BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION Computer Science

DIPLOMA THESIS

CelestiMap: Interactive 3D Star Mapping

Supervisor
Conf. STERCA Adrian-Ioan, PhD

Author
Carp Nicoleta

2025

ABSTRACT

CelestiMap is a web application that offers an engaging 3D star mapping experience with immersive interactions. It caters to a broad spectrum of users, from those with interest in astronomy to educators, students, and maybe even researchers. Its main goal is to provide engaging and modern tools for a deeper exploration and understanding of stars and their mapping. It encourages creativity and offers additional knowledge for curious minds through other incredible tools.

The front-end was developed using React while its back-end consists of Spring Boot and MySQL servers. For the 3D star mapping, the application integrates Three.js via the react-three-fiber renderer. Users can manipulate camera angles and interact with celestial bodies and draw constellations in real-time. Other functionalities include secure user authentication with JWT, an API for RESTful interaction with projects, guest profile support via localStorage, and data import/export capabilities for JSON and CSV formats. Users can access a publicly available project and are allowed to bookmark their favorites. CelestiMap also retrieves NASA's Astronomy Picture of the Day (APOD) via the public API and displays it in an in-app, with an optional feature to send the daily APOD to users via an email. In addition, the application incorporates real NASA near-Earth asteroid data (from the NeoWs service) into an interactive 3D visualization.

CelestiMap is peculiar in its educational delivery, hobbyist exploration, and professional project management, all within one web-based 3D interface. It distinguishes itself with easy, real-time interactivity, social sharing, and data manipulation, all done without the need for specialized equipment or software installation. This project demonstrates the potential impact of new Web technologies on the exploration, sharing, and understanding of celestial data, particularly in education and collaborative research.

Contents

Chapter 1

Introduction

CelestiMap provides a full-stack, web-based 3D star-mapping environment. It uses React [Met25] for the front-end, Spring Boot for the backend, and MySQL for the database. Users can navigate freely with a camera and access detailed information panels for celestial objects. All these tools and some more offer the possibility for educators, hobbyists, and researchers to share and remix their own star-map projects.

1.1 Context of the thesis

WebGL, based on OpenGL ES2.0, has become a standard in modern browsers, enabling hardware-accelerated 3D graphics without external plugins [Gro21]. Libraries like Three.js build on WebGL by providing reusable components (lights, cameras, materials, loaders), dramatically simplifying the creation of immersive web experiences.

While many online astronomy tools excel at data visualization and annotation, few combine these capabilities with social sharing and project management. CelestiMap fills that gap by offering a unified, browser-based platform for creating, collaborating on, and publishing interactive 3D star-map projects.

1.2 Motivation

Many astronomers and educators still rely on desktop planetarium software, spreadsheets, and separate publishing tools [DHG+18], resulting in fragmented workflows. They often wish for a unified, web-accessible interface.

CelestiMap fills that gap by combining browser-based 3D star visualization (React + Three.js), JWT-secured project management (Spring Boot + MySQL), and localStorage fallback. It streamlines creation, sharing, and learning in a single platform, fostering an online community of astronomy enthusiasts.

1.3 Objectives

This thesis sets out to:

- Design and implement a full-stack web application that integrates 3D star-map visualization, user authentication, project management, and social sharing in a single platform.
- Create an interactive 3D environmentbased on Three.js focusing on discoverability, annotation, and navigation.
- Provide JSON and CSV import/export strategies, enabling easy exchange with external tools and support both offline (guest) and online (authenticated) workflows.
- Offer community engagement, through the public gallery and allowing to favorite public projects.
- Integrate the NASA Astronomy Picture of the Day (APOD), enabling all users to fetch and view the daily cosmic image within the application.
- Authenticated users may choose to receive a scheduled daily email at 10 AM local time with the APOD content
- Incorporate a Near-Earth Object (NEO) visualization module, which retrieves upcoming asteroid approach data from NASA's public feed and presents these bodies in a 3D environment.

These objectives will guide the development process of CelestiMap, while discussing its contributions to web-based astronomical visualization and education

1.4 Generative AI and AI-assisted technologies

During the preparation of this work I used ChatGPT in order to support and enhance the quality of my writing. After using this tool/service, I reviewed and edited the content as needed and takes full responsibility for the content of the thesis.

Chapter 2

Related Applications

CelestiMap is a browser-based 3D star-mapping platform designed to serve everyone from amateur stargazers to classroom instructors and professional researchers. There are however other well established desktop and web-based planetarium applications like Stellarium, WorldWide Telescope, or mobile sky-mapping apps.

2.1 Stellarium Web

Core Functionality and Audience

Stellarium is a mature, open-source planetarium originally written in C++/Qt, providing accurate night-sky simulations for any time and location. It supports hundreds of millions of stars (e.g., Gaia catalog) and deep-sky objects, targeting amateur astronomers, educators, and planetarium operators [Con25, Dev25].

Technical Architecture

The desktop Stellarium uses OpenGL on Windows/Linux/macOS [Dev25]. The Stellarium Web Engine compiles its C++ core to WebAssembly via Emscripten, rendering via WebGL with a minimal JavaScript UI [Con25, Gro18]. CelestiMap, by contrast, is built entirely in JavaScript (React + Three.js) with a Spring Boot back end exposing REST APIs. Neither WebAssembly nor legacy C++ code is used.

Feature Comparison

Both applications enable interactive 3D navigation and constellation rendering. Stellarium offers comprehensive features—time controls, planetary orbits, multi-culture constellation art, realistic atmosphere, satellite tracking, and bundled catalogs of $\sim$220M stars and $\sim$1M deep-sky objects [Fre25]. CelestiMap uses a smaller star set

optimized for user maps but uniquely integrates live NASA feeds (APOD and NeoWs asteroids) and social functions (public gallery, favorites). Stellarium allows user catalogs and plugins but lacks CelestiMap's account-based project workflow and import/export capabilities.

Strengths and Limitations

Stellarium excels in data richness and simulation fidelity but requires installation or hosting assets for web use and offers no built-in sharing. CelestiMap prioritizes accessibility (zero install), modern UI, and collaborative features, though it does not match Stellarium's extensive catalogs or advanced time-domain controls [Gro17].

2.2 Sky Map (Mobile Handheld App)

Core Functionality and Audience.

Sky Map is an open-source Android app that overlays labels for stars, planets, and constellations on your phone's camera view, turning it into a handheld planetarium for casual stargazers [WfG25].

Technical Architecture.

Sky Map is a mobile app (originally by Google, now community-maintained) built in Java/Kotlin for Android (and iOS). It uses onboard sensors (GPS, compass, gyroscope) to orient a 2D star chart—either as an all-sky projection or over the camera view—entirely offline, with catalogs stored on the device [WfG25].

Feature Overlap and Differences.

Sky Map also lets you search for stars, constellations, and deep-sky objects by name, but it's a 2D/AR app tied to your phone's horizon and sensors—it shows only what's overhead, with no free 3D rotation or zoom. There's no project saving or data imports (no APOD or NeoWs feeds). In contrast to CelestiMap's custom 3D map creation and sharing, Sky Map is built solely for on-the-spot sky identification [WfG25].

Strengths and Limitations.

Sky Map excels in portability—offline, real-time star ID on your phone with minimal learning curve—but it lacks project saving, import/export, networked data, and uses simple 2D overlays. CelestiMap, by contrast, delivers a full 3D editor with

custom projects and live data feeds in the browser, though it can't serve as an AR field guide.

2.3 Conclusion

CelestiMap distinguishes itself by merging an interactive 3D star map with modern web accessibility and community features. In contrast to heavyweight desktop simulators like Stellarium, it requires no installation and offers an intuitive React-based interface. Unlike simple sky-viewers like Sky Map, it provides free 3D navigation and fully user-generated content. By integrating real NASA data (APOD images and NeoWs asteroid trajectories) and hosting a gallery of shareable projects, CelestiMap serves both educational and collaborative use cases. Its core innovation is leveraging contemporary web technologies (Three.js/react) to deliver an immersive, customizable, and community-oriented astronomy platform—bridging the gap between professional planetarium software and casual stargazing apps.

Chapter 3

System Requirements and Design

3.1 System Requirements and Specifications

This chapter outlines the functional and non-functional requirements, architectural structure, and system design specifications for CelestiMap, created using React, Spring Boot, and Three.js. The platform aims to offer interactive navigation through a virtual universe, coupled with user customization and dynamic star data handling capabilities.

3.1.1 Functional Requirements

User Authentication

The system provides secure user registration and login, issuing JWT tokens upon successful authentication. The front-end includes a `Register` page and `Login` page where users submit credentials. The backend uses Spring Security and a custom `AuthController` to register users and generate tokens. Logged-in users have their token stored and included in subsequent API requests via an `Authorization: Bearer "token"` header for authenticated actions.

Project Management

Authenticated users can create new star-map projects, edit them, and delete them. The `ProjectEditor` component provides a form and 3D editor for building maps. When the user saves a project, the frontend sends a JSON payload and the backend `ProjectController` handles `/project/save` and returns the new or updated project ID. Deletion is supported via the endpoint DELETE, which removes a project owned by the user. These operations are protected by JWT (the user's ID is validated against the project owner), through the use of `@AuthenticationPrincipal` in the controller methods.

3D Interactive Mapping

The core feature is a 3D star map editor using Three.js via React. The front-end renders a `Canvas` from react-three-fiber with ambient and point lights, a star field background, and dynamic star and line objects. Each star is a sphere at coordinates (x,y,z), and each connection is drawn as a line between stars. Users can click buttons in the editor UI to add or edit stars, which updates React state and re-renders the scene in real time.

Public Gallery

A `Gallery` page displays all projects marked public. It calls the backend supplying optional parameters for search keywords, sort order, page number, and page size. The response is a paginated list of projects, which the front end shows in a grid with controls for searching (by name or creator) and sorting (by date or name)

Favorites System

Authenticated users can mark public projects as favorites. In the Gallery clicking a "favorite" button triggers `POST /favorite/projectId`. The backend adds a record linking the user and project. Users can also view or delete their favorites on a `Favorites` page. These actions require a valid JWT and return data on the user's favorite projects or success/failure status.

Guest vs. Authenticated Users

The application's behavior changes based on login state. The Navbar component checks if a user is present in context and conditionally renders buttons. Some pages (like `ProjectEditor` and `MyProjects`) require authentication and redirect to login if visited by a guest. The backend also enforces this by using `@AuthenticationPrincipal`, so unauthenticated requests are denied.

Import/Export (JSON/CSV)

Users can import or export project data for portability. The front-end provides buttons to download the current project as JSON or CSV, or to upload a file to load a project. The export endpoint accepts a `CompleteProjectDTO` in the request body and returns a file (with appropriate Content-Disposition). The import endpoint accepts a multipart file upload and returns a parsed `CompleteProjectDTO`. This allows users to save their map as a file on their computers or bring in data created elsewhere.

APOD Integration

Each day, the latest Astronomy Picture of the Day is retrieved from NASA, [NAS25]. Additionally, users can subscribe to receive the daily APOD automatically via email at 10AM.

Email Notifications

The back-end has implemented a scheduled task that, every 24 hours, fetches the day's APOD and sends an email to subscribed users. The email is HTML-formatted and includes the image, title, and explanation. Users also have the option of unsubscribing.

