

BABEȘ-BOLYAI UNIVERSITY

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SeatSurfer: A Database-Centric Seat Management System

Bachelor's Thesis

Author: Prodan Radu-Matei

Supervisor: Lect. Dr. Pop Emilia

Specialization: Computer Science and Mathematics
(English)

Year: 2025

Abstract

Modern workplaces are increasingly adopting hybrid work models, where employees alternate between remote and on-site schedules. This evolution has introduced a major logistical problem: the efficient, flexible management of office seating. Traditional fixed desk assignments are no longer effective in such dynamic environments, leading to underutilized spaces, booking conflicts, and administrative overhead.

SeatSurfer directly addresses this challenge by providing a centralized, real-time seat reservation and management platform. The application enables employees to view available seats, make reservations, and manage their bookings through a responsive cross-platform interface built with Flutter. Administrators can configure office layouts, monitor occupancy levels, and maintain control over space allocation.

The system relies on a robust backend developed with Spring Boot and PostgreSQL, ensuring data integrity, transaction safety, and rapid query performance. Security is enforced through role-based access control implemented with Spring Security, differentiating user and administrator privileges. SeatSurfer's design emphasizes modularity, allowing future extensions such as intelligent seat suggestions, occupancy prediction, and hardware sensor integrations.

Development followed Agile methodology, enabling iterative delivery, continuous feedback, and rigorous testing. SeatSurfer demonstrates how modern database-driven architectures, combined with cloud-ready deployment strategies and user-centric design, can effectively solve critical problems of space management in hybrid work environments.

Contents

Abstract	1
1 Introduction	4
1.1 Context and Motivation	4
1.2 Purpose and Research Objectives	4
1.3 Structure of the Thesis	5
2 Related Work and System Benefits	6
2.1 Commercial Tools and Industry Trends	6
2.2 Academic and Open Source Approaches	6
2.3 Innovations Introduced by SeatSurfer	7
2.4 Benefits of the Custom-Built Architecture	8
3 Technologies Used	9
3.1 Overview of the Stack	9
3.2 Flutter: The User Interface Layer	10
3.3 Spring Boot: RESTful Backend	11
3.4 PostgreSQL: Relational Database Engine	12
3.5 Spring Security: Session and Role-Based Access	14
3.6 Development Workflow	15
4 System Architecture	17
4.1 Architectural Overview	17
4.2 Entity-Relationship Diagram	18
4.3 Backend Layered Architecture	19
4.4 Class Diagram	21
4.5 Booking Sequence Diagram	22
4.6 Data Flow and Deployment Model	22
5 Application Functionality and Testing	24
5.1 Overview of Functional Requirements	24
5.2 Features for Regular Users	25
5.3 Features for Administrators	26
5.4 Design Philosophy and UI Decisions	26
5.5 Testing Strategy	27
5.6 Screenshot Examples	28
5.7 Test Results and Known Limitations	28
5.8 Summary	29

6	Database Design and Optimization	30
6.1	Relational Schema Design	30
6.2	Normalization and Design Rationale	31
6.3	Query Examples	31
6.4	Indexes and Performance	32
6.5	Scalability Considerations	33
6.6	Summary	33
7	Security and Access Control	35
7.1	Security Objectives	35
7.2	Authentication Architecture	36
7.3	Authorization and Role-Based Access	36
7.4	Session and Token Handling	37
7.5	Vulnerability Mitigation Strategies	38
7.6	Logout and Expiry Behavior	39
7.7	Testing and Validation	39
7.8	Summary	40
8	Agile Development and Project Workflow	42
8.1	Agile Methodology in Practice	42
8.2	Team Structure and Roles	42
8.3	Sprint Planning and Management	43
8.4	Tools Used for Collaboration	43
8.5	Challenges and Lessons Learned	43
8.6	Summary	44
9	Analytics and Admin Dashboard	45
9.1	Overview	45
9.2	Implemented Feature: Occupancy Data by Date	45
9.3	Backend Logic and API Interaction	45
9.4	Limitations and Future Improvements	46
9.5	Summary	46
10	Future Enhancements and Research Directions	47
10.1	Overview	47
10.2	AI-Powered Seat Recommendations	47
10.3	Occupancy Sensor and IoT Integration	48
10.4	Enhanced Reporting and Analytics	48
10.5	Scalability and Multi-Tenant Support	48
10.6	Potential Academic Research Topics	49
10.7	Conclusion	49
A	Entity Class Example (Java)	50
B	REST Controller Sample	51
C	SQL Table Definitions	52
D	Flutter Code Sample	53

Chapter 1

Introduction

1.1 Context and Motivation

The COVID-19 pandemic catalyzed a fundamental shift in workplace models. Organizations worldwide began transitioning to hybrid systems, where employees alternate between working remotely and on-site. This shift introduced new logistical challenges — particularly regarding the use and optimization of physical office space.

The traditional static assignment of desks became inefficient in a flexible environment. Fixed seating arrangements often led to large areas of underutilized space, while certain days saw unexpected crowding. As workforces became more mobile and flexible, the need for a dynamic, real-time management solution became critical. Companies needed a way to dynamically manage and track seat usage, prevent overcrowding, respect safety regulations, and offer employees autonomy over their workspace choices.

Beyond logistical issues, poor space management directly impacts organizational costs. Maintaining large offices with inefficient seat utilization leads to increased operational expenses, underused real estate, and employee dissatisfaction due to lack of visibility or transparency in desk availability.

While several commercial seat booking and space management solutions exist, they present significant barriers to adoption. Many are either prohibitively expensive for small and mid-sized organizations, overly complex for simple needs, or lack proper integrations with existing internal tools. Furthermore, dependency on third-party cloud services often raises concerns about data ownership, security, and customization flexibility.

In this context, SeatSurfer was conceived as a lightweight, customizable, and secure solution to the growing challenges of hybrid workplace management. By leveraging open technologies and focusing on modular, database-driven architecture, SeatSurfer empowers organizations to regain control over their office spaces without the overhead of complex, vendor-dependent platforms.

1.2 Purpose and Research Objectives

This thesis is driven by the goal of designing and implementing a robust, modular, and database-centric platform for intelligent seat management in hybrid work environments. The development of SeatSurfer seeks not only to deliver a functional software product but also to demonstrate best practices in full-stack system design, scalable architecture,

and secure data handling.

