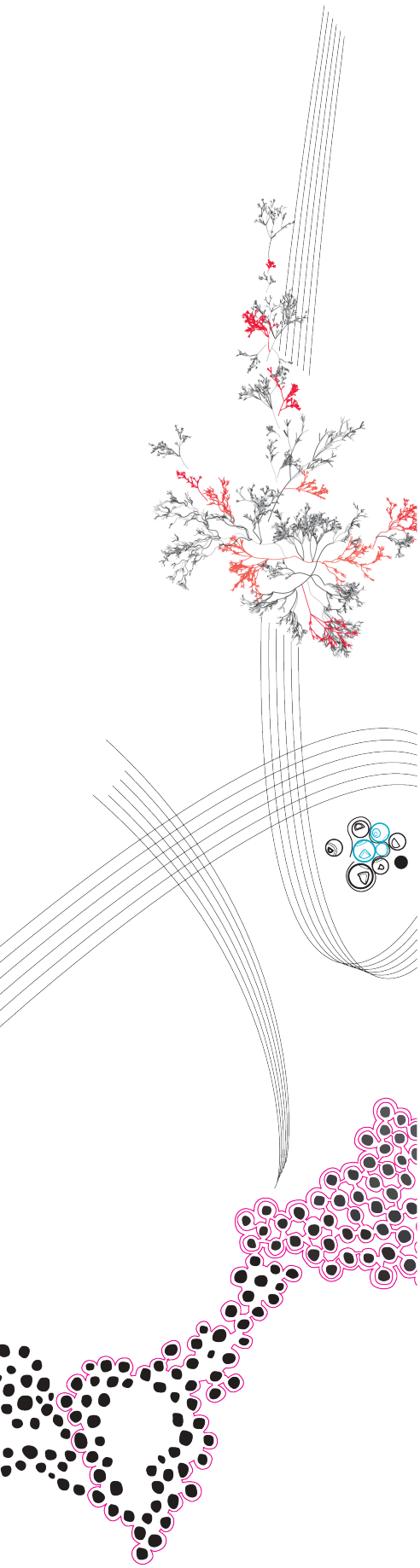

M11 - Design Project

Design Report

Mohamed Soliman
Oliver Hnat
Jorik Van Veen
Naut De Vroome
Arseniy Tokarev
Eren Atakan

Supervisor: Dr. Luiz Bonino da Silva Santos

November, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

1 Introduction

1.1 Background and Context

Accurate reporting of working hours plays a crucial role in modern organizations and companies, especially startups, which need to monitor multiple projects simultaneously with limited administrative resources. A Dutch startup, Hovyu, specializing in innovative processes for carbon dioxide capture, identified a growing need to organize and track working hours. As the company started to expand, it became increasingly difficult to maintain accurate reporting of working hours, employee involvement, and task distribution.

Because the company lacked a centralized platform, the reporting was mainly done manually using spreadsheets, which wasn't effective and led to errors and a lack of clarity. This not only made planning and internal accountability more difficult but also hindered accurate reporting for administrative tasks such as payroll calculations.

The team aims to create an online tracking platform to alleviate these challenges using a simple web solution for employees and management. The core functionalities include task management, real-time hour logging, and reporting features. This would allow the company to grow scalably with improved management backed by data.

1.2 Problem Definition

As of right now, Hovyu is missing a centralized system to track and manage working hours of employees. Because time registration is currently handled manually, it leads to inconsistency, incompleteness, and occasional delays in data entry. Consequently, management doesn't always have access to the full picture of what's going on with projects and the actual distribution of work and time across tasks.

Moreover, employees working across multiple projects have no unified system to switch between assignments and view their total workload. The main impact of this inefficiency is increased difficulty in decision-making for managers due to decreased accuracy and inability to evaluate overall performance.

The main issues can be summarized as follows:

1. **Missing centralization**
   - Time-tracking and project info are scattered across multiple sources.
2. **Limited transparency**
   - Management can't easily see or monitor logged hours per team member, project, or task.
3. **Manual errors**
   - Manual entries pose a greater likelihood of errors and inconsistency.
4. **No mobility**
   - Employees cannot register hours via their mobile phones.

To address these problems, the system must be scalable and secure to support both automated and manual tracking, with a clear distinction of roles between users and admins.

1.3 Project Goals and Objectives

The primary goal of this project is to develop a web platform for tracking working hours across the team, enabling Hovyu to accurately monitor, control, and analyze time spent on projects

and tasks. The system must optimize administrative tasks, increase data reliability, and provide visualization and export capabilities to support decision-making.

Main goals of the project include:

1. **Easy work hour tracking**
   - Provide employees with an intuitive and easy-to-use interface for registering, viewing, and editing their working hours across tasks and projects.
2. **Administrative control**
   - Enable management to manage tasks and projects.
3. **Reduce forgotten and false hour tracking**
   - Make it easy for employees to log hours in real-time.
4. **Visualization and export of data**
   - Provide options for admins to view data as graphs or tables, with the ability to export them as spreadsheets for payroll calculations and analysis.
5. **Safety and scalability**
   - Ensure secure authentication and scalable storage, allowing future integrations with phones or external systems.

Ultimately, the system should assist Hovyu in reducing administrative workload, increasing the accuracy of working hour reporting, and creating a transparent foundation for evaluating employee productivity and project effectiveness.

1.4 Stakeholders

The Hovyu time tracking system will only be used internally by the Hovyu company. Within the Hovyu company there are two stakeholders that can be identified within the domain of the product. These are the emplyees and management. For our product, these will be called ‘users’ and ‘admins’ respecivly. The admins are a subset of the users, meaning that every admin is also a user, but not every user is an admin. Users want to have an easy way to declare their work hours and a clear overview of their declared work hours. Admins want to monitor the work hours of all users and make changes if needed. Admins want to export monthly work-hour summaries in formats suitable for, for example, bookkeeping and payroll processing.