NeoWs Asteroid Visualization

Near-Earth objects are shown in a 3D view with animation of motion. The system fetches NEO data from NASA's NeoWs API, [NAS21]. For each relevant asteroid, the 3D scene renders a sphere scaled in size and animates its position to simulate orbit motion. The interface includes controls to play/pause the animation and a slider to go through timea dn speed.

3.1.2 Non-Functional Requirements

Security

The system uses Spring Security and JWT to protect all sensitive operations. Endpoints under `/api/v1/project/**` and `/api/v1/favorite/**` require authentication. A custom `AuthTokenFilter` validates JWTs on incoming requests (rejecting invalid or missing tokens) before invoking controllers. Passwords are encoded and decoded server-side, and token generation is handled securely in the `AuthController`. Lastly, the NASA API key is stored securely as an environment variable.

Performance

The application must render 3D graphics smoothly. By using react-three-fiber (a React renderer for Three.js), the front-end takes advantage of WebGL hardware acceleration in modern browsers. The star field and objects are drawn by the GPU for real-time interactivity. Data operations like fetching or saving data use paginated or asynchronous calls to avoid blocking the UI.

Compatibility

The UI is built with standard web technologies (React, HTML/CSS, WebGL via Canvas), ensuring it runs on all major desktop browsers (Chrome, Firefox, Edge, etc.). The app has been tested on recent versions of these browsers to confirm the 3D canvas and UI controls work consistently.

Usability

The interface prioritizes clarity and feedback. Forms have client-side validation (e.g. password length, required fields) and error messages. When actions succeed or fail (like saving a project or adding a favorite), the app shows toast notifications or alerts so the user is informed in real time. The layout groups related controls (star listing, connection management) in a sidebar, and uses clear labels. The APOD email uses a HTML design so that images and text display properly on various devices. Overall, the UI aims to be intuitive: users see immediate visual feedback in the 3D view as they add stars or move the camera.

3.2 System Design

3.2.1 Architecture

CelestiMap is built on a three-tier architecture, which cleanly separates the presentation, application, and data layers. This modular design ensures scalability, maintainability, and clear communication between components while allowing each layer to evolve independently.

Presentation Tier: React

The front-end is implemented as a `Single-Page Application (SPA)` using React [Met25], enabling dynamic user interactions without requiring full page reloads[RM20]. The UI is structured around reusable components such as `Scene`, `Sidebar`, and `StarInfoPanel`, which are managed using React hooks (`useState`, `useContext`) for both local and global state. For instance, the `AuthContext` handles user session management, while `react-router-dom` facilitates seamless navigation between views [Sho25].

A critical aspect of the front-end is its 3D visualization capabilities, powered by `@react-three/fiber` and `@react-three/drei`. These libraries enable interactive rendering of stars and constellations, along with intuitive camera controls. Communication with the back-end is handled via Axios services in `API.js`.

Application Tier: Spring Boot / REST API

The backend is a Java/Spring Boot[JBS15] service that exposes RESTful endpoints, structured into three primary layers:

- **Controllers** handle HTTP requests, validate inputs, and delegate business logic to services. For instance, a `@PostMapping("/save")` endpoint processes project-saving requests by first verifying user permissions before proceeding.
- **Services** contain core application logic, such as project validation, permission checks, and data transfer object (DTO) conversions. These services interact with repositories to persist or retrieve data.
- **Repositories** extend Spring Data JPA, providing high-level abstractions for database operations [Red19]. Custom queries, such as searching for public projects, are defined here.

Security is enforced via JWT authentication, where the `AuthTokenFilter` validates incoming tokens, and `SecurityConfig` defines access control rules. The backend remains stateless, relying on tokens to identify users without server-side session storage.

Data Tier: MySQL Database

The database stores relational data, with JPA entities mapping directly to MySQL tables [Ora25]. The schema includes key entities such as `User`, `Project`, `Star`, `ConstellationLine`, `UserNotification` and `Favorite`, each with well-defined relationships, Table. 3.1.

Entity	Relationships	Key Fields
User	One-to-Many with **Project** One-to-Many with **Favorite**	`userId`, `userName`, `password`, `email`
Project	Many-to-One with **User** One-to-Many with **Star**	`projectId`, `name`, `isPublic`
Star	Many-to-One with **Project**	`id`, `x`, `y`, `z`, `color`
ConstellationLine	Many-to-One with **Project** Many-to-One with **Star** (startStarId, endStarId)	`id`, `startStarId`, `endStarId`
Favorite	Many-to-One with **User** Many-to-One with **Project**	`favoriteId`
UserNotification	One-to-One with **User**	`id`, `user`, `subscribedAt`

Table 3.1: Tables Relations

The data flow between layers follows a structured sequence: the front-end sends an HTTP request (e.g., `POST /project/save`), which the back-end validates, processes, and persists via the service and repository layers. The database confirms successful storage, and the response propagates back to the front-end, completing the cycle, Fig. 3.1.

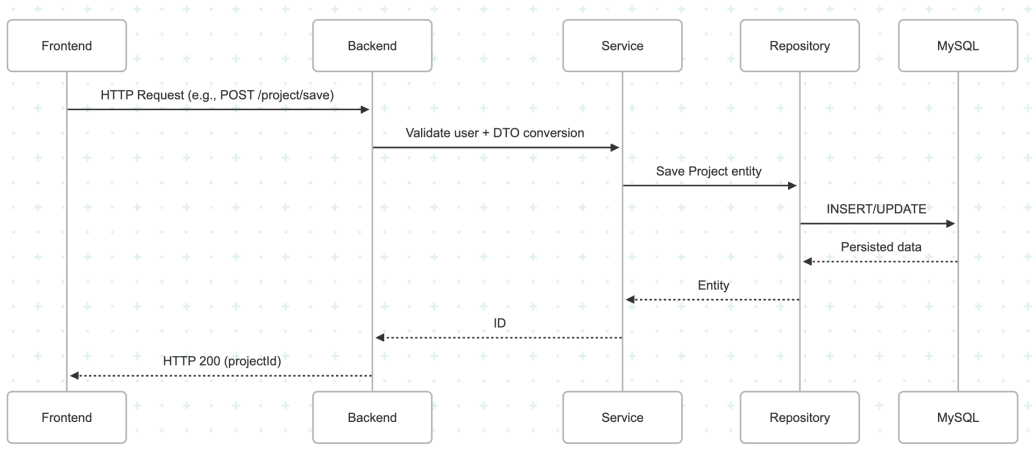

Figure 3.1: Flow Chart of Program

Extended Services for APOD and NEO Visualization

In addition to the core React-Spring-MySQL layers described above, CelestiMap includes two other service tiers in the application layer: an `APODService` (with its companion `APODController` and a scheduled task for daily email delivery), and a `NeoWsService` (with `AsteroidController`) that fetches near-Earth object data and exposes it via REST.

The `APODService` encapsulates logic to call NASA's APOD API, map the JSON into domain objects, and hand off to the existing `EmailService` for HTML-formatted notifications, fig. 3.2. The `NeoWsService` similarly fetches NASA's NeoWs feed, transforms the approach-data into DTOs, and makes them available to the front-end, fig. 3.3. Both services are parameterized via externalized API keys and scheduling settings.

3.2.2 Database Schema

Key Design Principles

The CelestiMap database schema, Taable. 3.1, is designed with normalization and referential integrity as foundational principles. To minimize data redundancy and maintain logical consistency, the schema adheres to third normal form (3NF). Each

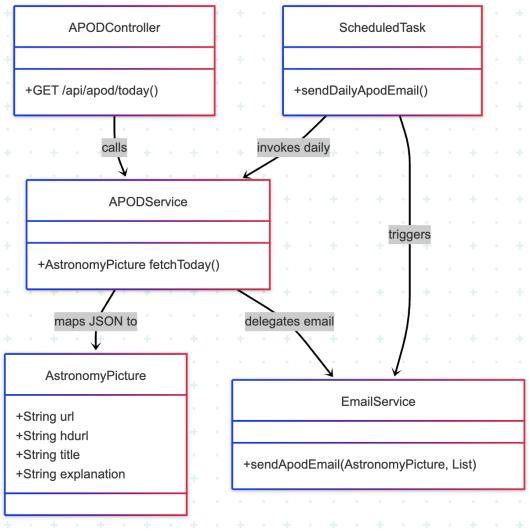

Figure 3.2: APOD Email Flow

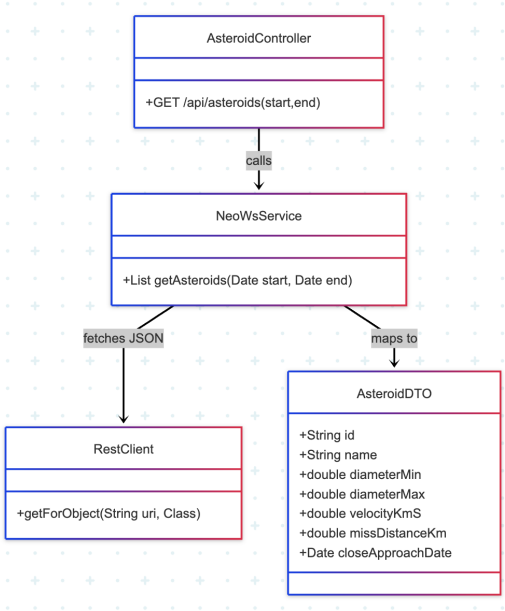

Figure 3.3: NeoWs Asteroid Visualization Flow

table is carefully structured to hold atomic values, ensuring that no columns contain multiple pieces of information. For example, the `Star` entity stores its 3D coordinates (`x`, `y`, `z`) directly, while connections between stars—such as constellation lines—refer to these stars via foreign keys rather than duplicating coordinate data.

Referential Integrity

Referential integrity is enforced using indexed foreign keys that maintain stable relationships across entities. For instance, each `Star` record includes a `projectId` that references the parent `Project`, and this relationship is protected by foreign key constraints.

To handle cascading behaviors effectively, the schema uses JPA features such as `CascadeType.ALL`, ensuring that deleting a `Project` automatically removes all associated `Star` and `ConstellationLine` records. Furthermore, uniqueness constraints are implemented where necessary, such as ensuring that `userName` in the `User` table is unique, preventing duplicate accounts and simplifying authentication.

Indexing Strategies

Indexing strategies play a crucial role in optimizing query performance. Unique indexes—like those on `userName` enable fast search during login and account validation.

Composite indexes are also employed for more complex filtering tasks, such

as browsing public projects by name (`isPublic + name`), supporting efficient gallery functionality. Additionally, clustered indexing on auto-incrementing primary keys (`projectId`, `starId`, etc.) facilitates fast range scans and ensures high performance in data retrieval operations.

Schema Optimization Highlights

Several optimization strategies have been implemented to enhance performance, scalability, and maintainability. Lazy fetching is applied to non-critical relationships, such as the association between `Project` and its owning `User`, using `FetchType.LAZY` to delay loading and reduce query overhead in read-intensive workflows. This improves response times for endpoints where full user details are unnecessary.

To handle potentially large volumes of textual content, the `additionalInfo` field in the `Star` entity uses the `TEXT` column definition. This choice allows efficient storage and retrieval of star annotations without compromising schema simplicity or performance.

Additionally, the use of auto-incrementing integer primary keys across all entities ensures consistent identity management and efficient indexing. These immutable identifiers help preserve referential integrity while also supporting high-throughput operations such as bulk inserts and range-based queries.

Together, these optimizations support the full range of application features from rendering real-time 3D visualizations to powering social functionalities like favorites and public galleries, while maintaining full ACID compliance through the use of Spring Data JPA and MySQL.