The specific research and development objectives of this project are:

- **Development of a fully functional booking platform:** SeatSurfer aims to provide a complete seat reservation system built entirely with open technologies, ensuring transparency, extensibility, and vendor independence.
- **Implementation of secure, role-based access control:** A core requirement is to differentiate functionalities between regular users and administrators, enforcing strict access control policies and ensuring that sensitive administrative operations are properly protected.
- **Optimization of the relational data model:** The database schema is designed to facilitate real-time occupancy tracking, dynamic seat layouts, and efficient query performance, enabling the system to scale to varying organizational sizes without degradation.
- **Validation through systematic testing and benchmarking:** The application must be rigorously evaluated through functional testing, security assessments, and performance benchmarks to confirm its reliability and readiness for real-world deployment.
- **Contribution to database-centric application design knowledge:** By analyzing the challenges and trade-offs encountered during development, this thesis offers insights into how database normalization, RESTful API architecture, and modular full-stack design can be effectively combined in modern applications.

Through these objectives, the project not only delivers a practical solution to the hybrid workspace management problem but also contributes academically to the field of applied software engineering and database systems.

1.3 Structure of the Thesis

The thesis is divided into the following chapters:

- **Chapter 2:** Examines existing tools and academic approaches
- **Chapter 3:** Explores the technology stack and tool choices
- **Chapter 4:** Describes the system architecture, database design, and data flow
- **Chapter 5:** Presents the application functionality and testing methodology
- **Chapters 6–10:** Go deeper into database normalization, security, agile development, analytics, and future work (next delivery)

Chapter 2

Related Work and System Benefits

2.1 Commercial Tools and Industry Trends

The rise of hybrid and hot-desking models in companies has led to a proliferation of software products designed to manage workplace occupancy. Notable examples include:

- **Robin** – Offers interactive maps and analytics, tailored for enterprise-level integration
- **Deskbird** – Focuses on user-friendliness and mobile-first experiences
- **Skedda** – Provides a more general space booking platform used in co-working spaces
- **Microsoft Places / Outlook Room Finder** – Integrated into Microsoft’s ecosystem, targeted at calendar-based reservations

While these tools offer functional solutions, they come with limitations:

- High subscription costs for teams
- Limited customizability
- Vendor lock-in with cloud infrastructure
- Lack of transparent access to data models or APIs

2.2 Academic and Open Source Approaches

Several academic research initiatives and open-source projects have tackled problems related to scheduling systems, particularly in the context of classroom management, laboratory reservations, and resource allocation in educational environments.

However, while valuable, these systems exhibit significant limitations when evaluated against the demands of a modern hybrid workplace. Common shortcomings include:

- **Administrative Focus:** Many academic or open-source systems are designed primarily for administrators to manage assets but do not offer rich, real-time user experiences for employees or students.

- **Poor Usability and Limited Mobile Access:** Usability and accessibility are often secondary priorities, resulting in interfaces that are non-responsive or difficult to use on smartphones and tablets.
- **Static Data Models:** Many systems assume static resource inventories rather than dynamically changing floor plans, which are essential in flexible hybrid workspaces.
- **Security and Access Control Gaps:** Several open-source schedulers lack modern authentication mechanisms like OAuth2 or role-based access control, limiting their applicability in corporate environments.

In contrast, **SeatSurfer** provides a dynamic, scalable, and secure system explicitly designed to meet the needs of hybrid workplaces. By leveraging modern frameworks such as Flutter for the frontend and Spring Boot for the backend, and by focusing on user-centered design and real-time interaction, SeatSurfer addresses gaps left open by earlier academic and open-source efforts.

The platform offers the following key advantages:

- **Full-stack architecture:** A responsive, modern user interface developed in Flutter, integrated with a robust, RESTful backend powered by Spring Boot.
- **Custom data model:** A relational schema specifically designed for flexible desk and seat management, supporting dynamic layouts and real-time availability tracking.
- **Transparency and extensibility:** An openly structured database optimized for future enhancements, including advanced analytics, reporting modules, and integrations with third-party tools or sensors.

2.3 Innovations Introduced by SeatSurfer

SeatSurfer introduces a series of key innovations that clearly distinguish it from conventional off-the-shelf SaaS seat management platforms. Rather than enforcing a rigid, vendor-controlled framework, SeatSurfer was designed with openness, flexibility, and user empowerment as core principles.

First, SeatSurfer enables organizations to retain full sovereignty over their infrastructure by offering a self-hosted deployment model. Unlike SaaS offerings that require data and operations to reside on third-party servers, SeatSurfer can be installed and managed internally, ensuring complete ownership of sensitive corporate data and compliance with internal security policies.

Second, the system offers full control over workspace layouts, seat configurations, and booking rules. Administrators are not constrained by predefined templates or limited customization options. Every aspect of the seat management logic — from the physical floor map to booking restrictions and user access levels — can be tailored to the organization’s specific operational needs.

Third, SeatSurfer introduces intuitive, real-time floor visualization capabilities. Employees can interact with dynamic seat layouts that accurately reflect the current availability, allowing for immediate booking decisions. This visual approach improves

user experience, enhances booking efficiency, and reduces friction compared to text-based or list-driven interfaces commonly found in legacy systems.

Finally, SeatSurfer is designed as a modular platform ready for future enhancements. Its architecture supports the seamless integration of additional features such as occupancy analytics, automated notifications, AI-driven seat suggestions, or Internet-of-Things (IoT) sensor inputs without major restructuring. This extensibility ensures that SeatSurfer can evolve alongside organizational requirements, offering a long-term, future-proof solution for hybrid workspaces.

2.4 Benefits of the Custom-Built Architecture

Developing SeatSurfer entirely in-house brings both pedagogical and strategic advantages that extend far beyond the immediate technical implementation.

First, from an educational standpoint, designing the entire system — from database schema to frontend interaction — fosters a much deeper understanding of full-stack architectures. It exposes developers to critical aspects of modern software engineering, including client-server communication, database normalization, API design, security best practices, and cross-platform user interface development. This holistic exposure is rarely achievable when working with pre-packaged third-party solutions.

Second, the custom-built approach enforces best practices in software engineering, such as reusable component design, clean separation of concerns between application layers, and modular code organization. By adhering to these principles, SeatSurfer achieves greater maintainability, scalability, and adaptability over time.

Strategically, an in-house system offers unparalleled flexibility. Unlike closed commercial platforms, SeatSurfer can integrate seamlessly with existing internal tools such as Single Sign-On (SSO) systems, employee databases, building access controls, or corporate calendars. New features and integrations can be developed without reliance on external vendors or prohibitive licensing models.