2 Requirements Analysis

This section outlines the functional and non-functional requirements of the HOVYU Time Tracking Platform. The goal of this stage is to clearly define what the system should accomplish and how it should behave from both the user's and the administrator's perspectives. The requirements were gathered through discussions with the client, analysis of existing time-tracking tools, and feedback from potential users within the company. To ensure prioritization and clarity, the requirements are categorized using the **MoSCoW** method identifying which features are **Must**, **Should**, **Could**, or **Won't** be included in the current development phase.

2.1 User Requirements

Priority	User Story
Must Have	As a user, I want to record the time I spend on tasks and projects through the web interface, so that my working hours are accurately tracked.
Must Have	As a user, I want to be able to edit or update my hours within the current month, so that I can correct mistakes or add missed entries.
Should Have	As a user, I want to view my logged hours in a calendar format, so that I can easily understand how my time is distributed.
Should Have	As a user, I want to use start and stop buttons to track my time automatically, so that I can capture precise work durations.
Could Have	As a user, I want to adjust what time window I see in my calendar, so that I can focus on specific days or weeks.
Won't Have	As a user, I want to use a mobile app for tracking hours.
Must Have	As an admin, I must be able to add, edit, and remove projects.
Must Have	As an admin, I must be able to add, edit, and remove tasks.
Should Have	As an admin, I should have access to a project dashboard displaying aggregated hours per project.
Should Have	As an admin, I should have access to a user dashboard summarizing individual working hours.
Should Have	As an admin, I should be able to export working hours per user.
Should Have	As an admin, I should be able to export working hours per project.

2.2 System Requirements (Functional)

Priority	Requirement
Must Have	The system must allow users to create an account and log in, so that their work hours are linked to a personal profile.
Must Have	The system must securely store user, project, and time entry data in the database.
Must Have	The system must automatically calculate total working hours per project and per user.
Must Have	The system must enforce the grace period logic, allowing time edits only within the current month.

Must Have	The system must use role-based access control to distinguish between user and admin permissions.
Must Have	The system must allow admins to add, edit, and delete tasks and projects.
Should Have	The system should allow admins to view dashboards summarizing project or user activity.
Should Have	The system should provide export functionality for reports in CSV or PDF format.
Should Have	The system should automatically log important actions (such as login attempts, edits, and deletions) for auditing purposes.
Could Have	The system could generate weekly or monthly summary reports.
Could Have	Admins could impersonate users to view or modify their time entries for troubleshooting.
Won't Have	Integration with external payroll or HR systems.

2.3 System Requirements (Non-Functional)

Category	**Requirement**
Usability	The system should have a clear and intuitive user interface.
Performance	The system should record and display time entries instantly.
Security	All user accounts and data must be protected using authentication and authorization mechanisms.
Reliability	The system should function correctly under multiple simultaneous users.
Scalability	The system should support future integration with other company tools.

3 System Architecture and Design

3.1 System Overview

This project has a completely separated frontend and backend, we chose this architecture because one of the nice to have requirements was a mobile app. This way we can repackage the frontend as a mobile app using a webview and ship that, or we can write a separate mobile app that can ineract with the same backend as the web frontend. The frontend is a SPA written in Svelte 5, routing is handled on the client side using svelte5-router. The backend is a Django application using the Django Rest Framework, this gives us a nice REST API and also a corresponding admin panel that lets administrators easily view, update, create, delete and export relevant data. Our database is SQLite, this database is simply to use and deploy, and since we are not working with large datasets the performance is more than good enough for our purposes.