3.2.3 API Design

The CelestiMap API follows RESTful principles with stateless communication, leveraging HTTP methods and resource-oriented endpoints. Authentication uses JSON Web Tokens (JWT) for secure stateless sessions, while error handling employs standardized HTTP codes and structured responses.

RESTful Design Principles

The API is designed around RESTful principles, with each endpoint corresponding to a distinct resource and HTTP method combinations defining the intended operations [Fie00]. For example, the `/project` resource is manipulated via a `POST` request to `/project/save` for creation, a `PUT` request to `/project/update` for modification, and a `DELETE` request to `/project/delete/id` for removal. Similarly, user favorites are managed under the `/favorite` resource with comparable

method mappings.

To maintain statelessness, every request must carry all necessary context in its headers—specifically, a JSON Web Token (JWT). By embedding authentication and authorization data within each JWT, the server can process requests independently, without relying on session storage or retaining client state between interactions.

Finally, the API embraces HATEOAS (Hypermedia as the Engine of Application State)[Kel23] to facilitate client navigation through paginated collections. When returning, for instance, a gallery of items, responses include metadata such as `totalPages` and the current `page` number. This information allows clients to discover available actions (like fetching the next or previous page) dynamically, simply by inspecting the response payload rather than relying on out-of-band documentation.

Authentication and Authorization

The authentication mechanism is based on JSON Web Tokens (JWT) [JBS15], which are issued to clients upon successful login and used to secure subsequent requests. When a user attempts to log in, they send their credentials via a `POST /auth/login request`. The backend validates these credentials and, if successful, generates a JWT that encapsulates the user's identity and any associated roles.

Once issued, JWTs enable stateless validation of each request. An AuthTokenFilter intercepts incoming HTTP requests and extracts the token from the Authorization header. It then verifies the token's signature and validity period. If the token is valid, the filter retrieves the claims (including userId and roles) and populates the SecurityContextHolder with an authenticated principal. This approach eliminates the need for server-side session storage, as all necessary identity information is contained within the signed token itself.

To enforce role-based access control, the application performs ownership and role checks at the business logic layer, Table. 3.2. This ensures that sensitive operations are restricted to the resource owner or users with appropriate roles, preventing unauthorized access or modifications.

Endpoint	**Access**
POST /project/save	Authenticated users only
GET /project/public	Public (no auth)
DELETE /favorite/{id}	Owner of the favorite entry

Table 3.2: Authorization Rules

Chapter 4

Application implementation

4.1 Authentication: JWT Management and Spring Security Filters

Authentication in CelestiMap is based on JSON Web Tokens (JWT), enabling a stateless, scalable approach. When a user registers or logs in, the server issues a signed JWT; on subsequent requests, a custom filter inspects incoming tokens, validates them, and populates the security context. The following subsections, Fig. 4.1, describe the main components: how tokens are generated, how they are validated, and how Spring Security is configured to integrate the filter into the request pipeline.

4.1.1 Generating JWT

When a new user registers or an existing user logs in, the AuthController handles the request. In registration, `POST /api/v1/auth/register`, after creating the user via UserService, the controller loads user details and generates a token. In login `POST /api/v1/auth/login`, the controller authenticates credentials via AuthenticationManager, then generates a token upon success. The core token creation happens in JWTUtils.generateToken(UserDetails), Fig. 4.2. A map of claims is generated, the user's internal ID is added as custom claim. A token is then built with subject equal to the username, issued-at timestamp, expiration (10 days ahead), and signed with a symmetric key (HS512) generated at startup.

This mechanism ensures that after registration or login, the client receives a JWT containing their unique username in its subject and userId as a custom claim, signed by the server's secret key. The token's expiration enforces that clients must refresh credentials, like re-login, after a period, supporting security best practices.

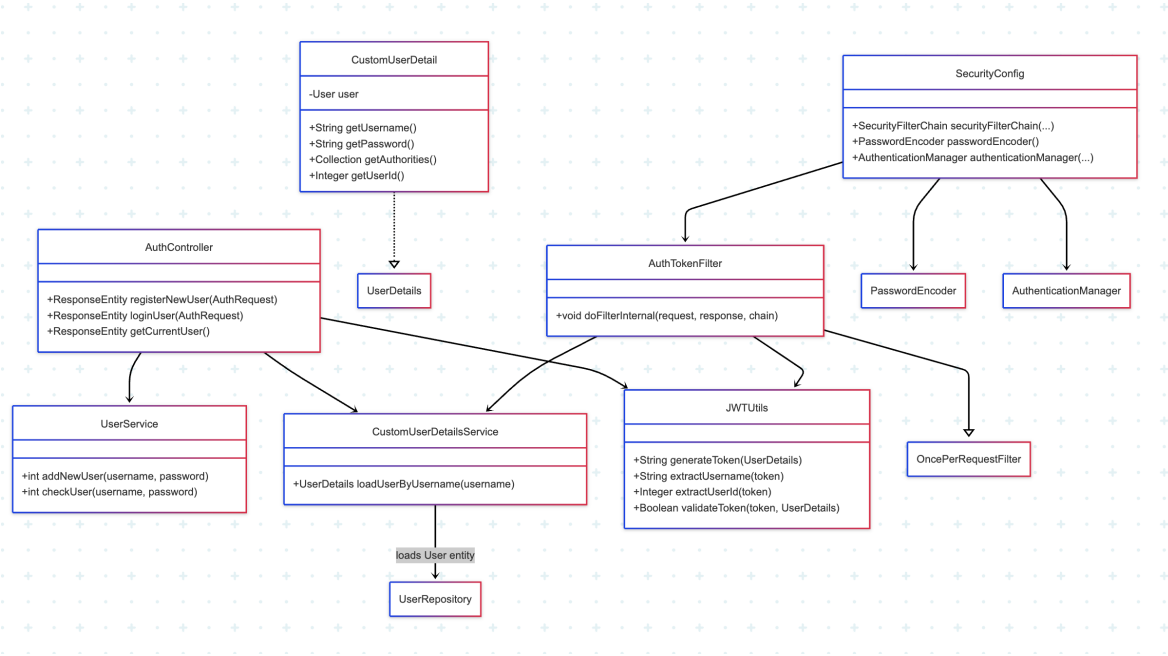

Figure 4.1: Security UML

```java
public String generateToken(UserDetails userDetails) {
    Map<String, Object> claims = new HashMap<>();
    if (userDetails instanceof CustomUserDetail) {
        claims.put("userId", ((CustomUserDetail) userDetails).getUserId());
    }
    return createToken(claims, userDetails.getUsername());
}

private String createToken(Map<String, Object> claims, String subject) {
    return Jwts.builder()
            .setClaims(claims)
            .setSubject(subject)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + EXP_TIME))
            .signWith(SECRET)
            .compact();
}
```

Figure 4.2: Code for generating JWT

4.1.2 JWT Validation

For every request to protected resources, the backend must verify that the provided JWT is valid (signature correct, not expired) and corresponds to an existing user. This is handled by a custom filter `AuthTokenFilter` which extends `OncePerRequestFilter`. The filter intercepts each HTTP request before it reaches controller endpoints:

1. It inspects the `Authorization` header. If present and starting with 'Bearer', it extracts the token substring.
2. It calls `jwtUtil.extractUsername(jwt)` to retrieve the subject (username in our case) from the token.
3. If the username is non-null and there is no existing authentication in `SecurityContextHolder`, it loads `UserDetails` with `UserDetailsService`.
4. It invokes `jwtUtil.validateToken(jwt, userDetails)`. This checks that the username in the token matches the loaded `userDetails`' username and that the token is not expired.
5. If successful, it creates a `UsernamePasswordAuthenticationToken` with the `userDetails` and sets it into the security context. Additionally, it extracts the `userId` claim from the token and attaches it to the request attributes, so downstream code (e.g., controllers/services) can retrieve the current user's ID without another search.

In JWTUtils, the following methods are availabie: `extractUsername`, `extractUserId`, `extractExpiration` and `validateToken` and they rely on parsing the token with the signing key. Validation, fig. 4.3, confirms the token has not been tampered with and is within its validity period.

4.1.3 Spring Security Configuration and Filter Integration

Spring Security is configured via a `SecurityConfig` class annotated with `@Configuration` and `@EnableWebSecurity`. The central component of this configuration is a `SecurityFilterChain` bean, fig. 4.4, which defines the security behavior for HTTP requests.

Everything stars with CSRF protection being explicitly disabled. This is because JWT (JSON Web Token) authentication eliminates the need for server-side sessions, rendering CSRF protection unnecessary in this context. Next, CORS (Cross-Origin

```
if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
    jwt = authorizationHeader.substring(7);
    username = jwtUtil.extractUsername(jwt);
}

if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
    UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

    if (jwtUtil.validateToken(jwt, userDetails)) {
        UsernamePasswordAuthenticationToken authenticationToken =
                new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.get
        authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(r
        SecurityContextHolder.getContext().setAuthentication(authenticationToken);

        // Attach userId claim for later use
        Integer userId = jwtUtil.extractUserId(jwt);
        request.setAttribute("userId", userId);
    }
}
```

Figure 4.3: jwtValidation

```
http
    .cors().and()
    .csrf(AbstractHttpConfigurer::disable)
    .authorizeHttpRequests(auth -> auth
        .requestMatchers(HttpMethod.GET, "/api/v1/project/public").permitAll()
        .requestMatchers(HttpMethod.GET, "/api/v1/project/view/**").permitAll()
        .requestMatchers("/api/v1/auth/**").permitAll()
        .requestMatchers(HttpMethod.POST, "/api/v1/project/import").permitAll()
        .requestMatchers(HttpMethod.POST, "/api/v1/project/export").permitAll()
        .anyRequest().authenticated()
    )
    .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ST

http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class
```

Figure 4.4: Security Filter Chain

```java
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowedOrigins(List.of("http://localhost:5173"));
    config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
    config.setAllowedHeaders(List.of("*"));
    config.setAllowCredentials(true);

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", config);
    return source;
}
```

Figure 4.5: CORS Configuration

Resource Sharing) is configured, fig. 4.5 to permit requests from specified front-end origins, such as `http://localhost:5173`. This ensures that the browser can successfully communicate with the backend despite cross-origin restrictions.

The configuration then establishes authorization rules. It permits unrestricted access to endpoints under `/api/v1/auth/**`, as well as to endpoints responsible for fetching public projects and handling import/export operations. All other endpoints require the user to be authenticated.

Furthermore, the application enforces stateless sessions by setting the session creation policy to `SessionCreationPolicy.STATELESS`. This aligns with the stateless nature of JWT authentication, where the server does not maintain any user session state.

Finally, the `AuthTokenFilter` is registered in the filter chain before Spring's `UsernamePasswordAuthenticationFilter`. This ensures that JWT validation is performed early in the request processing pipeline, allowing authenticated context to be correctly established before reaching protected endpoints.

By injecting the `AuthTokenFilter`, Spring Security ensures that token extraction and validation occur for every request, except those explicitly permitted by the security configuration. Additionally, the `PasswordEncoder` bean, configured with `BCryptPasswordEncoder`, secures password storage and verification within the `UserService`, providing a strong cryptographic mechanism for handling user credentials.