Moreover, SeatSurfer dramatically improves organizational responsiveness to change. As company needs evolve — for example, relocating offices, reconfiguring seating plans, or adjusting reservation policies — the system can be updated swiftly and at minimal cost. This agility is a significant strategic asset in the post-pandemic era of hybrid work, where office configurations may need to change rapidly based on employee attendance patterns, health guidelines, or space optimization strategies.

Finally, by combining academic software development principles with practical usability goals, SeatSurfer establishes itself as a viable, sustainable alternative to expensive commercial seat management platforms. It empowers organizations with full ownership of their systems and data, providing the rare combination of customization, cost efficiency, operational control, and technical extensibility.

Chapter 3

Technologies Used

3.1 Overview of the Stack

The SeatSurfer platform was built using a carefully selected, modular technology stack designed to address the unique requirements of hybrid office space management. This stack supports rapid feature development, cross-platform deployment, secure access control, and efficient database operations. Each component was chosen for its maturity, community support, and alignment with the project’s goals of scalability, usability, and robustness.

- **Frontend:** Flutter is Google’s open-source UI toolkit that enables developers to build high-performance applications from a single codebase for Android, iOS, web, and desktop. For SeatSurfer, Flutter powers the entire user interface, including seat layout visualizations, calendar-based bookings, and responsive administrative dashboards. Its widget-based, declarative framework facilitates the creation of clean, interactive designs while its hot-reload feature greatly speeds up the development and testing cycle. The use of Flutter ensures platform consistency and significantly reduces the time and effort required to build and maintain separate mobile and web applications.
- **Backend:** Spring Boot serves as the backbone of SeatSurfer’s backend logic, offering a rapid way to build production-ready RESTful APIs. Leveraging the broader Spring ecosystem, it provides robust dependency injection, automatic configuration, and a layered architecture that separates business logic from presentation and persistence concerns. In SeatSurfer, Spring Boot handles all core services such as booking validation, user management, access control, and database interactions. It also facilitates rapid prototyping and testing with embedded servers and in-memory databases for development purposes.
- **Database:** PostgreSQL is a powerful, enterprise-grade relational database system that offers full SQL compliance, ACID transactions, and advanced data modeling capabilities. It was selected for SeatSurfer due to its performance with complex relational queries, support for constraints and foreign keys, and ability to handle normalized schemas efficiently. The schema includes interconnected entities such as users, floors, seats, and bookings, all maintained with referential integrity. PostgreSQL also supports indexing strategies that optimize lookup speed for real-time seat availability and conflict detection during booking operations.

- **Authentication:** *Spring Security* — This framework integrates seamlessly with Spring Boot to provide comprehensive authentication and authorization capabilities. It enforces access control via role-based permissions (e.g., USER vs. ADMIN), supports form-based and token-based login methods, and protects endpoints against unauthorized access. In SeatSurfer, user credentials are securely stored using BCrypt hashing, and all protected operations (such as floor configuration or booking deletion) are guarded by endpoint-level access rules. This guarantees that users can only access resources appropriate to their assigned roles.
- **Other Tools:**
 - **Postman** — An API client used extensively for testing and debugging back-end endpoints. It allowed developers to verify request and response formats, test edge cases (e.g., double bookings), and inspect HTTP headers for role-based access testing.
 - **GitHub** — Served as the version control system, enabling collaborative development, pull request reviews, issue tracking, and continuous integration. Git branches were used to isolate features and fix bugs in parallel by different team members.
 - **Trello** — A task management tool used to organize the team’s Agile workflow. Trello boards contained categorized task columns such as To Do, In Progress, In Review, and Done, which facilitated sprint planning, daily stand-ups, and retrospectives.
 - **IntelliJ IDEA** — The primary integrated development environment for Java and Spring Boot development. Its features like live templates, refactoring tools, integrated terminal, and Maven support improved backend productivity and code maintainability.
 - **Visual Studio Code** — Used for developing and debugging the Flutter frontend. Its extensive plugin ecosystem and support for Dart and Flutter extensions made it the ideal editor for building responsive UIs and integrating with REST APIs.

This stack was deliberately selected to balance development speed, scalability, cross-platform support, and ease of deployment. By using technologies that are open-source and widely adopted in industry, SeatSurfer ensures long-term maintainability and adaptability to future requirements.

3.2 Flutter: The User Interface Layer

Flutter, developed by Google, is a modern open-source framework that empowers developers to build natively compiled applications from a single codebase for Android, iOS, web, and desktop. This cross-platform capability is crucial for projects like SeatSurfer, which aim to provide a seamless and consistent user experience across a variety of devices without duplicating development effort.

Flutter’s architecture is built around a high-performance rendering engine (Skia), allowing it to bypass native UI components and instead render every pixel directly,

ensuring full control over the look and feel of the application. It provides a fast development cycle, with features such as stateful hot reload, making it ideal for iterative UI development and debugging.

Key features that made Flutter suitable for SeatSurfer include:

- **Declarative widget system:** Inspired by React, Flutter uses a tree of composable widgets for building UIs. Each widget describes part of the interface, and the framework efficiently rebuilds only the affected parts of the UI when state changes occur. This approach promotes reusable, testable components.
- **Rich UI controls and animation libraries:** Flutter includes a vast set of pre-built widgets that adhere to both Material Design (Android) and Cupertino (iOS) principles, along with powerful animation APIs for building polished interfaces. This enabled the SeatSurfer team to design intuitive transitions, dialogs, and dynamic visual feedback (e.g., seat hover and selection animations).
- **Integrated HTTP packages:** The `http` and `dio` packages allow Flutter to easily consume RESTful APIs. These were used extensively in SeatSurfer to communicate with the backend services for actions like login, fetching seat availability, creating or canceling bookings, and accessing analytics data.

In the context of SeatSurfer, Flutter was instrumental in implementing a responsive, interactive interface tailored for both regular users and administrators:

- **Visual matrix-based seat layout:** A dynamic, color-coded grid that visually represents the seat availability on each floor. Each seat cell can be tapped to trigger bookings or view status, with real-time UI updates based on API responses.
- **Interactive date picker and floor selection screens:** These components allow users to quickly select the date and floor for which they want to view or reserve seats. These selections trigger live data updates from the backend, ensuring that users always see accurate seat availability.
- **Responsive admin dashboards and booking listings:** Admins access a separate interface for configuring layouts, viewing usage statistics, and managing bookings. The UI adapts fluidly across devices and screen sizes, maintaining usability whether accessed from a mobile phone, tablet, or desktop browser.

By leveraging Flutter's cross-platform consistency, rich visual tools, and high development velocity, the SeatSurfer frontend achieved a professional, mobile-first interface with minimal overhead. This not only improved user experience but also accelerated development and testing cycles throughout the project.