3.2 Database Design and ER Diagram

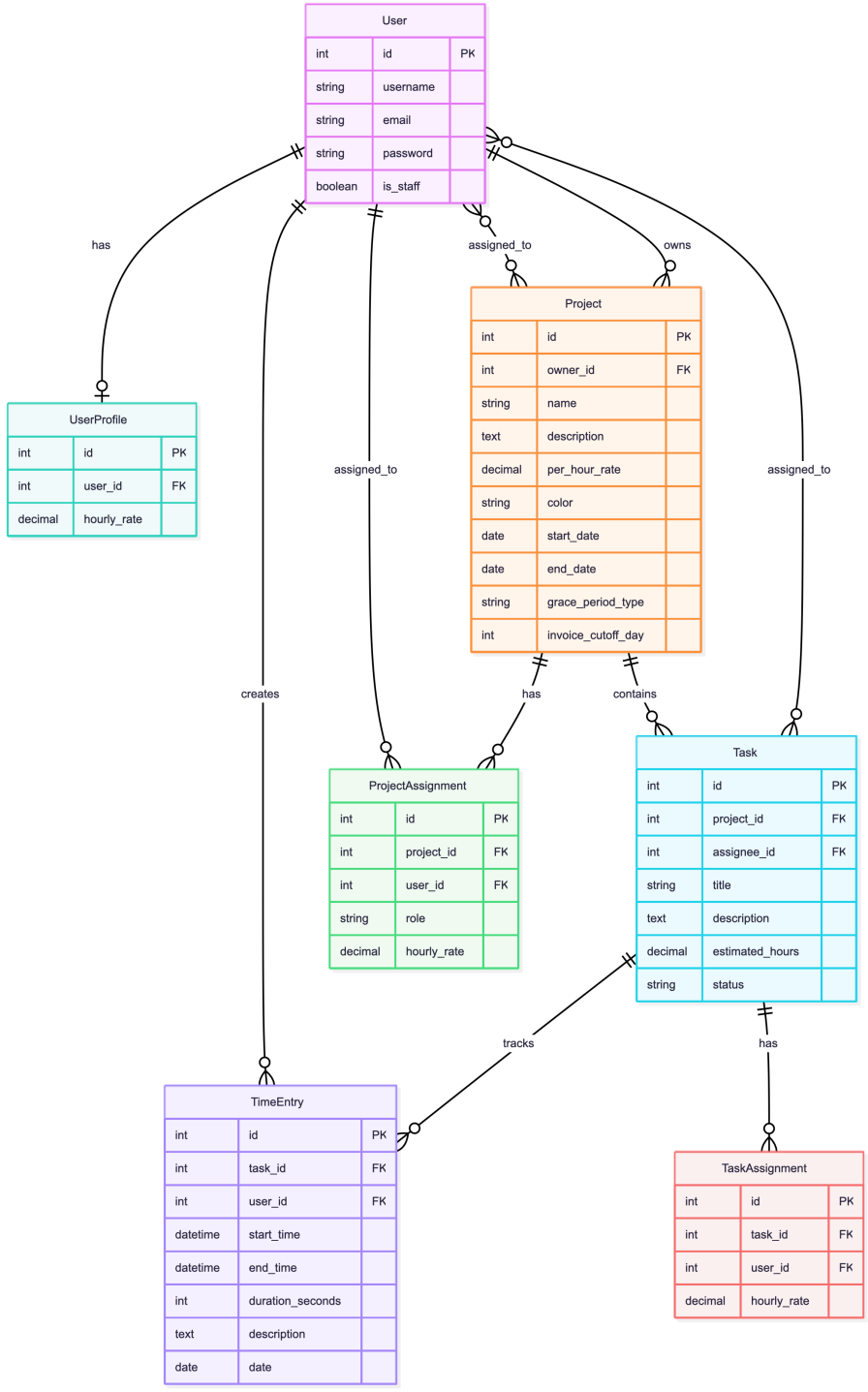

Figure 1: Database Schema

The database for the HOVYU Time Tracking Platform was designed to efficiently manage users, projects, tasks, and logged working hours, while maintaining flexibility for billing and reporting. The design follows a relational model to ensure data consistency, integrity, and scalability. It also supports role-based access control and customizable billing configurations.

3.2.1 User (Django Built-in)

The `User` table is based on Django's built-in authentication model and handles account creation, login, and role management. Each user has basic attributes such as `username`, `email`, and `password`, along with a Boolean `is_staff` field to distinguish administrators from standard users.

3.2.2 UserProfile

Each user has an associated `UserProfile` record through a one-to-one relationship. The profile stores additional user-specific information that is not part of Django's default user model, such as the default hourly rate used for billing calculations. This separation helps maintain Django's modular structure while allowing customization specific to HOVYU's use case.

3.2.3 Project

A `Project` represents a unit of work managed by HOVYU. Each project is owned by a user (the project manager) and may have multiple assigned users through the `ProjectAssignment` table. Projects also store configuration fields such as:

- **Per-hour billing rate** (used as a fallback when no custom rate is defined)
- **Color** for visual identification in the interface
- **Start and end dates** for scheduling
- **Grace period settings** for controlling when time entries can be edited
- **Invoice cutoff day** for payroll or reporting cycles

This structure provides flexibility in how time and billing data are managed per project.

3.2.4 ProjectAssignment

The `ProjectAssignment` table acts as a junction table between `Project` and `User`. It defines which users are assigned to which projects and what role they play (e.g., manager, member, or viewer). This table also supports an optional custom hourly rate override, allowing billing differences between members of the same project.

3.2.5 Task

A `Task` belongs to a single project and represents an individual piece of work. Each task can have one or more assignees through the `TaskAssignment` table. Tasks contain a description, estimated hours, and a status field that indicates whether the task is `todo`, `in_progress`, or `done`. This structure supports both task-level time tracking and progress monitoring.

3.2.6 TaskAssignment

`TaskAssignment` acts as a junction table between `Task` and `User`, allowing multiple users to collaborate on the same task. Similar to `ProjectAssignment`, it includes an optional hourly rate override field, giving fine-grained control over billing rates at the task level.

3.2.7 TimeEntry

The `TimeEntry` table tracks all recorded work hours. Each entry links to a specific task and user, with fields for start and end timestamps, duration in seconds, and a textual description of the work done. The system supports running timers, where the `end_time` remains null until the user stops tracking. Duration and date values are automatically calculated when the entry is saved.

3.2.8 Billing Rate Hierarchy

To handle flexible billing scenarios, the system applies a hierarchical billing rate logic, selecting the most specific rate available:

1. `TaskAssignment.hourly_rate` : highest priority, per-user per-task
2. `ProjectAssignment.hourly_rate` : per-user per-project
3. `UserProfile.hourly_rate` : user default
4. `Project.per_hour_rate` : fallback rate applied globally to the project

This hierarchy ensures accurate and fair billing while allowing flexibility in how different users and tasks are compensated.

3.3 Data Flow Diagram (DFD)

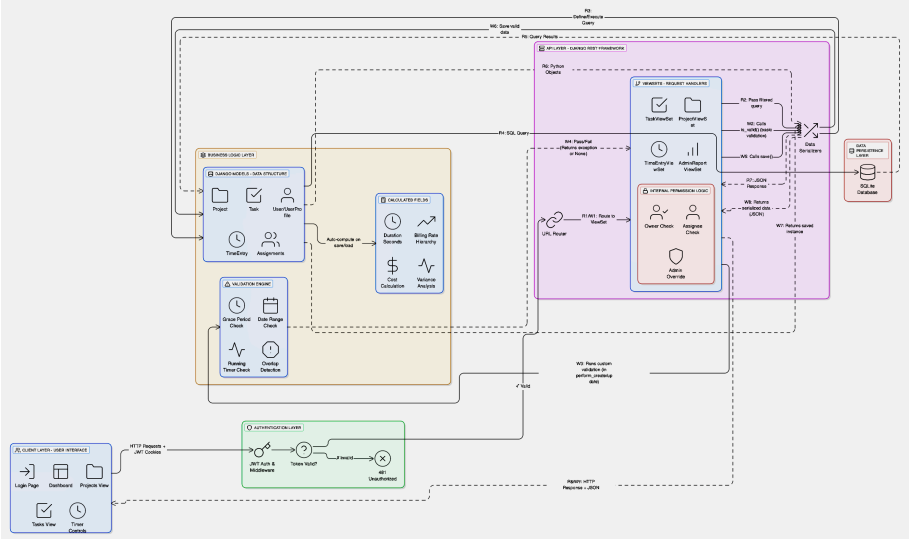

Figure 2: Data Flow Diagram

As you can see in Figure 2, our HOVYU time tracking app utilizes a layered architecture, a standard industry pattern that separates concerns into distinct components such as client and business logic layers. This approach significantly enhances codebase maintainability and enables a more robust, comprehensive security implementation.