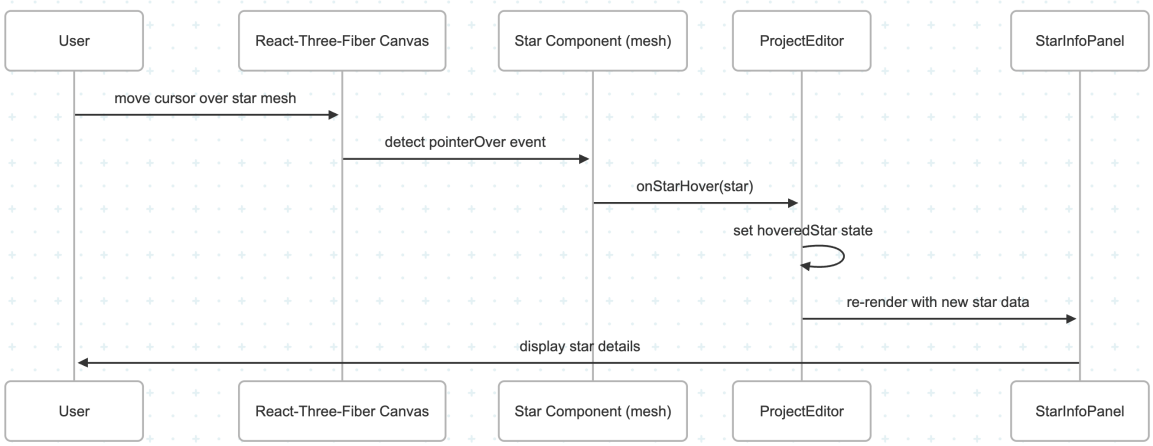

Figure 4.6: UI Flow Chart

4.2 3D Visualization: Star and Constellation Rendering with Three.js

In this section, I examine how CelestiMap realizes its immersive 3D star-mapping interface by leveraging react-three-fiber (a React renderer for three.js) together with utilities from drei. I begin by discussing the conceptual underpinnings of WebGL-based 3D rendering in a browser context and then describe how the React component hierarchy is structured to represent stars, labels, and constellation lines.

4.2.1 Foundations of Browser-based 3D Rendering

WebGL is a low-level API that allows hardware-accelerated 3D graphics in modern browsers. Three.js simplifies much of WebGL's setup by providing scene graphs, camera tools, geometry, materials, and lighting models [FH20]. React-three-fiber builds on Three.js by linking the scene graph to React's component model: components match Three.js objects, and React's reconciliation keeps the scene aligned with application state [Poi25b]. Drei is a helper library that makes common tasks easier by offering ready-to-use components like camera controls, background stars, and HTML overlays [Poi25a].

In CelestiMap, the main job of the 3D engine is to display a set of stars, each with spatial coordinates (x, y, z), color, name, and optional extra metadata, as well as the connections between them, known as constellation lines [BPM15]. The React application holds arrays of star objects and connection pairs. During each render, these arrays go into react-three-fiber's Canvas, creating a live 3D scene. OrbitControls from drei allows for free camera movement, including zooming, rotating, and panning, so users can explore the virtual star field.

```jsx
export default function Scene({ stars, connections, onStarHover }) {
    return (
        <Canvas camera={{ position: [0, 0, 10], fov: 60 }} shadows>
            <ambientLight intensity={0.2} />
            <pointLight position={[10, 10, 10]} intensity={1.2} />
            <StarBG radius={100} depth={50} count={5000} factor={4} fade />
            <Stars stars={stars} onStarHover={onStarHover} />
            <Constellations connections={connections} stars={stars} />
            <OrbitControls enableZoom enableRotate enablePan />
        </Canvas>
    );
```

Figure 4.7: Scene Set Up

4.2.2 Scene Composition and React Component Structure

The entry point for 3D rendering is a React component, often named `Scene`, fig. 4.7, which encapsulates the `Canvas` setup and includes lighting, background effects, and sub-components for stars and constellation lines [Cab10].

Here, a `<Canvas>` element sets up the WebGL [Gro21] rendering context. Camera parameters such as position and field of view (`fov`) establish an initial view. Ambient and point lights create a basic illumination model: ambient light ensures that no part of the scene is completely dark, while a point light simulates a localized light source, providing shading cues on spherical stars. The `Stars` component (used as `StarBG`) from the `drei` library injects a particle-based star field background, giving the impression of distant stars or nebulae enveloping the scene.

Within the `<Canvas>` I render two custom components, `<Stars>` and `<Constellations>` in the main content based on arrays passed as props. Finally, `<OrbitControls>` attaches interactive controls to the camera, enabling free exploration.

4.2.3 Star Representation and Labeling

Each star in CelestiMap is represented by a small emissive sphere in 3D space, fig. 4.8.

Each `Star` is placed in the 3D scene using a `<group>` element, whose `position` property is specified as an array `[x, y, z]`. A child `<mesh>` defines both the geometry and material: `sphereGeometry` with a radius of 0.25 and moderate segment counts produces a smooth sphere, while `meshStandardMaterial` with an `emissive` color simulates self-illumination. This allows stars to appear as glowing objects even when not directly illuminated by scene lighting.

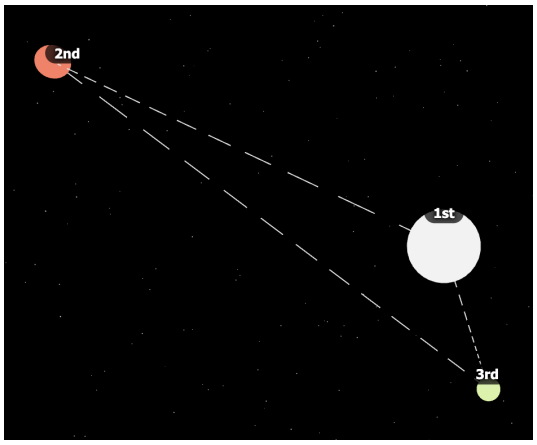

Figure 4.8: Star and Constellation UI

Figure 4.9: Info Panel UI

```
<group position={star.position}>
    <mesh onPointerOver={() => onStarHover(star)} onPointerOut={() => onStarHover(null)}>
        <sphereGeometry args={[0.25, 32, 32]} />
        <meshStandardMaterial
            emissive={star.color || 'white'}
            emissiveIntensity={2}
            color={star.color}
        />
    </mesh>
    <Html position={[0, 0.5, 0]} center>
        <div className="star-label" style={{
            color: 'white',
            textShadow: '0 0 3px black, 0 0 5px black'
        }}>
            {star.name}
        </div>
    </Html>
</group>
```

Figure 4.10: Star Code

Hover interactions are handled using the `onPointerOver` and `onPointerOut` event handlers, which detect when a star is being hovered. These interactions are communicated back to parent components, fig. 4.6, allowing for dynamic behavior such as displaying information panels, fig. 4.9 or triggering highlight effects.

To display a label above the star, the `<Html>`, fig. 4.10, utility from the `drei` library is employed. This utility overlays a DOM element at a specified 3D coordinate, enabling crisp, styled text without the complexity of rendering text directly in 3D space. The position offset `[0, 0.5, 0]` ensures the label appears just above the sphere. Styling is handled via CSS—such as white text with a black text-shadow—to maintain readability against varied backgrounds.

4.2.4 Constellation Lines Rendering

Constellation connections are rendered as dashed lines between star positions. The `Constellations` component typically transforms an array of connections and

```
return (
    <group>
        {connections.map(([fromId, toId], idx) => {
            const start = starPositions[fromId];
            const end = starPositions[toId];
            if (!start || !end) return null;
            return (
                <Line
                    key={idx}
                    points={[start, end]}
                    dashed
                    dashSize={0.5}
                    gapSize={0.3}
                    lineWidth={1}
                    onClick={() => onLineClick(fromId, toId)}
                />
            );
        })}
    </group>
);
```

Figure 4.11: Constellation Code

stars into a set of `<Line>` primitives, fig. 4.11. Connections are represented as pairs of star IDs. For each connection, if both endpoints exist, a `<Line>` from `drei` renders a line segment in 3D. The `dashed` property, along with `dashSize` and `gapSize` parameters, creates a stylized dashed effect, fig. 4.8, evoking constellation diagrams.

4.2.5 Data Flow and State Management

The 3D components receive data via props: an array of star objects (each containing id, position, color, name, additionalInfo) and an array of connection pairs. These props originate from React state in pages such as `ProjectEditor`, which manages the star list via local state or fetched data. When a user adds, edits, or deletes a star or connection, the state updates, and React's reconciliation triggers react-three-fiber to update the scene: new `<Star>` components re-render, positions change, or `<Line>` components update. This declarative pattern ensures synchronization between application state and 3D view without manual WebGL imperative calls.

```
useEffect(() => {
    const saved = JSON.parse(localStorage.getItem(LOCAL_STORAGE_KEY));
    if (saved) {
        setStars(saved.stars || []);
        setConnections(saved.connections || []);
        const minId = Math.min(...saved.stars.map(star => star.id), 0);
        setNextId(minId - 1);
    }
}, []);
```

Figure 4.12: Use of localstorage, getting the information

4.2.6 Lighting, Materials, and Visual Effects

Choosing appropriate lighting and material settings is crucial for an appealing UI. Using a moderate-intensity ambient light prevents overly dark regions, while a point light simulates a distant light source, providing subtle shading cues on spherical stars. The `meshStandardMaterial` with emissive color makes stars appear to glow; `emissiveIntensity` tuned to a value ensures visibility even when the scene is dim. For distant `starfield`, the `<Stars>` background from drei often uses procedural generation to scatter points across a spherical shell, contributing depth and immersion.

4.3 Project Management

Project management in CelestiMap encompasses two complementary modes: an offline mode for guest or temporarily disconnected users, and an online mode for authenticated users interacting with the backend. In addition, the ability to import and export projects in CSV or JSON formats is realized via a Strategy pattern implemented in the backend.

4.3.1 Offline Mode (Local Persistence)

The offline mode enables guest users to build and persist star-map projects entirely in the browser. This is realized through the use of the Web Storage API, specifically `localStorage`, to store the current project state (stars and connections). Upon mounting the editor component, the application attempts to load any previously saved state, fig. 4.12, resulting in changes to stars or connections automatically update local storage.

The editor restores the star and connection lists and sets up a negative nextId counter so that newly added stars have temporary client-only identifiers (e.g., -1, -2, etc.) rather than colliding with backend-assigned positive IDs.

```
useEffect(() => {
    localStorage.setItem(LOCAL_STORAGE_KEY, JSON.stringify({ stars, connections }));
}, [stars, connections]);
```

Figure 4.13: setItem in localstorage

Another effect hook writes changes back to localStorage, fig. 4.13, whenever stars or connections change.

Thus, the offline persistence is automatic: each addition, modification, or deletion of stars/connections triggers serialization of the current model into localStorage. This design allows the user to freely edit without an active network connection, and for the data to remain available even when the browser refreshes. .

4.3.2 Online Mode (Authenticated Project Management)

Authenticated users may save, update, delete, and retrieve projects via REST endpoints on the backend. The frontend's SaveDialog allows choosing "online" as a save format when the user is signed in, fig. 4.14. On invoking save in online mode, the frontend constructs a projectData object matching the backend's CompleteProjectDTO shape, fig. 4.15.

