
  - **TimeEntryDataSource**: Interacts with the database to manage time entry records.

4.3.4 Project Management Module

- **Description**: Facilitates project creation and assignment.
- **Components**:
  - **ProjectService**: Implements business logic for managing projects.
  - **ProjectRepository**: Interfaces with the **ProjectDataSource** for database operations.
  - **ProjectDataSource**: Handles direct database interactions for project-related data.

4.3.5 Role and Permission Module

- **Description**: Implements role-based access control (RBAC) for managing permissions.
- **Components**:
  - **RoleManagementService**: Manages roles and their associated permissions.
  - **RoleRepository** and **PermissionRepository**: Interface with their respective DataSources.
  - **RoleDataSource** and **PermissionDataSource**: Provide database access for roles and permissions.

4.4 Data Flow

1. **Frontend Request**: The user interacts with the React-based frontend, which sends HTTP requests to the backend.
2. **Service Layer**: Handles the requests, applying business logic and orchestrating operations.
3. **Repository Layer**: Acts as an intermediary between the service layer and the data sources, abstracting persistence logic.
4. **DataSource Layer**: Directly interacts with PostgreSQL using Prisma for data manipulation.
5. **Response**: The processed result is returned to the service layer and forwarded to the frontend for display.

4.5 Database Design

The database schema supports all core entities and their relationships.

4.5.1 User Table

Stores user information.

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

4.5.2 TimeEntry Table

Tracks user time entries.

- **Columns**:
  - **id**: UUID, primary key.
  - **user_id**: UUID, foreign key referencing **User**.
  - **start_time**, **end_time**: Timestamps.
  - **description**: Text (optional).
  - **project_id**: UUID, foreign key referencing **Project**.

4.5.3 Project Table

Stores project data.

- **Columns**:
  - **id**: UUID, primary key.
  - **name**, **description**: Strings.
  - **owner_id**: UUID, foreign key referencing **User**.

4.5.4 Role and Permission Tables

Manage RBAC functionality.

- **Role Table**:
  - **id**: UUID, primary key.
  - **name**: String, unique.
- **Permission Table**:
  - **id**: UUID, primary key.
  - **name**: String, unique.
- **RolePermission Table**: Links roles and permissions.

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
|   |   |-- repositories/   # Implementaton of Repositories
|   |   `-- services/       # Implementation of API Endpoints
|   |       `-- dto/        # Data Transfer Objects
|   |-- infrastructure/     # Infrastructure layer
|   |   |-- data/           # DataSources implementations
|   |   `-- config/         # Configuration management
|   `-- interfaces/         # Interface layer
|       `-- http/           # HTTP handlers
|-- 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
|   |   |-- repositories/   # Implementaton of Repositories
|   |   ‘-- services/       # Implementation of API Endpoints
|   |       ‘-- dto/        # Data Transfer Objects
|   |-- infrastructure/     # Infrastructure layer
|   |   |-- data/           # DataSource implementations
|   |   ‘-- config/         # Configuration management
|   ‘-- interfaces/         # Interface layer
|       ‘-- http/           # HTTP controllers
|-- 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 in production 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."
    }
  }
}
```