3.3.1 Architecture Overview

The system architecture is fully decoupled with the Svelte 5 frontend operating separately from the Django REST Framework backend. All information transfer between them occurs through secure RESTful API calls which are authenticated using JWT over HTTPS.

3.3.2 Client Layer (Blue)

The client layer is made of 5 main views: *Login, Dashboard, Project View, Task View, and Timer Controls.* All user interactions begin here, where they can authenticate themselves, control tracking timers, and check their projects and tasks.

3.3.3 Authentication Layer (Green)

For an API request to be carried out, it first has to go through the authentication layer. This is a minimal layer that contains the JWT authentication logic and a decision point to validate the said tokens. If the token is invalid, it returns a “401 Unauthorized” response. If it’s valid, it allows the request to proceed further with the authenticated user’s context.

3.3.4 API Layer (Purple)

The API layer handles all the HTTP routing and request processing. Incoming requests are forwarded to appropriate viewsets by the help of a URL router, where each viewset has its own internal permission logic that checks for ownership, status of the related assignment, and admin privileges before allowing any operations. Data serializers on this layer act as a translator between JSON and Python objects, ensuring correct transformation with data validation.

3.3.5 Business Logic Layer (Orange)

The core application logic resides here across three components:

- **Django Models:** Defines the data structures like *UserProfile, Task, TimeEntry, and Assignment* models.
- **Validation Engine:** Enforces business rules like *grace period checks, date range validation, running timer detection, and overlap prevention.*
- **Calculated Fields:** Computes derived values/decisions like *duration, billing rate, timer detection, and variance analysis.*

3.3.6 Data Persistence Layer (Red)

All the application data is stored in an SQLite database. While initial versions utilize SQLite for efficiency and simplicity, the system is also designed with PostgreSQL support in mind for production environments.

3.3.7 Request Flow Patterns

3.3.7.1 READ Requests

A read request arrives at the *API Router*. The flow is as follows:

- (R1) The router directs the request to the appropriate *ViewSet*.
- (Internal) The *ViewSet* filters its queryset (*get_queryset()*).
- (R2) The *ViewSet* passes the filtered query to the *Serializer*.
- (R3) The *Serializer* accesses the *Models* (which defines the query).
- (R4) The *Model* (ORM) sends the SQL Query to the *Database*.
- (R5) The *Database* returns Query Results to the *Model*.
- (R6) The *Model* returns Python Objects to the *Serializer*.
- (R7) The *Serializer* formats the objects into a JSON Response and returns it to the *ViewSet*.
- (R8) The *ViewSet* sends the final HTTP Response + JSON back to the Client.

3.3.7.2 WRITE Requests

A write request also arrives at the *API Router*. The validation and save process is as follows:

- (W1) The router directs the request to the appropriate *ViewSet*.
- (W2) The *ViewSet* calls the *Serializer*'s *is_valid()* method for basic validation.
- (W3) The *Serializer* returns a Pass/Fail result to the *ViewSet*.
- (W4) The *ViewSet* (*perform_create/update*) runs custom logic via the *Business Validation engine*.
- (W5) The *Business Validation* returns a Pass/Fail result to the *ViewSet*.
- (W6) The *ViewSet* calls the *Serializer*'s *save()* method.
- (W7) The *Serializer* saves valid data to the *Model* instance.
- (W8) The *Model* sends the Updates to the *Database*.
- (W9) The *Database* Confirms the Update to the *Model*.

- (W10) The *Model* returns the saved instance back to the *Serializer*.
- (W11) The *Serializer* returns the serialized data (JSON) to the *ViewSet*.
- (W12) The *ViewSet* sends the final HTTP Response + JSON back to the Client.

3.4 Exporting

To comply with the two requirement mentioned in Section 2 *"As an admin, I should be able to export working hours per user."* and *"As an admin, I should be able to export working hours per project."* there are two exporting functionality.

The first one is in the fronted. There are two export buttons in the admin panel. One for exporting working hours per user and one for exporting working hours per project. When an admin clicks on one of these buttons a request is sent to the backend to get the relevant data. The backend then generates a CSV file with the relevant data aggragted by month and sends it back to the frontend.

The second exporting functionality is in the backend admin panel. Here admins can select multiple time log entries and export them as a CSV file. This is useful for exporting specific time log entries instead of all time log entries for a user or project.

3.5 Security

Security was an essential consideration in the development of the HOVYU Time Tracking Platform especially since the system stores sensitive user information and directly influences payroll calculations. The following measures were implemented to ensure a secure and trusted platform.

3.5.1 Cross-Site Scripting (XSS) Protection

The front-end of the application is built using Svelte, which provides built-in protection against cross-site scripting attacks. By default, Svelte automatically escapes any content inserted into the Document Object Model (DOM), making it significantly more difficult for malicious scripts to execute. This means that user-generated inputs, such as task names or descriptions, cannot be interpreted as executable code unless explicitly marked as safe.

3.5.2 SQL Injection Prevention

On the back-end, Django's Object-Relational Mapper (ORM) is used to handle all interactions with the database. Django's ORM constructs parameterized queries, which separates user input data from SQL commands. This approach ensures that user data cannot alter the structure of the underlying queries, effectively preventing SQL injection attacks. As a result, even if a user attempts to input malicious SQL code, it will be treated purely as text rather than as part of a database command.