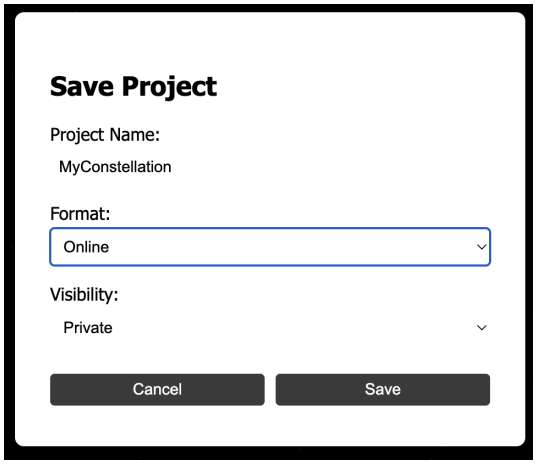

Figure 4.14: SaveDialog Component

```
public record CompleteProjectDTO(
        @Nullable Integer id,
        String name,
        boolean isPublic,
        LocalDateTime createdAt,
        List<StarDTO> stars,
        List<ConnectionLineDTO> connections
) { }
```

Figure 4.15: CompleteProjectDTO

When saving a project online, the application checks whether a `currentProjectId` exists. If it does, the code calls the update endpoint; otherwise, it invokes the save endpoint, fig. 4.16, to create a new project on the server. The backend endpoints are defined in the `PronjectController` and related other related controllers:

- `POST /project/save` receives a `CompleteProjectDTO` and persists it.
- `PUT /project/update` updates an existing project.
- `DELETE /project/delete/{projectId}` removes a project.

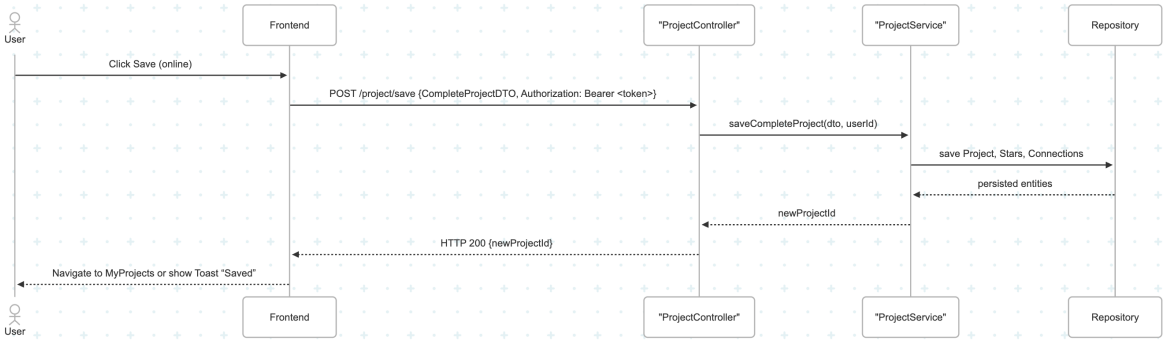

Figure 4.16: FLow Chart for saving online

- `GET /project/{id}` and `GET /project/view/{id}` retrieve project details for editing or viewing purposes.

On the backend, the `ProjectService` class manages the conversion between DTOs and entities. It handles both new and existing stars, determined by whether the star's ID is `null` or greater than zero. It also deals with updating and removing entities. It ensures referential integrity for constellation connections. All operations occur within a transactional context using DTO converters and repository classes, ensuring consistency when multiple related entities (such as stars and connections) are saved or updated together.

4.3.3 Import/Export using Strategy Pattern

A key feature of the application is the ability to import and export projects in CSV or JSON formats. Rather than hard-coding format-specific logic, the backend employs the *Strategy* design pattern.

An interface `FileFormatStrategy` defines the methods `supports(String format)`, `exportProject(CompleteProjectDTO)`, and `importProject(MultipartFile)`. Two concrete implementations are registered as Spring components: `CsvFormatStrategy` and `JsonFormatStrategy`, fig. 4.17.

CSV Format

CsvFormatStrategy uses OpenCSV to read and write tabular representations. During export, it writes a header row, followed by one row per star with columns: `type="STAR"`, `id`, `name`, `x`, `y`, `z`, `color`, and `additionalInfo`. Rows for connections follow, using `type="CONNECTION"`, `id`, with empty values for non-relevant fields, and fields for `startId` and `endId`.

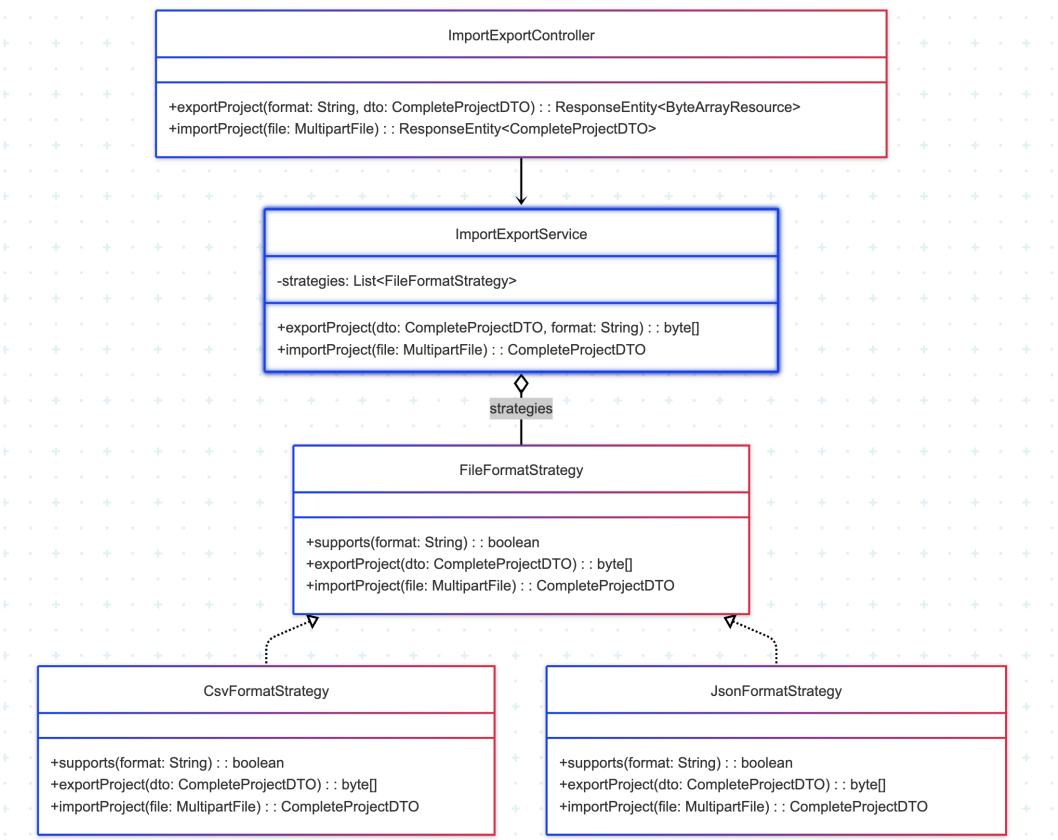

Figure 4.17: UML for Import/Export Strategy

During import, it reads the file, skips the header, distinguishes rows by the `type` column, and constructs either `StarDTO` or `ConnectionLineDTO` objects accordingly. The result is assembled into a `CompleteProjectDTO` with a `null` project ID, empty name, default visibility set to `false`, and a timestamp for `createdAt`. The front-end may then prompt the user to enter a project name or adjust settings before saving online or exporting again.

Json Format

The **JsonFormatStrategy** leverages Jackson's `ObjectMapper` to serialize and deserialize the entire `CompleteProjectDTO` as pretty-printed JSON for export, or to parse uploaded JSON files during import.

Workflow

These strategy implementations are injected into the `ImportExportService`, which iterates through the available strategies and selects the one where `supports(format)` returns `true`. This decouples controller logic from format-specific handling and simplifies future extensions (e.g., support for XML formats).

The import/export functionality is exposed by the `ImportExportController`, with the following endpoints:

- `POST /project/export?format={json|csv}`: Accepts a `CompleteProjectDTO` in the request body, delegates to `ImportExportService.exportProject()`, and returns a downloadable resource with appropriate `Content-Disposition` and MIME type headers.
- `POST /project/import` (multipart/form-data): Accepts an uploaded file, determines the format from the file extension, delegates to `ImportExportService.importProject()`, and returns the parsed `CompleteProjectDTO` as JSON for the frontend to consume.

On the frontend, the `exportProject` function sends the project data to the export endpoint, receives a `Blob` in response, and triggers a file download in the browser. The `importProject` function uploads a file using `FormData`, and the server responds with a `CompleteProjectDTO` object, which is then merged into the editor merging the stars and connections appropriately with what is already availbale in the project.

4.4 Collaboration Features

In CelestiMap, the collaboration environment is composed of three features: a public gallery with pagination and search/sort capabilities, a favoriting system enabling users to bookmark projects, and strict isolation of user-owned projects to enforce access control.

4.4.1 Public Gallery Pagination

To efficiently handle large numbers of public projects, CelestiMap uses pagination. On the backend, Spring Data JPA's `Pageable` interface powers a query that returns only public projects matching an optional search term (by project name or creator), fig. 4.18. The service layer maps each `Project` to a lightweight `DisplayProjectDTO`, and the controller exposes parameters for `page`, `size`, `sort`, and `search`, returning a `Page<DisplayProjectDTO>` that includes `totalPages` and `totalElements` for client navigation.

Front-end Implementation

On the frontend, fig. 4.19, a `usePagination` hook tracks `page`, `size`, `totalPages`, and `totalElements`. The Gallery component builds its API call from these values

```
@Query("SELECT p FROM Project p WHERE p.isPublic = true AND (:searchTerm IS NULL OR " +
       "LOWER(p.name) LIKE LOWER(CONCAT('%', :searchTerm, '%')) OR " +
       "LOWER(p.user.userName) LIKE LOWER(CONCAT('%', :searchTerm, '%')))")
Page<Project> findPublicProjects(@Param("searchTerm") String searchTerm, Pageable pageable
```

Figure 4.18: Repository, query to filter public projects in Pageable

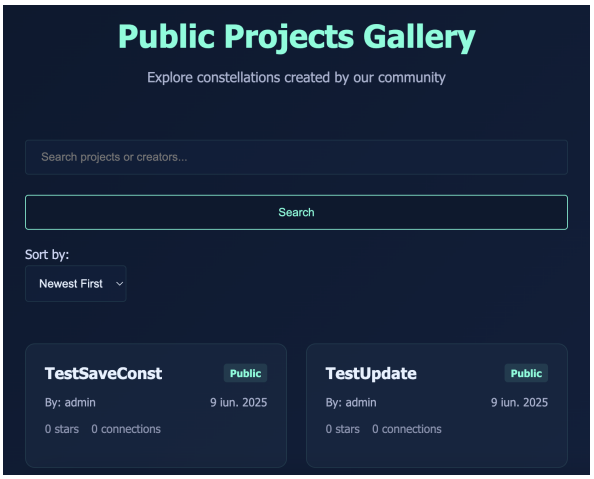

Figure 4.19: Public Gallery UI

plus `sortOption` and `searchTerm`, invoking the endpoint illustrated in Fig. 4.20. Responses update the hook's state, enabling or disabling the "Previous"/"Next" buttons and displaying "Page X of Y." Syncing these parameters into the URL via React Router ensures views remain shareable and bookmarkable.

4.4.2 Favoriting System

Favoriting (bookmarking) enables users to mark items for quick access later, fostering engagement between users. Key features include making sure that only public projects can be favorited, preventing duplicates, and providing an interface to view and remove favorites.