3.3 Spring Boot: RESTful Backend

Spring Boot is a robust, production-grade framework built on top of the Spring ecosystem. It simplifies the setup and development of stand-alone, enterprise-grade applications in Java by providing pre-configured defaults, embedded servers (like Tomcat), and a vast ecosystem of modular components. Spring Boot enables rapid development of RESTful APIs, supports microservice architectures, and includes essential features

such as dependency injection, aspect-oriented programming, and robust testing frameworks.

At the core of Spring Boot lies the concept of convention over configuration, significantly reducing boilerplate code and letting developers focus on business logic. The framework also integrates smoothly with popular tools and technologies such as Maven, Gradle, and Docker, making it ideal for scalable backend services.

In the SeatSurfer platform, Spring Boot serves as the backbone of the server-side logic. It is responsible for managing business workflows, securing endpoints, validating input data, and coordinating database operations. Specifically, Spring Boot handles:

- **Booking creation, validation, and cancellation logic:** When a user attempts to book a seat, the system checks for availability, prevents double-booking, and persists the request. Similar workflows exist for booking cancellation and conflict resolution, all encapsulated within service-layer components.
- **Role-based endpoint security using Spring Security:** Spring Security provides a comprehensive authentication and authorization framework. In SeatSurfer, it ensures that only authenticated users can access the booking endpoints, and that administrative privileges are required for management actions such as editing floor layouts or generating occupancy data.
- **Layered separation (Controller → Service → Repository):** The backend is structured following the layered architecture model:
 - The **Controller layer** maps HTTP requests to Java methods and handles incoming API calls.
 - The **Service layer** contains business logic, such as validating booking dates or checking user roles.
 - The **Repository layer** interacts with the PostgreSQL database using JPA repositories.

This clean separation promotes modularity, testability, and easier maintenance.

Spring Boot also includes built-in support for automatic serialization of Java objects into JSON responses, making it ideal for use in REST APIs. Through annotations like `@RestController`, `@GetMapping`, and `@PostMapping`, the backend exposes intuitive, well-structured endpoints that are consumed by the Flutter frontend.

To persist data, Spring Boot uses **Spring Data JPA**, a high-level abstraction over the Java Persistence API. It simplifies the implementation of repositories through interfaces, automatically generating queries based on method naming conventions, and supporting custom query definitions using JPQL or native SQL when needed.

By leveraging the Spring Boot framework, SeatSurfer benefits from a powerful and secure backend infrastructure that scales easily, integrates well with modern databases, and remains maintainable as the system evolves. It enables high developer productivity while upholding industry best practices for backend service development.

3.4 PostgreSQL: Relational Database Engine

PostgreSQL is a powerful, open-source relational database system known for its reliability, feature richness, and strict adherence to SQL standards. It was chosen for

the SeatSurfer platform because it provides an optimal balance between performance, flexibility, and data integrity, making it particularly suitable for systems with complex relational logic like seat reservations and occupancy management.

Key reasons for selecting PostgreSQL include:

- **Full SQL support and transaction safety:** PostgreSQL supports all standard SQL operations and features ACID compliance, which ensures atomicity, consistency, isolation, and durability of transactions. This guarantees that booking operations in SeatSurfer, such as seat allocation or cancellation, are executed reliably and safely — even under concurrent access.
- **Strong indexing capabilities for read optimization:** PostgreSQL allows the creation of advanced index types (e.g., B-tree, hash, GIN, and GiST), which drastically improve the performance of frequent queries. SeatSurfer takes advantage of this by indexing critical columns like `booking_date`, `seat_id`, and `floor_id` to enable fast lookups for available or occupied seats.
- **Support for complex joins and relational constraints:** The platform relies on multiple table relationships (e.g., many-to-one between bookings and users), which PostgreSQL handles seamlessly. It supports foreign key constraints, cascading actions, and sophisticated joins, enabling accurate tracking of bookings across users, seats, and floors.

In SeatSurfer, PostgreSQL serves as the persistent data store for all core entities, modeled as normalized tables to maintain data consistency and avoid redundancy. The main entities include:

- **Users:** Stores essential authentication and authorization details, including `id`, `email`, and `role`. This table is directly linked to the bookings via a foreign key, enforcing referential integrity.
- **Floors:** Represents physical office layouts with attributes like `id`, `name`, and layout dimensions (e.g., number of rows and columns). Each floor acts as a container for a specific group of seats.
- **Seats:** Each seat is associated with a particular floor and is defined by its row and column coordinates, along with its activation status (available or not). This table supports real-time seat visualization in the frontend.
- **Bookings:** Captures each reservation made by users, storing the `userId`, `seatId`, and the `booking_date`. Constraints ensure that a seat cannot be double-booked on the same date.

Additionally, PostgreSQL's extensibility allows for future enhancements such as stored procedures, triggers for auditing changes, or even partitioning strategies for scalability. Its compatibility with modern cloud platforms (e.g., Railway, Supabase, or AWS RDS) ensures that SeatSurfer's database layer can evolve alongside growing user demands.

Through its robust relational modeling, referential integrity enforcement, and performance tuning capabilities, PostgreSQL forms the reliable foundation of the SeatSurfer application's backend ecosystem.

3.5 Spring Security: Session and Role-Based Access

Authentication and authorization in SeatSurfer are implemented using **Spring Security**, a highly customizable framework that integrates seamlessly with Spring Boot to protect both web and REST APIs. It serves as the cornerstone of the platform's access control mechanism, ensuring that only authorized users can access sensitive features and data.

Spring Security enables robust, production-grade security policies with minimal boilerplate code. In the context of SeatSurfer, it offers full-stack protection through:

- **Basic login via email and password:** Users authenticate using email credentials and a securely hashed password. The system employs the `BCryptPasswordEncoder` algorithm to store credentials as salted hashes, significantly reducing the risk of credential theft and brute-force attacks.
- **Role-based access control (RBAC):** Each user is assigned a role, stored in the PostgreSQL database as either `ROLE_USER` or `ROLE_ADMIN`. These roles dictate which parts of the application and API endpoints a user can access. For example, only administrators can modify floor layouts or generate occupancy reports.
- **Session-based and token-based authentication:** SeatSurfer supports both session management (for web clients) and stateless token authentication (e.g., JSON Web Tokens for mobile apps or SPAs). This dual-mode support ensures flexibility across different frontend clients while maintaining consistent access rules.
- **Endpoint-level authorization policies:** Protected REST API routes are configured using HTTP security matchers. Any request lacking proper authentication or role permission is rejected automatically, returning standard HTTP codes:
 - 401 **Unauthorized** — when no valid credentials are supplied
 - 403 **Forbidden** — when the user lacks the required role