3.5.3 Authentication

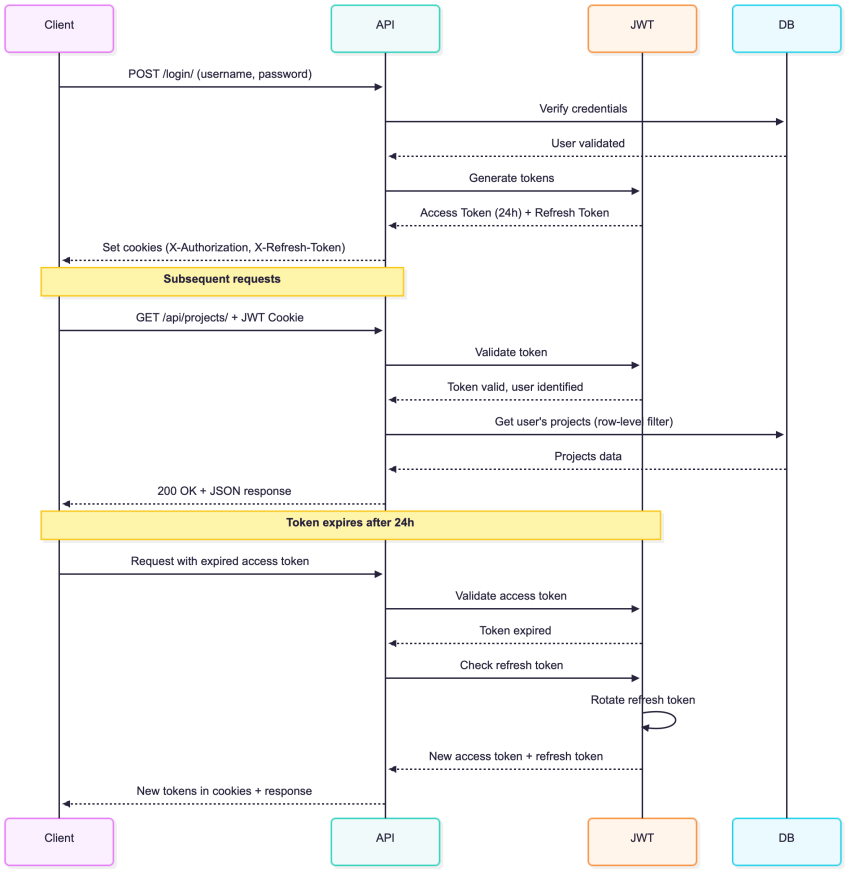

Figure 3: Authentication Flow

User authentication is managed through Django's built-in authentication framework, which provides a robust and secure method for handling user credentials. Django employs strong, salted password hashing algorithms (such as PBKDF2, Argon2, or bcrypt) to ensure that stored passwords are not easily compromised. Salted hashing ensures that even if two users choose the same password, their stored hashes remain unique. This greatly enhances the overall security of user accounts.

In Figure 3, the authentication flow is illustrated. When a user attempts to log in, their credentials are sent to the backend, which verifies them against the stored hashed passwords. Upon successful authentication, a JSON Web Token (JWT) is generated and returned to the client. This token is then included in the header of subsequent requests (as `X-Authorization` header) to authenticate the user. The backend validates the token on each request to ensure that it is still valid and has not been tampered with. If the token is close to expiration, the client can request a new token using a refresh token mechanism, allowing for seamless user experience without frequent logins.

3.5.4 Session Management

For session management, the platform uses SimpleJWT (JSON Web Token). This library generates signed and encrypted tokens for each authenticated session, allowing the application to verify user identity without storing session data on the server. By using stateless JWTs, the system can efficiently scale across multiple instances without centralized session tracking. Each

token is cryptographically signed, ensuring that it cannot be tampered with or forged, and it can be easily revoked or refreshed as needed for enhanced security and user control.

4 Implementation Plan

4.1 Frontend

The product uses a standalone fronted build with Svelte 5. Svelte is a modern JS framework that gained popularity in recent years. Most team members had prior experience with Svelte. The main reasons for choosing Svelte are its simplicity, and ease of use. It also has a big ecosystem and good documentation and automatic XSS protection.

4.2 Backend

The backend of the product is built with Django and Django Rest Framework (DRF). Django is a high-level Python web framework. Like Svelte, most team members had prior experience with Django. One of the features of Django that made it an attractive choice is the built-in admin panel that comes with it. This admin panel allows administrators to easily manage users, projects and tasks. Django and DRF also provide a s secure way to build RESTfull APIs. Django's ORM also makes it easy to interact with the database without writing raw SQL queries, which prevents SQL injection attacks.

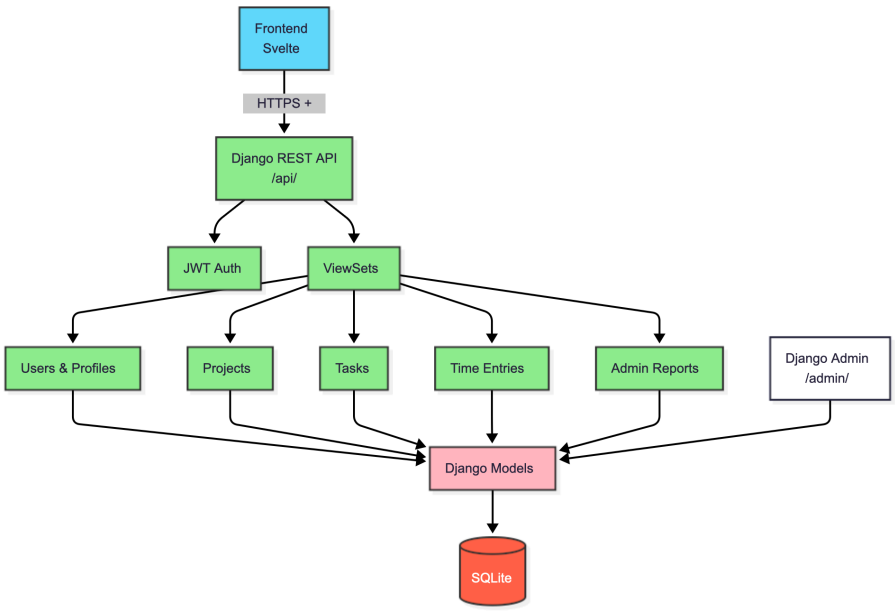

Figure 4: Backend Architecture

In Figure 4 you can see a high level overview of the backend architecture. The backend is a REST API that communicates with the frontend via HTTP requests. The backend uses JWT for authentication and authorization, it also has a built-in admin panel that allows administrators to manage users, projects and tasks. There is a separate viewset for each model. Each viewset handles the CRUD operations for its respective model. The backend also has custom endpoints for exporting data as CSV files. The backend uses SQLite as the database.