Data Model and Relationships

In the database, favorites are modeled as an entity linking User and Project. This many-to-many relationship (a user can favorite many projects and a project can be favorited by many users) is materialized via the Favorite join table.

```
const response = await fetchPublicProjects({ page: pagination.page, size: pagination.size,
const data = response.data;
setProjects(data.content);
updateMeta(data.totalPages, data.totalElements);
```

Figure 4.20: Call for pagination fetch of public projects

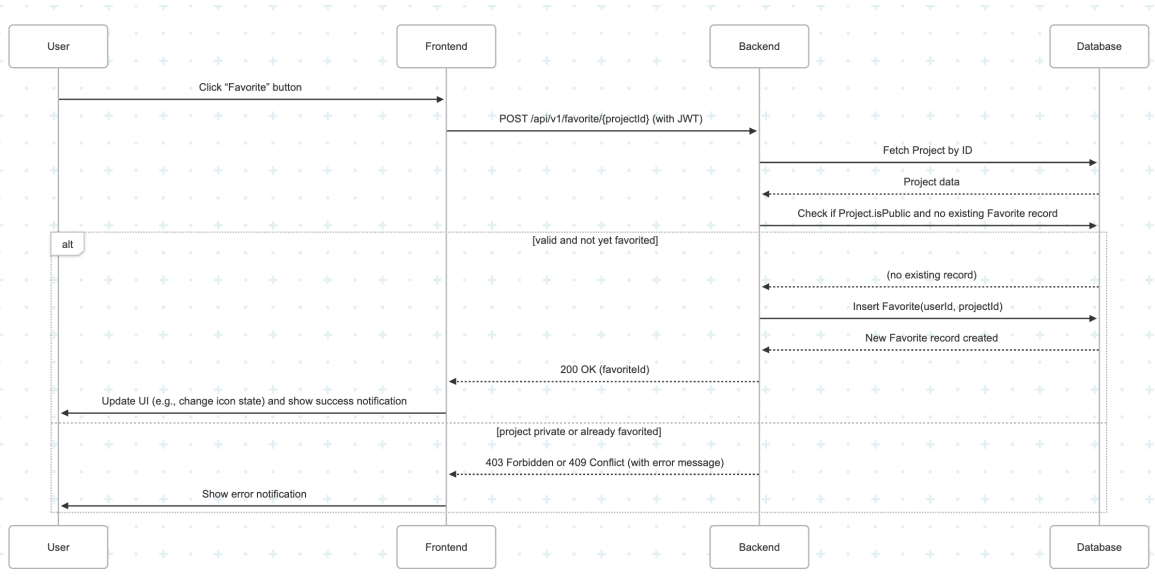

Figure 4.21: Favoriting System Flow Chart

Backend Service and Controller

The FavoriteService deals with the following CRUD operations related to Favorite, fig. 4.21:

- **Add to favorites** - verifies if the target project exists and is public. Then checks if the (user, project) pair already exists to avoid duplicates. Finally, it saves a new Favorite entity.
- **Remove favorite** - locates the Favorite entry by project and user and deletes it.
- **List favorites** - retrieves all Favorite entities for a user, map to Project entities to return the data.

The controller uses the Spring Security's @AuthenticationPrincipal to obtain the current user, ensuring only authenticated users can favorite or view their favorites.

Frontend Implementation

On the client side, two main components relate to favorites `Gallery` and `FavoriteList`.

In Gallery, fig. 4.22, for each project card, if the user is authenticated, a 'Favorite' button is shown. Clicking it invokes addFavorite(user.token, project.id) to POST /favorite/projectId.

FavoriteList is a dedicated page listing all favorited projects. On mount, it calls fetchFavorites(user.token) to GET /favorite. It displays project cards, fig. 4.23 ,similar to the gallery, with a 'Remove' button that calls removeFavorite(user.token, projectId) and updates local state to remove the card upon success.

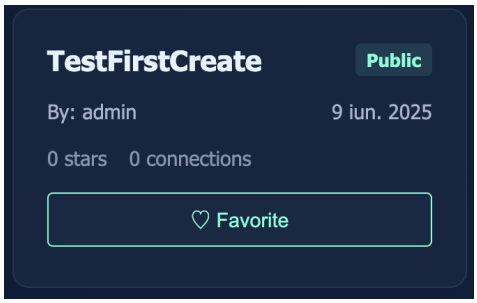

Figure 4.22: Project card UI in Gallery with 'Favorite' button

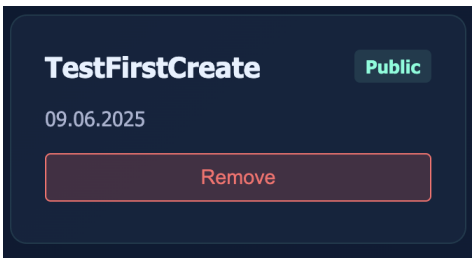

Figure 4.23: Project card UI in Favorite with 'Remove' button

```
public CompleteProjectDTO getProject(Integer projectId, int userId) {
    Project project = projectRepository.findById(projectId)
            .orElseThrow(() -> new EntityNotFoundException("Project not found"));
    if(project.getUser().getUserId() != userId) {
        throw new AccessDeniedException("You don't own this project");
    }
    return completeProjectDTOConverter.createFromEntity(project);
}
```

Figure 4.24: getProject implementation based on isPublic

4.4.3 User-Owned Project Isolation

Access control is critical to ensure users can read/write only their own data. In CelestiMap, projects can be public or private. Private projects should be visible and editable only by their owner. However, public projects may be viewable by anyone (authenticated or guest) but editable only by the owner.

Back-end implementation

The backend enforces isolation where it's necessary.

When Retrieving a project for editing (GET /project/projectId), fig. 4.24, the service method checks that the authenticated user's ID matches the project's owner ID, else throws AccessDeniedException.

Updating or deleting similarly checks if the owner is the right one before performing update or delete operations.

Viewing a public project in read-only mode doesn't need an owner check. It returns the project data but the front-end treats it as non-editable.

Controllers leverage Spring Security to inject the authenticated principal; unauthorized or unauthenticated attempts to access protected endpoints will result in 401/403 responses, fig. 4.25. Thus, private projects cannot be read or mutated by other users.

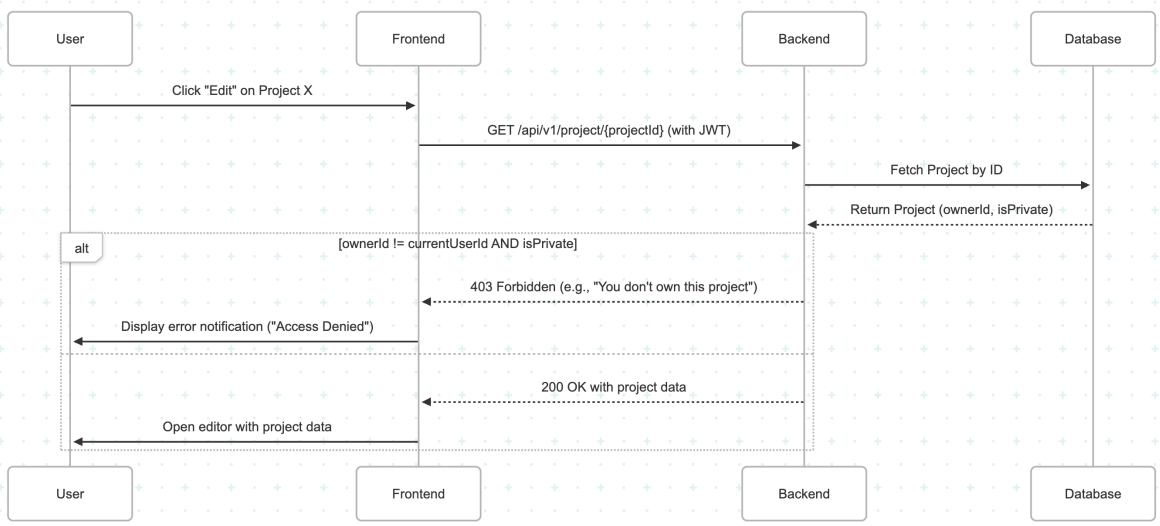

Figure 4.25: Sequence diagram on Project Isolation functions

Front-end implementation

In front-end I have a couple components that deal with the user-owned projects.

My Projects page fetches only the authenticated user's projects with GET /project/myProjects. The backend returns only those owned by the user. UI lists projects with "Edit", "Remove", and "View" actions. Private projects are labeled "Private" and cannot be accessed by other.

When we open ProjectEditor page to edit an already existing project, the front-end retrieves the project with getProject(user.token, projectId). If the user is not the owner the back-end will forbid it, and the front-end would handle the error. However theoretically, a non-authenticated user or a user accessing a project that they don't own should not be possible either way becasue of the way the front-end is rendered.

When navigating from gallery or favorites to ProjectViewer, the frontend calls getProjectView(token?, projectId) which invokes GET /project/view/projectId, allowing any user to load the 3D view in read-only mode.

Theory on Access Control

This design follows the principle of least privilege: endpoints are protected via owner checks, sensitive operations (update/delete) require authentication and ownership verification. Public read endpoints are open, encouraging sharing while preventing unauthorized modifications. In conclusion, this pattern aligns with RESTful best practices for resource-based permissions.

4.5 NASA Astronomy Picture of the Day (APOD)

To enrich CelestiMap with daily astronomical content, I integrated NASA's Astronomy Picture of the Day (APOD) service as both an interactive in-app feature and an optional email notification.

4.5.1 In-App APOD View

Back-end Implementation

On the server side, `APODController` exposes a `GET /api/v1/apod` endpoint. This method delegates to `ApodService.fetchToday()`, which constructs a URI using `UriComponentsBuilder` to call NASA's APOD API (`https://api.nasa.gov/planetary/apod?api_key=\{key\}`), converts the JSON into an `ApodDTO`, and returns it.

Error handling ensures timeouts or invalid API keys result in a controlled 500 response rather than an uncaught exception. By centralizing NASA integration in `ApodService`, I maintain separation of concerns and facilitate unit testing of the REST call logic.

Front-end Implementation

CelestiMap's React front-end includes an `APODModal` component that fetches and displays NASA's daily Astronomy Picture of the Day. An 'APOD' button triggers React state to open the modal. When the modal mounts, a `useEffect` hook invokes the `fetchApod()` service, which issues a GET request to `/api/v1/apod`. On success, the component stores the returned JSON, containing `url`, `hdurl`, `title`, and `explanation`, in local state. Then, the front-end then renders the component fig. 4.26.

4.5.2 APOD Email Notifications

Back-end Email Scheduling and Delivery

The Spring Boot back-end enables scheduling by annotating the main application with `@EnableScheduling`. A dedicated `ApodScheduler` component is defined, fig. 4.27.