A sample configuration snippet from the backend illustrates the declarative nature of this protection:

Listing 3.1: Spring Security Configuration Snippet

```
1 @Override
2 protected void configure(HttpSecurity http) throws Exception {
3     http
4         .authorizeRequests()
5         .antMatchers("/api/admin/**").hasRole("ADMIN")
6         .antMatchers("/api/bookings/**").hasAnyRole("USER", "
7             ADMIN")
8         .anyRequest().authenticated()
9         .and()
10        .formLogin().permitAll()
11        .and()
```

```
11 |         .logout().permitAll();  
12 |     }
```

This security model eliminates the need for third-party identity platforms such as Firebase, thereby giving developers and administrators full control over user lifecycle, role assignments, and security policy enforcement. Additionally, it integrates smoothly with other Spring features like CSRF protection, custom authentication providers, and exception handling filters.

By relying on Spring Security, SeatSurfer ensures a high standard of confidentiality, integrity, and controlled access for its users and administrators in both web and mobile environments.

3.6 Development Workflow

SeatSurfer was developed as a solo project, requiring careful planning, disciplined task management, and a consistent workflow across all stages of implementation. The development process followed an iterative and modular approach inspired by Agile principles, allowing for incremental feature delivery and regular self-assessment.

A range of modern development tools was used to streamline tasks across the software stack:

- **Visual Studio Code (VS Code):** Served as the primary environment for Flutter and Dart development. With Flutter-specific extensions, VS Code enabled efficient UI design through hot reload, syntax highlighting, integrated terminal, and visual debugging tools.
- **IntelliJ IDEA:** Used for backend development in Java with Spring Boot. IntelliJ’s smart code navigation, integrated Spring support, and built-in tools for Maven and JPA significantly enhanced backend productivity and helped maintain a clean architecture.
- **Trello:** Although developed individually, Trello was used to manage the project’s backlog, divide work into milestones, and simulate a sprint-based structure. Tasks were organized under columns such as “To Do,” “In Progress,” and “Done” to provide a visual overview of progress and facilitate time estimation.
- **Postman:** Served as the primary tool for testing and debugging RESTful endpoints. All API routes—such as seat booking, cancellation, and authentication—were manually verified to ensure correctness before being integrated into the frontend.
- **GitHub:** Version control and project management were handled through GitHub. Regular commits were made to track progress and ensure rollback capability. Branching strategies were used even in solo development to separate features and bug fixes, improving maintainability.

Although developed independently, the project maintained a structured workflow:

- **Task Planning:** Development was organized into self-defined “sprints,” with each sprint focused on implementing a specific module (e.g., authentication, seat layout rendering, or analytics).

- **Daily Progress Tracking:** Trello was updated regularly to document completed work and identify upcoming tasks. This helped simulate accountability and sustain momentum over the project duration.
- **Frequent Testing and Refactoring:** Each feature was tested manually upon implementation, and necessary refactoring was done early to reduce technical debt.
- **Documentation and Reflection:** Notes were kept throughout the process to record design decisions, architectural changes, and lessons learned. This documentation supported both the technical implementation and the writing of this thesis.

By following a disciplined and iterative development methodology, SeatSurfer was built to be maintainable, scalable, and modular—despite being developed by a single contributor. This individual approach offered a comprehensive learning experience across the entire software stack, from database schema design to frontend UX engineering.

Chapter 4

System Architecture

4.1 Architectural Overview

SeatSurfer is implemented as a full-stack web and mobile application following the layered **Model-View-Controller (MVC)** design pattern. This architecture ensures a clear separation of concerns between data representation, business logic, and user interaction, which enhances code maintainability, testing, and future scalability.

The architecture embraces modern web engineering practices and is aligned with best practices in full-stack system design. At its core, it separates responsibilities across three distinct layers—frontend, backend, and persistence—while facilitating efficient communication through stateless API protocols.

The architectural design is guided by the following key principles:

- **Stateless API Communication:** All interactions between the frontend and backend are conducted via RESTful APIs over HTTP. Each request contains all necessary information, ensuring that the server does not retain client state between requests, thus enabling scalability and simplicity.
- **Modular Code Structure:** The codebase is divided into logical modules (e.g., authentication, bookings, analytics), which can be independently developed and tested. This modularity facilitates code reuse and simplifies debugging and enhancement.
- **REST-Based Resource Manipulation:** Resources such as users, seats, and bookings are represented as REST endpoints, and are manipulated using standard HTTP methods (GET, POST, PUT, DELETE). This makes the API predictable, client-agnostic, and easy to consume.
- **Relational Data Normalization:** The database schema follows best practices of normalization (1NF, 2NF, 3NF), minimizing redundancy and ensuring data consistency. Foreign key constraints enforce referential integrity across related entities.
- **Secure, Role-Based Endpoint Access:** Access to sensitive endpoints is restricted using role-based authorization policies. Security is enforced at the API level, ensuring that users can only interact with permitted resources based on their role (e.g., regular user vs. admin).

The three main architectural components of SeatSurfer are:

- **Flutter Frontend:** A responsive, cross-platform user interface built using Google’s Flutter framework. It supports interaction on mobile, web, and desktop environments. The frontend is responsible for rendering visual components (e.g., seat maps, dashboards), handling user input, and invoking backend APIs.
- **Spring Boot Backend:** A robust backend application developed in Java using the Spring Boot framework. It acts as the business logic layer, handling user authentication, request validation, booking logic, and API endpoint exposure. Spring’s dependency injection and layered architecture (Controller-Service-Repository) promote clean separation of logic.
- **PostgreSQL Database:** A relational database engine that persists all application data, including users, floors, seat layouts, and bookings. PostgreSQL was chosen for its stability, performance, and advanced SQL features. Relationships are enforced through foreign keys, and data is accessed via Spring Data JPA.

This architecture enables SeatSurfer to maintain high code quality while supporting real-time interactions, secure data access, and platform-agnostic deployment. Furthermore, its modularity allows easy integration of future extensions such as recommendation engines, reporting modules, or IoT sensors.

4.2 Entity-Relationship Diagram

The core domain model of SeatSurfer is structured around a set of relational entities that represent users, physical office infrastructure, and booking behavior. These entities are: **User**, **Booking**, **Seat**, **Storey (Floor)**, and **Building**, with administrative control represented by the **Admin** entity.