4.3 Packaging

One of the most pervasive issues in software is that of reproducibility, also known as the "it works on my machine" problem. Reproducibility issues often come up during development when

two developers have a different version of a required dependency. During our development phase we did not face any such issues because we used the Nix package manager to manage all of our dependencies. Nix was used to build our frontend, backend, docker image and even our report. For example, to build the docker image you can run `nix build .#backend-docker-image`. This same command is used in CI to build artifacts for all changes to the main branch and all merge requests.

4.4 Hosting

For a testing/development environment we used Jorik's homelab to host the docker image. Our client and supervisor could use this environment to test the latest version since the main branch would automatically be deployed when there are new changes.

Our client was not familiar with deploying a production web application. We do consider hosting to be part of a full solution so we added it as a requirement. There are a nearly infinite number of methods and providers to host a web application but we wanted to find one that was right for our client. We settled on some more specific requirements that the chosen provider should abide by:

1. Must be cheap
2. Should be compatible with other providers (migration to a different provider should not be a project of its own)
3. Must have periodic backups that are easy to restore by the client.

We needed to host three things which do not necessarily have to be with the same provider: a Django application, a frontend consisting of static files and a database. We figured that this application is not likely to serve hundreds or thousands of concurrent users or extremely large data sets, therefore a simple SQLite database should suffice and we do not need a separate database provider.

We considered hosting all 3 on a VPS from a provider like Vultr or DigitalOcean. This would cost about $6-7 dollars per month including snapshot backups. This fullfills the requirements of being cheap and having proper backups, but setting up the VPS and administrating it will require a significant amount of technical expertise.

Another solution would be fly.io, this provider scales your application down to 0 when it is not in use. For our application this would save a considerable amount of money since it is unlikely that anyone is using it at night. In the worst case this provider would cost $2.34 per month and in the best case as low as $1 per month. The main drawback would be that backups are unreasonabely expensive, we considered building a custom solution where we would periodically upload the database to some kind of cloud storage. The restore process would not be very user friendly if we decided to go this route.

In the end our supervisor decided that it is good enough to provide a docker image and that the production deployment is not our responsibility.

A working docker configuration should look something like the one shown in Listing 1, this uses Nix syntax but converting to docker-compose syntax should be trivial.

Note that this image only contains the backend, in order to host the frontend you should simply serve the static files using your preferred webserver. Listing 2 shows an example NixOS configuration for Caddy.

```
        {
          virtualisation.oci-containers.containers.hovyu-backend = {
            image = "hovyu-backend:latest";
            environment = {
              "DJANGO_BASEDIR" = "/data";
            };
            volumes = [
              "/home/main/data/hovyu-backend:/data"
            ];
            ports = [
              "127.0.0.1:8000:8000"
            ];
          };
        }
```

Listing 1: The NixOS configuration used in our test hosting environment to run the backend Docker image

```
{ pkgs, ... }:
{
  services.caddy = {
    enable = true;
    package = pkgs.caddy;
    virtualHosts = {
      "hovyu-backend.jorik-dev.com".extraConfig = ''
        root /srv/hovyu-static
        handle_path /static* {
          file_server
        }
        handle {
          reverse_proxy http://localhost:8000
        }
      '';
      "hovyu.jorik-dev.com".extraConfig = ''
        root * /srv/hovyu-frontend
        try_files {path} /index.html
        file_server
        encode gzip

        # Strict security headers for production
        header {
            Strict-Transport-Security "max-age=63072000; includeSubDomains;
preload"
            X-Content-Type-Options nosniff
            X-Frame-Options DENY
            X-XSS-Protection "1; mode=block"
            Referrer-Policy strict-origin-when-cross-origin
        }
      '';
    };
  };
}
```

Listing 2: The NixOS Caddy configuration used in our test hosting environment

5 Testing and Evaluation

5.1 Testing Plan and Methodology

The HOVYU Time Tracker utilizes the Django's built-in TestCase framework, which is the industry standard for isolated tests, on top of using the Django REST Framework's APIC client. This enabled us to test entire user flows by simulating responses from all API endpoints, testing the system as a whole through a multi-layered approach covering unit, integration, and system levels. Tests were executed using Django's manage.py test main.tests command, which creates a dummy database for each run to ensure test independence. In addition, we made it so all tests are inherited from a BaseTestCase class which streamlined creating user and client instances to simplify testing across these levels.

5.1.1 Unit Testing

At the unit level, we isolated individual components like model methods and calculated fields for independent verification. This helped us ensure core business rules such as time span calculations, expense computations and dynamic rate structure work properly at fundemental level. For example, in the CalculatedFieldTests class, test_project_total_estimated_hours directly calls the total_estimated_hours() method on a model instance to see if it returns the expected sum, without involving any clients, HTTP requests or serializers.

5.2 Integration Testing

At the integration level, as mentioned before, we used the Django REST Framework's APIClient to simulate HTTP requests and test the full request-response cycle. This helped us validate different components working together correctly such as URL routing, view logic, authentication, serializers, and database operations. For instance, in the TimeEntryViewSetTests class, the test_stop_timer method authenticates a user, creates a running timer entry in the database, and sends a POST request to the custom /api/time-entries/stop_timer/ endpoint. The test then checks whether the server returns a 200 OK status and confirms the integration by refreshing the entry from the database and verifying whether its end_time field was successfully updated.

5.3 Permission Testing

At the permission level, we made sure that our row-level security was extensively tested, as the project was aimed to be for enterprise use with critical data (working hours and pay calculations) and multiple users with different access levels. Tests were implemented to ensure correct operation of ownership-based access control and unauthorized attempts are properly rejected. For example, in the PermissionTests class, the test_user_cannot_see_others_projects function creates 2 projects owned by user1 and user2 separately, then authenticates itself as user1. When it sends a GET request to the /api/projects/ endpoint, it tries to assert whether the response list only contains the project under user1 to confirm unauthorized projects are properly filtered.