The cron expression (defaulting to 10AM daily) is configurable via `application.properties`. `UserNotification` entities track subscriptions with a one-to-one mapping to `User`. The `EmailService` uses `JavaMailSender` and `MimeMessageHelper` to compose an HTML email: it wraps the APOD image

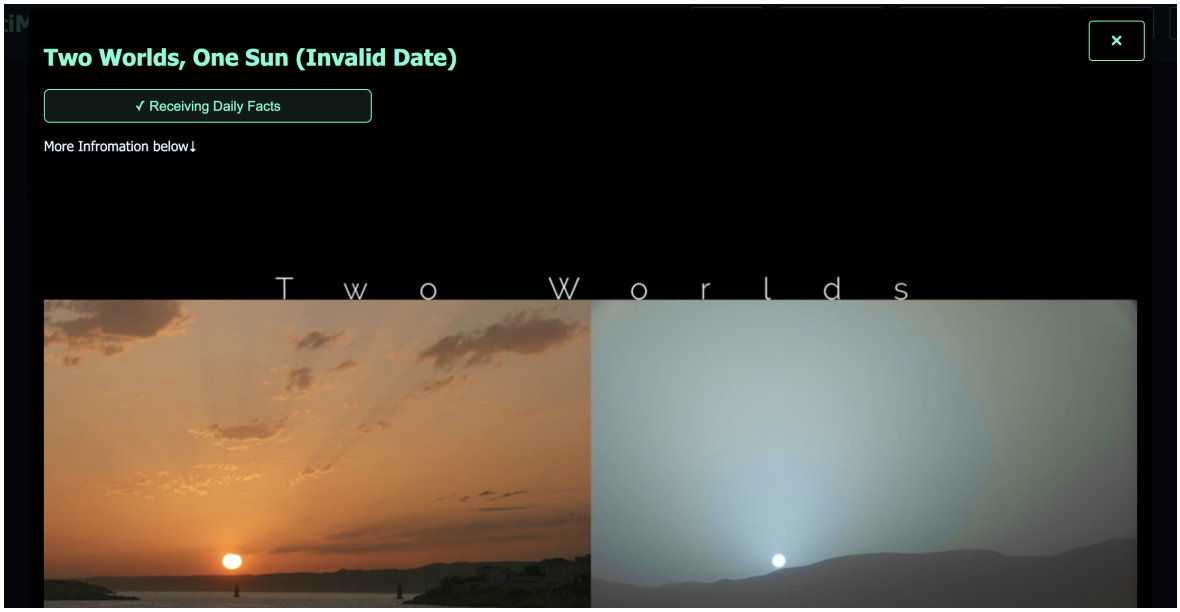

Figure 4.26: ApodModal interafce

```java
@Scheduled(cron = "0 0 10 * * *")
public void sendDailyApod() {
    List<UserNotification> subscribers = userNotificationRepo.findAll();
    String content = apodService.getFunFact();
    for (UserNotification sub : subscribers) {
        try {
            emailService.sendApodEmail(sub.getUser().getEmail(), "Your Daily Astronomy
        } catch (MessagingException e) {
            System.err.println("Failed to email user: " + sub.getUser().getEmail());
        }
    }
}
```

Figure 4.27: ApodScheduler implementation

Figure 4.28: Email View

(linked by URL), title, and explanation in a template that ensures compatibility across major email clients.

Front-end Subscription Management

Authenticated users may opt in or out of daily APOD emails via a toggle in their profile page. The toggle invokes `subscribeApod(token)` or `unsubscribeApod(token)`, POST and DELETE calls to `/api/v1/apod/subscribe` and `/api/v1/apod/unsubscribe`, respectively, using the stored JWT (in `AuthContext`). A third endpoint, `GET /api/v1/apod/subscribed`, checks subscription status on page load to render the subscribe button correctly.

4.6 NeoWs Asteroid Visualization

To bring real near-Earth object (NEO) data into CelestiMap's immersive 3D environment, I implemented a NeoWs Asteroid Visualization feature. This inludes a Three.js component that renders and animates asteroids, and a Spring Boot backend service that fetches and formats NASA NeoWs data.

Back-end Implementation

In teh back-end, the `AsteroidController` exposes an endpoint like `GET /api/asteroids`. This controller calls `NeoWsService.getAsteroids(start, end)` which in turn requests data from NASA's NeoWs feed endpoint. As documented, the NeoWs API's "feed" endpoint returns a list of asteroids based on closest approach dates, for teh moment taken as today's date. The service parses the JSON into asteroid models. Each model includes the NEO's name, size (min/max diameter), velocity, and miss distance. This data is sent to the front-end, which initializes each mesh's state.

Front-end Implementation

In the front-end there is an `NeoVisualization` component which encapsulates the entire 3D view, fig. 4.29. It begins by invoking the custom service `fetchUpcomingNeos(startDate, endDate)` when the user navigates to the 'NEO' page. Once the array of asteroid DTOs is stored in state, `NeoVisualization` renders a `<Canvas>` configured with a distant camera (`position=[2,2,2]`) and interactive `<OrbitControls>`. Each asteroid is a sphere whose radius is proportional to the asteroid's estimated diameter (scaled up by a constant factor to be visible).

A `TimeUpdater` child component uses the `useFrame` hook to advance a normalized time parameter `timeT` from 0 to 1 at a speed controlled by a user-adjustable slider. Each frame, the asteroid's position is recomputed as

$$\mathbf{pos}(t) = \mathbf{startPos} + t\,(\mathbf{endPos} - \mathbf{startPos}),$$

where `startPos` and `endPos` are scaled down by 10^6 to fit the scene.

The controls overlay offers Play/Pause/Replay and a speed slider, and hovering or clicking an asteroid toggles an info panel showing its name, miss distance, and approach date. This declarative rendering and animation loop leverages WebGL's GPU pipeline for smooth interactivity even with dozens of moving objects.

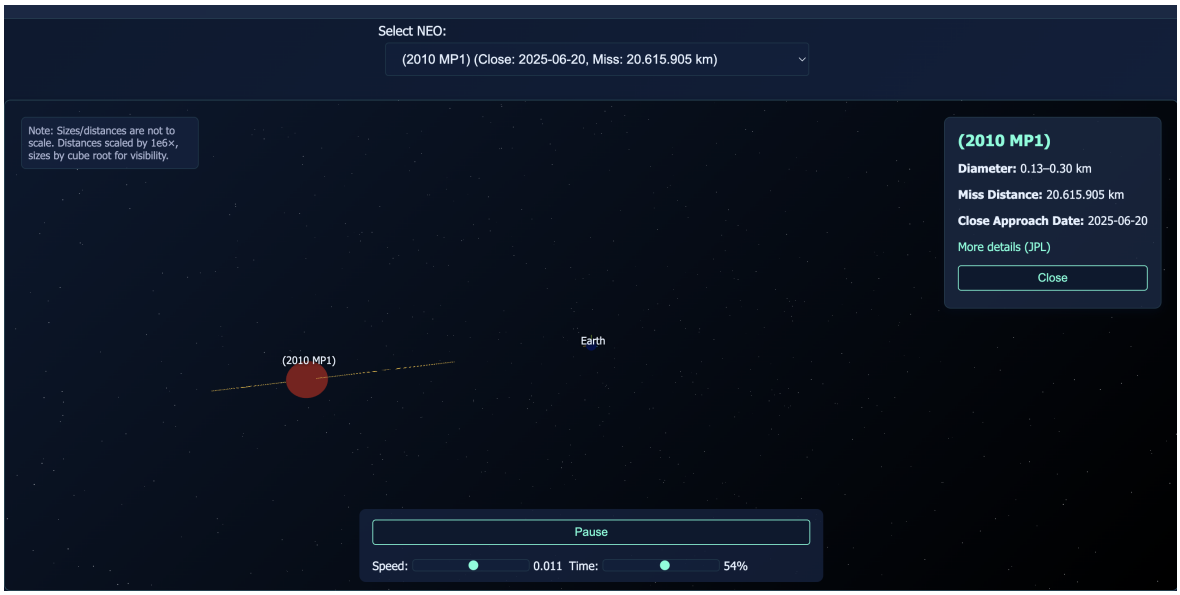

Figure 4.29: NEO scene

Chapter 5

Testing

Testing in modern software is often guided by the test pyramid concept, which advocates having many low-level unit tests, a smaller number of integration tests, and even fewer end-to-end or system-level tests.

For CelestiMap, I'll focus heavily on unit for services and security components. Later it will also be showcased some structures for Postman to drive API-level tests, ensuring that endpoints behave correctly (including authentication flows, permission checks, and business logic).

5.1 Unit Testing Service Layer with JUnit and Mockito

Unit tests for services focus on isolating the service class by mocking its dependencies (repositories, converters, etc.). We commonly use JUnit 5 and Mockito. Each test class is annotated with @ExtendWith(MockitoExtension.class) to enable Mockito annotations.

UserService

Key behaviors include addNewUser, fig. 5.1, which should throw IllegalStateException when username exists, and encode passwords via PasswordEncoder. Another method, checkUser, fig. 5.2, verifies matching via PasswordEncoder.matches.

ProjectService

Tests here mock ProjectRepository, UserRepository, StarRepository, ConstellationLineRepository, and converters, fig. 5.3, verifying that addNewProject, saveCompleteProject, updateProject, getProject and deleteProjectById, fig. 5.4 behave correctly under normal and exceptional conditions.

```
@Test
void addNewUser_whenNew_shouldSaveAndReturnId() {
    String username = "newUser";
    String rawPass = "mypassword";
    when(userRepository.findUserByUserName(username)).thenReturn(Optional.empty());
    when(passwordEncoder.encode(rawPass)).thenReturn("encodedPass");
    User savedUser = new User(1, username, "encodedPass");
    when(userRepository.save(any(User.class))).thenReturn(savedUser);

    int returnedId = userService.addNewUser(username, rawPass);
    assertEquals(1, returnedId);
    verify(userRepository).save(argThat(user -> user.getUserName().equals(username)
        && user.getUserPassword().equals("encodedPass")));
}
```

Figure 5.1: Add new user Test

```
@Test
void checkUser_validCredentials_shouldReturnUserId() {
    String username = "user";
    String rawPass = "raw";
    User user = new User(2, username, "encoded");
    when(userRepository.findUserByUserName(username)).thenReturn(Optional.of(user));
    when(passwordEncoder.matches(rawPass, "encoded")).thenReturn(true);

    int id = userService.checkUser(username, rawPass);
    assertEquals(2, id);
}
```

Figure 5.2: Check new user test

```
@Mock private ProjectRepository projectRepository;
@Mock private UserRepository userRepository;
@Mock private StarRepository starRepository;
@Mock private ConstellationLineRepository constellationLineRepository;
@Mock private ProjectDTOConverter projectDTOConverter;
@Mock private CompleteProjectDTOConverter completeProjectDTOConverter;
@Mock private StarDTOConverter starDTOConverter;
@Mock private ConnectionLineDTOConverter connectionLineDTOConverter;

@InjectMocks private ProjectService projectService;

private User testUser;
private Project testProject;
```

Figure 5.3: Project service tests - mock data

```java
@Test
void deleteProjectById_ifOwner_shouldDeleteProject() {
    when(projectRepository.findById(10)).thenReturn(Optional.of(testProject));
    projectService.deleteProjectById(10, 1);
    verify(projectRepository).delete(testProject);
}

@Test
void deleteProjectById_ifNotOwner_shouldThrowAccessDenied() {
    when(projectRepository.findById(10)).thenReturn(Optional.of(testProject));
    assertThrows(AccessDeniedException.class, () -> projectService.deleteProjectById(10,
    verify(projectRepository, never()).delete(any());
}
```

Figure 5.4: Delete project by id test

```java
void sendApodEmail_returnsFormattedHtml() {
    ApodDTO apod = new ApodDTO();
    apod.setTitle("Galactic Star");
    apod.setUrl("http://image.jpg");
    apod.setExplanation("This is an explanation.");
    ReflectionTestUtils.setField(apodService, "apiKey", "DEMO_KEY");

    when(restTemplate.getForObject(anyString(), eq(ApodDTO.class))).thenReturn(apod);

    String html = apodService.sendApodEmail();
    assertTrue(html.contains("<h2>Galactic Star</h2>"));
    assertTrue(html.contains("<img src=\"http://image.jpg\""));
    assertTrue(html.contains("This is an explanation."));
}
```

Figure 5.5: APOD Send Email Test

APODService

`APODServiceTest` mocks the REST call to NASA's API (using Mockito) and verifies that fetchToday() correctly parses the JSON fields into an `ApodDTO` object. I also test the email logic by mocking JavaMailSender, so we can verify that EmailService.sendApodEmail() sends an email with the expected subject and contains the APOD title in the body, fig. 5.5.