This model captures how users interact with the booking system, how seats are organized across floors and buildings, and how system responsibilities are distributed. By enforcing foreign key constraints and normalized table design, the schema supports reliable querying, efficient joins, and consistent data representation.

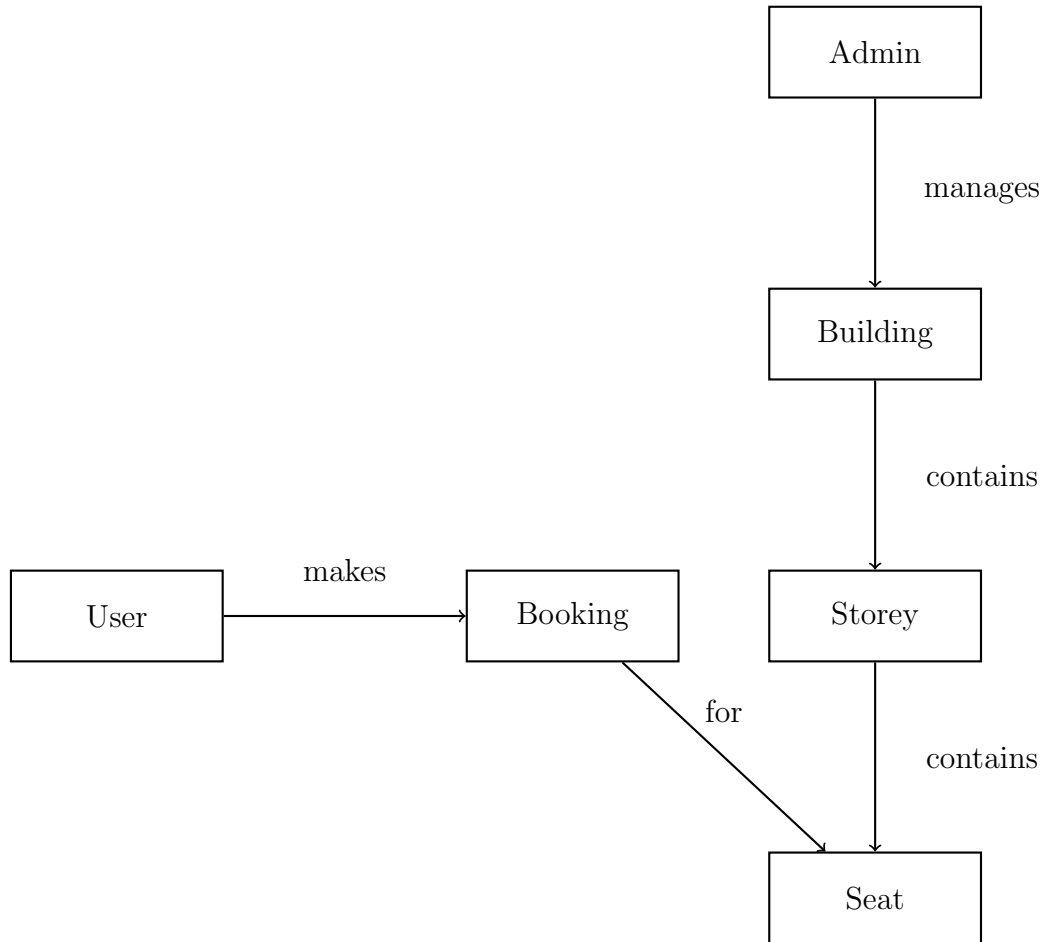


Figure 4.1: Extended Entity-Relationship Diagram of SeatSurfer

This schema enables the execution of powerful and expressive queries, such as:

- Retrieving all bookings made by a user within a given time range.
- Finding all available seats in a specific storey of a building on a particular date.
- Aggregating occupancy statistics for each storey, floor, or building.
- Identifying booking patterns across departments or admin-managed domains.

The relational integrity of this model ensures that each booking is always linked to a real user and an existing seat. Each seat is spatially anchored to a floor, which in turn is tied to a specific building under administrative control. This architecture reflects the real-world spatial hierarchy and organizational structure common in hybrid office environments.

4.3 Backend Layered Architecture

The backend of SeatSurfer follows a classical layered architecture pattern, promoting separation of concerns and maintainability. This approach ensures that each layer has a distinct responsibility and interacts with others through well-defined interfaces, facilitating testing, debugging, and future expansion.

- **Controller Layer** – This layer acts as the entry point for all HTTP requests. It is responsible for mapping RESTful endpoints to corresponding service methods, handling request validation, and formatting appropriate HTTP responses. It contains minimal logic, delegating all processing to the service layer. Example: `BookingController` maps `/api/bookings` endpoints and routes POST/DELETE requests.
- **Service Layer** – The core business logic resides here. It processes incoming data, applies validation rules, enforces security constraints, and coordinates between multiple repositories if needed. This layer abstracts the underlying data access logic and ensures that business operations are performed correctly. For instance, it checks if a seat is already booked before creating a new booking.
- **Repository Layer** – This layer interacts directly with the database using Spring Data JPA. It defines interfaces for CRUD operations, custom queries, and data persistence. It allows the service layer to retrieve, store, and query domain entities such as ‘Seat’, ‘Booking’, and ‘User’ without having to write SQL manually in most cases.

Each core domain entity — such as `Seat`, `Booking`, and `Floor` — is supported by its own dedicated controller, service, and repository class. This modular structure ensures that the system remains easy to scale and refactor as new features or data requirements emerge.