5.4 Validation Testing

At the validation level, we made tests purposefully create invalid data, such as missing fields and constraint violations, to ensure business rules were enforced properly. For example, in the ProjectDateRangeTests class, the test_entry_before_start_date_rejected method verifies if a user tries to POST a time entry with a start_time before the project's official start_time, the API rejects it properly, protecting data integrity.

5.5 Edge Case Testing

At the edge case level, we implemented tests for error scenarios we could think of and boundary conditions including stopping already-stopped timers, creating entries with negative durations, deleting non-existent records, and assigning non-existent users. For example, in the EdgeCaseTests class, the test_stop_already_stopped_timer function simulates a user trying to stop a timer that is already stopped. It creates a TimeEntry that already has an end_time field then sends a POST request to entries own /stop_timer/ endpoint. Then it confirms whether the server detects the invalid action and responds with an error message instead of crashing.

5.6 Coverage

Our entire testing process includes 66 comprehensive tests covering all major features. We achieved %100 critical path coverage for authentication, authorization, and data manipulation on top of the edge case validation.

5.7 Test Cases and Scenarios

5.7.1 Test Cases

Test Case	Objective
Authentication & Authorization	
Current User Retrieval	Authenticated users can retrieve their profile data
Unauthorized Access Rejection	Unauthenticated requests receive a HTTP 401 error
Project Management	
Listing Projects with Filtering	Users only see projects they are related to
Project Creation	Project creation correctly assigns the user as the owner
Project Retrieval	Project details can be fetched by its ID
Updating Project Details	PATCH operations for editing project details work correctly
Project Deletion	Deletion removes the project from the database correctly
Project Task Retrieval	Retrieving all tasks associated with a project is successful
Project User Assignment	Users can be assigned to or removed from projects with specific roles
Project Assignee Listing	All users assigned to a project can be retrieved
Task Management	
Accessible Task Listing	Tasks are filtered based on user access rights
Task Filtering by Project	Tasks can be filtered by the project ID
Task Filtering by Status	Status-based filtering (TODO/In Progress/Done) is correct

Task Creation	Tasks can be created connected to the related project
Task Status Update	Task status transitions instantly and system-wide
Task User Assignment	Users can be assigned to specific tasks
Time Entry Management	
Time Entry Listing	Users can only see their own time entries
Time Entry Creation	Time entries can be created specificed start and end time
Running Timer Validation	Users can't run multiple timers at the same time
Time Entry Update/Deletion	Modification and deletion with authorization checks work correctly
Timer Stopping	Running timers can be stopped
Running Timer Retrieval	All currently active timers for a user can be retrieved
Time Entry Filtering	
Filtering by Task ID	Time entries can be filtered to a specific task
Filtering by Project ID	Time entries can be filtered to a specific project
Filtering by Date Range	Time entries can be filtered by date range
Filtering by Timer Status	Time entries can be filtered by timer status
Permission & Access Control	
Project Access Control	Users see only their projects or assigned projects
Project Modification Prevention	Modifying others' projects returns a HTTP 404 error
Time Entry Access Control	Users only see time entries for projects they can access
Task Visibility Control	Assigned users can see the tasks of that project
Grace Period Validation	
No Set Cutoff Date Modification	Entires can be modified as long as there is no inovice date set already
Project Date Range Validation	
Date Range Rejection	Entries before the start date or after the end date return a HTTP 400 error
Date Range Allowance	Entries within the valid date range are allowed

No Set Start/End Date Handling	Entries can be modified without limits as long as their is no start or end date set yet
Assignment Management	
Project Assignment Listing	All user-project assignments can be retrieved in the same view
Task Assignment Listing	All user-task assignments can be retrieved in the same view
Edge Cases & Error Handling	
Stop Stopped Timer	Stopping an already-stopped timer returns a HTTP 400 error
Negative Duration Entry	Entries set with end time before start time are rejected
Delete Non-Existent Entry	Deleting a non-existent entry returns a HTTP 404 error
Assign Non-Existent User	Assignment of a non-existent user returns a HTTP 404 error
Assignment without User ID	Assignment without a user ID returns a HTTP 400 error
Unassign Non-Assigned User	Removing a user who is not assigned returns a HTTP 404 error
Invalid Hourly Rate	Hourly rate accepts only valid decimal values
Calculated Fields & Model Methods	
Project Hour Summary	Task estimates and time entries calculates correctly at the project level
Task Hour Summary	Total tracked time calculated correctly for each task
Duration Calculation	Time entry converts correctly from seconds to hours
Cost Calculation	Time entry cost calculates correctly using hourly rates
Hours Variance Calculation	Variance between actual and estimated hours calculates correctly
Data Validation	
Project without Name	Project creation without a name returns a HTTP 400 error
Task without Title	Task creation without a title is rejected
Invalid Time Entry	Time entries require task and start time to work
Invalid Task Status	Task status only accepts valid values (TODO/In Progress/Done)
Admin Reports	
Admin Access Enforcement	Unauthorized users receive a HTTP 403 error when try to access the admin control panel

Project Summary Report	Project summary reports generate correctly
System-Wide Summary Report	System-wide summary reports involve all projects
Missing Parameter Handling	Reports with missing parameters return a HTTP 400 error
Date-Filtered Reports	Reports can be filtered by a date range

5.7.2 Test Scenarios

Due to the nature of our project, there are too many combinations, thus scenarios possible. As it would be impractical to detail all of them, below are three scenarios that shows the validation of a few of the system's most critical business rules and user workflows. They are presented in "Given-When-Then" format which is the industry standard for behavior driven development methodology that we followed.