NeoService

Similarly, NeoWsServiceTest verifies that the service correctly handles the NASA NeoWs feed response, fig. 5.6. I provide a sample JSON with known NEO data and check that getUpcoming(start, end) returns a non-empty list with the correct fields.

5.2 Testing Security Components

JwtUtils

For JWTUtils, unit tests verify token creation and validation logic, fig. 5.7.

```
@Test
void getUpcoming_returnsAsteroidsList() {
    List<NeoDTO> mockAsteroids = List.of(
        new NeoDTO("1", "AsteroidX", 123.0, 200.0, "2025-06-10", 50000.0, "http://
    );

    when(mockClient.fetchFeed(any(), any())).thenReturn(Mono.just(mockAsteroids));

    List<NeoDTO> result = neoService.getUpcoming(LocalDate.now(), LocalDate.now().
    assertNotNull(result);
    assertEquals(1, result.size());
    assertEquals("AsteroidX", result.get(0).getName());
}
```

Figure 5.6: NeoService test

```
@Test
void generateAndValidateToken_shouldSucceed() {
    UserDetails userDetails = new CustomUserDetail(new User(1, "user", "pwd"));
    String token = jwtUtils.generateToken(userDetails);
    assertNotNull(token);
    assertTrue(jwtUtils.validateToken(token, userDetails));
    assertEquals("user", jwtUtils.extractUsername(token));
    assertEquals(1, jwtUtils.extractUserId(token));
}
```

Figure 5.7: JWTUtils test

AuthTokenFilter

Testing AuthTokenFilter involves simulating an HTTP request with a valid or invalid Authorization header, and verifying that SecurityContextHolder is populated or not. This can be done through integration testing.

5.3 Testing Import/Export Strategies

For JsonFormatStrategy, tests assert that exporting a CompleteProjectDTO yields byte arrays that parse back to the same DTO, fig. 5.8.

For CsvFormatStrategy, tests ensure that CSV rows are correctly generated and parsed, including handling missing or malformed values. For instance, testing import of a CSV file where a star row has invalid numeric strings, expecting default zero or exception handling, fig. 5.9.

5.4 API-Level Testing with Postman

API-level testing using Postman offers a practical means of validating the deployed REST API. These tests simulate real-world HTTP requests and responses, allowing for verification of endpoint correctness, security enforcement, and error handling

```java
@Test
void exportImportRoundTrip_shouldReturnEquivalentDto() throws Exception {
    CompleteProjectDTO dto = new CompleteProjectDTO(
        null,
        "TestProject",
        true,
        LocalDateTime.of(2025, 6, 12, 10, 0),
        List.of(new StarDTO(null, "Sirius", 1.0, 2.0, 3.0, "#FFFFFF", "Bright star")),
        List.of(new ConnectionLineDTO(null, 1, 2))
    );
    byte[] data = jsonFormatStrategy.exportProject(dto);
    MockMultipartFile file = new MockMultipartFile("file", "test.json", "application/json
    CompleteProjectDTO imported = jsonFormatStrategy.importProject(file);
    assertEquals(dto.name(), imported.name());
    assertEquals(dto.stars().size(), imported.stars().size());
    assertEquals(dto.connections().size(), imported.connections().size());
}
```

Figure 5.8: Json format strategy test export

```java
@Test
void exportImportRoundTrip_shouldPreserveData() throws Exception {
    CompleteProjectDTO dto = new CompleteProjectDTO(
        null,
        "CsvTest",
        false,
        LocalDateTime.now(),
        List.of(new StarDTO(null, "Alpha", 0.1, 0.2, 0.3, "#FF0000", "info")),
        List.of()
    );
    byte[] bytes = csvFormatStrategy.exportProject(dto);
    MockMultipartFile file = new MockMultipartFile("file", "test.csv", "text/csv", by
    CompleteProjectDTO imported = csvFormatStrategy.importProject(file);
    assertEquals(1, imported.stars().size());
    StarDTO s = imported.stars().get(0);
    assertEquals("Alpha", s.name());
    assertEquals("#FF0000", s.color());
}
```

Figure 5.9: CSV format strategy test export

under realistic usage conditions. The following outlines the structure of a Postman test collection designed for CelestiMap's backend API.

Postman Environment Setup

Environment variables are used to streamline test execution. The base URL ({{baseUrl}}) typically points to http://localhost:8080/api/v1 or a deployed endpoint. Upon login or registration, the token is extracted and then applied to all authenticated requests via a pre-configured Authorization header:

```
Authorization: Bearer {{token}}
```

Authentication Flow

The test sequence begins by simulating user registration via a POST request to /api/v1/auth/register, supplying a JSON payload with a username and password. Upon success, a JWT token is returned and extracted from the response body. This token is stored as a Postman environment variable ({{token}}) and is reused in the Authorization header of subsequent requests.

Login is verified through a POST /api/v1/auth/login request. Both successful logins and failed attempts (e.g., due to invalid credentials) are tested. The expected status codes are 200 OK for valid credentials and 401 Unauthorized for invalid ones.

Project Operations

Once authenticated, the API allows users to create, retrieve, update, and delete projects:

- **Create Project:** POST /api/v1/project/save with a JSON body representing a CompleteProjectDTO. A successful request returns the ID of the newly created project.
- **Retrieve Project:** GET /api/v1/project/{id} returns the project's full data structure.
- **Update Project:** PUT /api/v1/project/update modifies the name or visibility status.
- **Delete Project:** DELETE /api/v1/project/delete/{id} removes a project owned by the user.

Public Gallery Access

Publicly available star map projects can be queried via `GET /api/v1/project/public`. Parameters such as `search`, `sort`, and `pagination` are tested to ensure the gallery behaves correctly under various query combinations. The response is verified for the presence of pagination metadata and correct project attributes.

Favorites Functionality

Tests simulate a user favoriting a public project through `POST /api/v1/favorite/{projectId}`. Retrieval of the user's favorites is performed via `GET /api/v1/favorite`, while unfavoriting is tested using `DELETE /api/v1/favorite/{projectId}`. Tests ensure correct status codes (`200 OK` or `204 No Content`) and data consistency.

Chapter 6

Conclusion

CelestiMap is an exciting blend of modern web technologies that creates an engaging and user-friendly platform for exploring and annotating the night sky. It combines a React-based front-end with Three.js for real-time 3D visualizations, backed by a Spring Boot/MySQL server secured with JWT authentication. A guest-mode with local persistence and authenticated online project management lets users seamlessly save, share, and remix their creations, while JSON and CSV import/export capabilities ensure smooth interoperability with other tools and collaborators—all without the need for specialized software.

In addition to these core capabilities, CelestiMap also integrates NASA's Astronomy Picture of the Day, delivering a fresh cosmic image and its explanation each day, both in-app and automatically to subscribed users by email. It also incorporates a NeoWs Asteroid Visualization, fetching real near-Earth object data and animating asteroid trajectories in a scaled 3D scene, thereby bringing the dynamic motions of asteroids into the browser for deeper understanding.

In conclusion, this thesis illustrates the value of using contemporary web standards like WebGL, RESTful APIs, JWT security, and responsive front-end frameworks, to create a unified platform for 3D star mapping. CelestiMap's architecture and implementation not only achieve the immediate objectives of interactive visualization, project management, and community engagement but also point toward broader possibilities in web-based scientific tools. By demonstrating how such an application can be built, tested, and deployed, this work contributes both a practical system and a reference model for future endeavors in web-driven astronomical exploration and education.

Bibliography

[BPM15] Emmanuel Bertin, Ruven Pillay, and Chiara Marmo. Web-based visualization of very large scientific astronomy imagery. *Astronomy and Computing*, 10:43–53, 2015.

[Cab10] Ricardo Cabello. Three.js: Javascript 3d library. `https://threejs.org/`, 2010. Accessed: 2025-06-12.

[Con25] Stellarium Web Engine Contributors. Stellarium web engine. JavaScript/WebAssembly planetarium renderer, 2025. `https://github.com/Stellarium/stellarium-web-engine`.

[Dev25] Stellarium Developers. Stellarium – desktop planetarium software. `https://stellarium.org`, 2025.

[DHG+18] Tim Dykes, Adrian Hassan, Claudia Gheller, Darren Croton, and Maximilian Krokos. Interactive 3d visualization for theoretical virtual observatories. *Monthly Notices of the Royal Astronomical Society*, 477(2):1495–1505, 2018.

[FH20] Loraine Franke and Daniel Haehn. Modern scientific visualizations on the web. *Informatics*, 7(4):37, 2020.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[Fre25] Freshcode. Stellarium 25.1 release notes. freshcode.club project release summary, 2025. Overview of Gaia DR3 star catalog (¿220 million stars).

[Gro17] Khronos WebGL Working Group. Webgl – javascript 3d graphics api. `https://khronos.org/webgl`, 2017.

[Gro18] Stellarium Discussion Group. Discussions on webassembly port of stellarium. Google Groups post, 2018. stellarium-web-engine architecture discussion.

[Gro21] Khronos Group. Webgl 2.0 (opengl es 3.0) specification version 2.0.5. Technical report, 2021. Online specification: urlhttps://registry.khronos.org/webgl/specs/latest/2.0/.

[JBS15] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, Internet Engineering Task Force, 2015.

[Kel23] Mike Kelly. JSON Hypertext Application Language. Internet-Draft draft-kelly-json-hal-11, Internet Engineering Task Force, October 2023. Work in Progress.

[Met25] Meta Platforms, Inc. *React: A JavaScript library for building user interfaces*, 2025. Official documentation.

[NAS21] NASA Jet Propulsion Laboratory. Nasa's "eyes on asteroids" reveals our near-earth object neighborhood. *NASA JPL News*, Dec 2021. Describes interactive 3D NEO visualization using JPL data.

[NAS25] NASA. Astronomy picture of the day. NASA website, 2025. Accessed 2025-06-15.

[Ora25] Oracle Corporation. *MySQL 8 Reference Manual*, 2025. Official documentation.

[Poi25a] Poimandres Open Source. *Drei: Useful helpers for react-three-fiber*, 2025. Official documentation.

[Poi25b] Poimandres Open Source. *React-Three-Fiber: A React renderer for Three.js*, 2025. Official documentation.

[Red19] Red Hat, Inc. *Java Persistence API (JPA) Documentation*, 2019. Red Hat EAP Development Guide.

[RM20] Prateek Rawat and Archana N. Mahajan. Reactjs: A modern web development framework. *International Journal of Innovative Science and Research Technology*, 5(11):698–702, 2020.

[Sho25] Shopify (React Router Team). React router (v7) documentation, 2025. Official documentation.

[WfG25] Dj Walker-Morgan and Sky Map Team (formerly Google). Sky map: Open-source hand-held planetarium for android, 2025. `https://github.com/sky-map-team/stardroid`.