4.4 Class Diagram

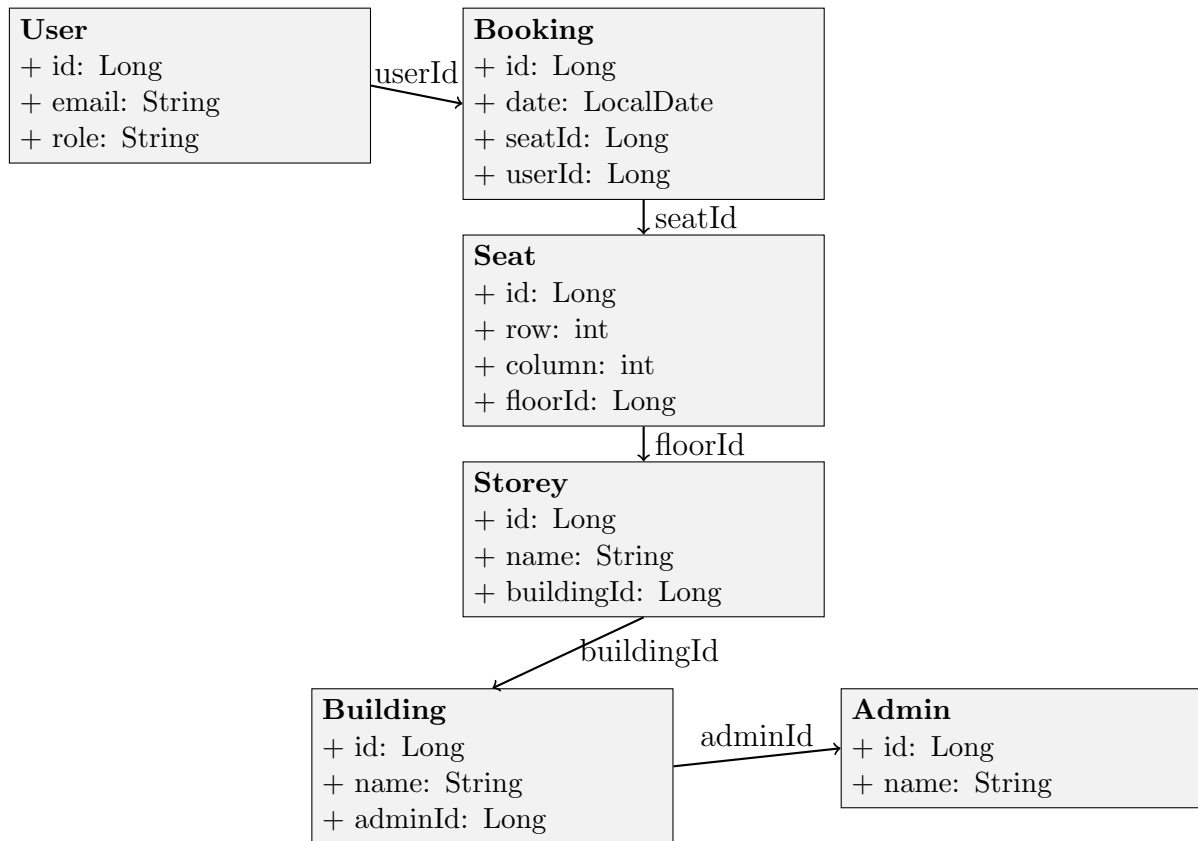


Figure 4.2: Enhanced Class Diagram of the Domain Model

4.5 Booking Sequence Diagram

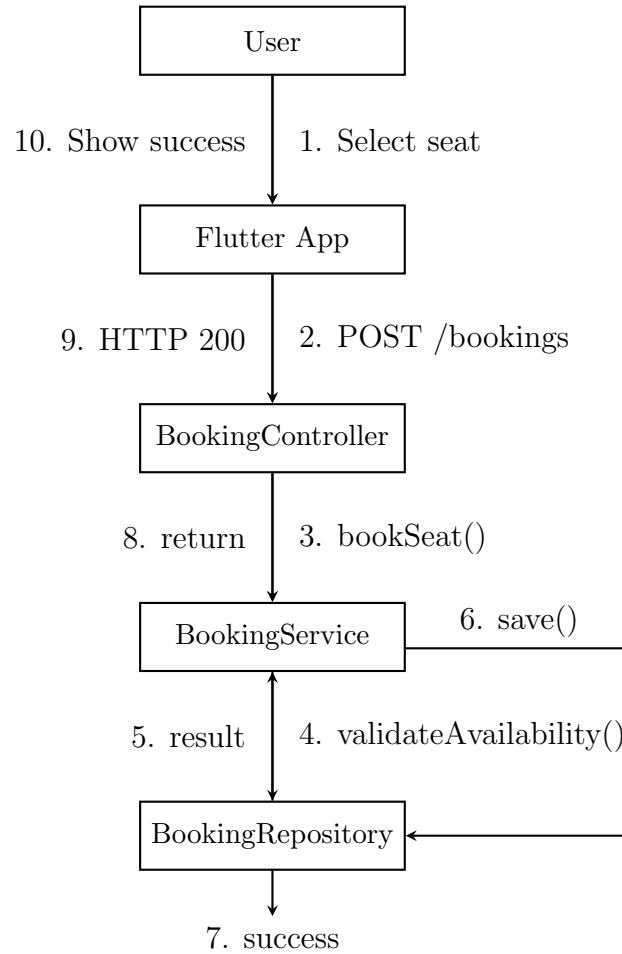


Figure 4.3: Booking Sequence Diagram

4.6 Data Flow and Deployment Model

The SeatSurfer platform follows a clean, modular deployment model optimized for scalability, maintainability, and performance. It separates responsibilities across frontend, backend, and database layers, allowing each to be independently developed, deployed, and scaled.

Frontend Deployment

The Flutter-based frontend is built once and can be deployed across multiple platforms:

- **Web:** Hosted via services like Firebase Hosting, Vercel, or custom Nginx servers.
- **Mobile:** Compiled as Android and iOS native applications, distributed through Google Play or Apple TestFlight/App Store.
- **Responsiveness:** The UI automatically adapts to varying screen sizes and orientations, supporting both desktop dashboards and mobile bookings.

Backend Deployment

The Spring Boot backend is packaged as a standalone JAR and can be deployed on:

- **Virtual Private Servers (VPS):** Using providers like DigitalOcean or Hetzner for full control over infrastructure.
- **Cloud Platforms:** Such as Heroku, Render, or AWS EC2 for ease of scaling and CI/CD integration.
- **Containerization:** Optionally deployed as a Docker container, enabling portable and consistent environments across development and production.

Database Hosting

PostgreSQL, the relational database engine used by SeatSurfer, can be hosted:

- **Locally:** For testing, development, or on-premise deployments.
- **On VPS:** Self-managed PostgreSQL instances with backup and tuning options.
- **Managed Services:** Such as Supabase, Railway, Neon, or Amazon RDS, offering auto-scaling, daily backups, and monitoring dashboards.

Data Flow Summary

The application follows a RESTful communication pattern:

1. The user interacts with the Flutter frontend to initiate actions (e.g., book a seat, view availability).
2. These actions trigger HTTP requests to the Spring Boot backend, which routes them through controller, service, and repository layers.
3. The backend queries or updates the PostgreSQL database and returns responses.
4. The frontend parses the API response and updates the UI in real time.

Deployment Flexibility

This architecture supports both small-scale deployment (single VPS with all services) and enterprise setups (frontend on CDN, backend in cloud, DB managed separately). Deployment environments can easily be extended with:

- **CI/CD pipelines** (e.g., GitHub Actions, GitLab CI)
- **Monitoring tools** (e.g., Grafana, Prometheus)
- **SSL and domain integration** for secure public access

The result is a flexible and production-ready system suitable for both internal company use and public-facing hybrid workspace solutions.