Scenario 1: Row-Level Security and Access Control	
Importance	This scenario tests one of the fundemental requirements in enterprise software, that users cannot interact with data that isn't related to them. It is critical for data integrity and user privacy.
Given	• "User A" and "User B" are both normal users • "User A" creates a project named "Project A" and logs X hours on a task • "User B" isn't assigned to "Project A"
When	• "User B" makes a request to list all projects or list all time entries
Then	• The system shouldn't include neither "Project A" or "User A"'s X hour time entry in the response • If "User B" attempts to fetch "Project A" directly by its ID, the system should return a HTTP 404 error

Scenario 2: Admin-Only Report Export	
Importance	This scenario confirms exporting data for payroll calculations (and analysis) by an admin, which is essential for the client. It also confirms the permission difference between admins and normal users.
Given	• "User A" is a normal user • "User B" is an admin • Database has already multiple time entries from various users and projects
When	• "User A" tries to access the endpoint for exporting working hours per project • "User B accesses the same endpoint
Then	• The system should reject the request of "User A" and return a HTTP 403 error • The system must correctly process the request of "User B" and return a HTTP 200 response with the complete data as a CSV file

Scenario 3: Running Timer Validation

Importance	This scenario verifies the core functionality, "start/stop" work timer, works correctly and the enforcement of no duplicate and overlapping time entries as a critical business rule to prevent data errors in the future.
Given	• "User A" is logged in and assigned to "Task A" • "User A" has already started a timer for "Task A" that is currently active
When	• "User A" attempts to start a new timer for another task.
Then	• The system should prevent the new timer from starting • The system should return a HTTP 400 error and notify "User A" that a timer is already running • The original timer for "Task A" should continue running without changes

5.8 Results and Observations

5.8.1 Results

The entire collection of 66 tests passes with a %100 success rate and only takes 3-5 seconds approximately for complete execution.

5.8.2 Key Findings

The JWT authentication and row-level security proved to be tough and universal throughout all testing phases. Tests confirm that users are correctly isolated from each other's data with no leakage observed. The validation engine is consistent and prevents invalid data like running timer conflicts and entries outside projects' date ranges. All calculated fields passes succesfully, including the complex dynamic billing rate logic that changes with many different parameters. Error handling was confirmed to be robust as well through edge case testing. The system correctly returns the appropriate HTTP error codes with descriptive messages to show users was was wrong. Also, it's evident that the tests had efficient database handling and design by the low 3-5 seconds execution time.

5.8.3 Known Limitations

The grace period logic with month-end boundaries were difficult to test extensively and probably need further development to handle possible edge cases properly. Also, project and task assignment serializers use their own custom endpoints instead of a streamlined direct API creation which adds unnecessary complexity and possible issues with integration with third party systems. Additionally, while our permission tests are comprehensive, they focus primarily on ownership based access control and probably not covers all possible combinations that might occur in such a big scale multi-user environment.

5.8.4 Future Work

While the tests cover core functionality well, there wasn't any extensive testing for third party support. Future development must focus on implementing the CSV/JSON export functionality in a more robust manner. Also better testing for bulk operations, such as creating or updating many entries simultaneously, is needed. Race conditions might occur, especially for concurrent timer operations. Some logic should be implemented to ensure atomic operations and prevent data corruption when multiple users attempt to modify the same thing at the same time. Finally, load testing will also be necessary after deployment as the system will be used in a complex environment that will have multiple access points and way bigger scale.

5.9 Bug Fixing and Iterations

We have met with our supervisor every week as planned. In each meeting, we had a discussion about the bugs he encountered, what new functionality he wants and other small design decisions. We noted down his feedback and created issues on GitLab for proper tracking. This helped us to focus on fixing the bugs and implementing prioritized features systematically as the project progressed.

6 Project Management

6.1 Tools for Collaboration (GitHub, Trello, etc.)

In order to collaborate effectively, we used the GitLab instance provided by University of Twente. Our requirements are documented as issues within the GitLab and we use the Git to share our code and review changes with merge requests. Pushing directly to main was disabled to ensure that changes would have to be reviewed before being released to our testing environment. GitLab CI/CD was used to automatically build and test new changes. Changes to the main branch were automatically deployed to our testing environment such that our client and supervisor can use them and validate them. We also used WhatsApp and Microsoft Teams for communication and online meetings.

6.2 Development Process (Agile / Waterfall / Hybrid)

Throughout the development stage, we have followed a hybrid agile process. We used Waterfall model at first to have a well structured backbone for the project, then switched to agile for faster development and flexibility while doing so.

6.2.1 Waterfall Practices

- Started the project with a structured requirements gathering phase and made sure to clearly define our scope, expected timeline and other minor details.
- A complete database schema and architecture design was agreed upon before starting any real development.

6.2.2 Agile Practices

- Features were added weekly after each sprint.
- Had regular meeting with our stakeholder to get feedback after each iteration and clarify priorities for the next iteration.
- Main branch was set to auto deployment to enable the stakeholder to test new features instantly
- Tests were written while features being implemented to prevent regression.

6.3 Roles and Responsibilities

- Oliver: Frontend, Backend, Report
- Mohamed: Backend, Testing, Report
- Jorik: Frontend, Backend, Devops, Report
- Arseniy: Testing, Presentation Slides, Report
- Naut: Backend, Report
- Eren: Frontend, Report

7 Conclusion

The HOVYU Time Tracking Platform was developed to provide a practical, secure, and efficient solution for managing employee working hours within a growing organization.

Throughout the project, communication with the client played a key role in shaping the final product. Our contact person was a HOVYU employee who had a clear vision of the company's operational challenges but was not highly technical, which meant that many of our early meetings focused on translating abstract needs into actual system features. We learned to ask the right questions, simplify technical language, and iteratively refine our ideas into features that also made sense from a business perspective.

Through careful requirement analysis, we identified the key needs of the company and translated them into a functional and reliable web application. Our design choices such as separating the frontend and backend, implementing JWT-based authentication, and enforcing role-based permissions ensured that the platform remains secure, maintainable, and adaptable to future needs. The combination of Svelte and Django allowed us to achieve a balance between performance, developer efficiency, and long-term scalability. We also paid special attention to keeping data secure and reliable. Django's ORM helps prevent SQL injection, Svelte automatically escapes user input to stop cross-site scripting, and authentication is handled safely using JWT tokens with encrypted passwords. We also used Docker and Nix to make sure the project could be built and deployed consistently on any machine. These choices helped us avoid technical issues and focus on improving the user experience.

By the end of the project, we met our main goals: the system allows users to record and edit their hours, and admins can easily view, manage, and export that data. The platform makes time tracking more organized, reduces mistakes, and saves time for everyone involved.