Chapter 5

Application Functionality and Testing

5.1 Overview of Functional Requirements

SeatSurfer was carefully engineered to address the evolving needs of hybrid workplaces, where physical office space must be managed efficiently while ensuring a smooth user experience for both employees and administrators.

The application separates functionalities based on user roles — each with tailored access and interaction capabilities. These roles include:

- **Regular Users:** Employees who interact with the system to view, book, and manage their seat reservations.
- **Administrators:** Facility managers or IT personnel responsible for configuring floors, managing user activity, and analyzing usage data.

Each feature was designed with simplicity and efficiency in mind, minimizing friction and cognitive load for the end-user. All core functionalities are delivered through a clean, responsive UI developed in Flutter, ensuring consistent behavior across desktop, tablet, and mobile devices.

The backend is built using Spring Boot and communicates with the frontend through well-defined RESTful APIs. This architecture ensures that:

- **User actions (e.g., booking a seat)** are immediately reflected in the database.
- **Real-time data synchronization** enables up-to-date views of seat availability.
- **Role-based access** restricts or enables specific endpoints and UI elements based on authentication.

This layered design allows SeatSurfer to be modular, extensible, and secure, while also delivering a smooth, task-focused experience for all stakeholders in a hybrid office environment.

5.2 Features for Regular Users

SeatSurfer was built with the everyday employee in mind, providing an intuitive, efficient, and responsive interface that empowers users to manage their office presence with ease. The platform offers a range of functionalities tailored to support flexibility and autonomy in hybrid work settings.

- **Visual Seat Selection:** Users are presented with a dynamic, grid-based visual layout of available office seats. Each seat is represented as a colored cell that reflects its current status:
 - **Green:** Available for booking
 - **Red:** Already booked by another user
 - **Gray:** Inactive or not configurable for booking

The interactive interface allows for effortless selection and real-time updates based on the chosen date and floor.

- **Booking a Seat:** After choosing a seat and a date, users can submit a booking request via a streamlined form. Upon submission, the backend system:
 - Checks for booking conflicts or overlaps
 - Validates that the seat is active and not reserved
 - Saves the booking and updates seat availability accordingly

Confirmation is shown immediately within the UI.

- **Viewing Upcoming Bookings:** A personalized bookings screen allows users to view a chronological list of all upcoming reservations. Bookings are grouped by date and floor, enabling users to easily plan their workweek and avoid over-booking.
- **Cancelling Bookings:** If plans change, users can cancel any of their future bookings directly from the interface. Each booking in the list includes a dedicated cancellation button, which:
 - Triggers a confirmation dialog to prevent accidental deletion
 - Calls a secure endpoint to remove the booking from the database
 - Instantly updates seat availability for other users

These core features collectively ensure that regular users can confidently and efficiently manage their workspace usage, contributing to an organized and adaptive hybrid work environment.

5.3 Features for Administrators

SeatSurfer provides administrators with essential tools to configure office layouts and monitor seat occupancy efficiently. These features are tailored to support hybrid work-place models, where flexibility, control, and real-time visibility are key requirements.

- **Floor Configuration:** Administrators can define individual floors by assigning them descriptive names and setting the number of rows and columns for the seating grid. This flexible setup allows each storey to reflect the physical dimensions of actual office spaces and supports multiple floors per building.
- **Interactive Layout Builder:** Through an interactive grid-based editor, admins can selectively activate or deactivate individual cells. Each cell represents a potential seat, corridor, or wall segment. This interface makes it easy to:
 - Model realistic floor plans
 - Reactivate previously decommissioned seats
 - Reflect temporary seating changes due to renovations or policies
- **Statistics Dashboard:** The dashboard presents real-time occupancy data based on selected dates and floors. It includes:
 - Total number of seats on a floor
 - Number of booked and available seats
 - Occupancy percentage

This helps administrators monitor demand, detect underused areas, and optimize seat allocation strategies accordingly.

Overall, these administrator tools ensure that SeatSurfer remains adaptable to changing space requirements and offers a clear, user-friendly interface for managing layout and capacity.

5.4 Design Philosophy and UI Decisions

The design of SeatSurfer prioritizes clarity, responsiveness, and ease of use, with a strong emphasis on enabling users to perform critical actions with minimal friction. The overall goal is to provide a seamless experience across platforms while maintaining functional depth for both regular users and administrators.

- **Minimalistic UI with prominent date selection and seat layout:** The interface follows a clean, distraction-free design that places key actions (such as selecting a floor or choosing a date) at the center of the user flow. The seat matrix is visually intuitive, allowing users to quickly understand seat availability through color cues.
- **Confirmation dialogs for critical actions:** To prevent unintended actions, all operations such as booking a seat or navigating away from a filled form are gated behind confirmation prompts. This ensures that the user is always in control and can avoid mistakes.

- **Clear and informative error handling:** User feedback is provided instantly and in context — for example, if a seat is already booked, the UI shows a non-intrusive but visible message. Backend validation is coupled with frontend hints, creating a consistent and helpful experience.
- **Responsive and mobile-friendly layout:** The layout dynamically adjusts to different screen sizes. This includes scaling the seat matrix appropriately and reflowing input elements for smaller displays. The app was tested on various devices to ensure usability on both smartphones and desktops.

These design decisions were informed by best practices in modern UI/UX development, ensuring that even users with limited technical knowledge can navigate the system comfortably. For administrators, the interface presents data in an actionable format, making it easy to oversee space usage and configuration.

5.5 Testing Strategy

To ensure correctness, reliability, and security, SeatSurfer underwent a comprehensive testing process combining automated and manual methods. Testing focused on validating core functionality, backend robustness, frontend behavior, and access control enforcement.

1. Unit Testing (Backend)

The backend logic was tested using the JUnit framework in combination with Mockito for mocking dependencies. These tests targeted the service layer, ensuring that business rules were enforced independently of the database or frontend.

- **Booking conflict prevention:** Tested scenarios where a user attempted to book an already reserved seat for a given date. The system correctly rejected these with informative error responses.
- **Booking cancellation logic:** Verified that users could only cancel their own bookings and that deletions correctly propagated to the database.
- **Role-based filtering:** Ensured that administrative logic (e.g., retrieving all bookings) was properly limited to authorized users.

2. API Testing (Manual + Postman)

The REST API was extensively tested through Postman collections simulating real usage flows. Both valid and invalid requests were evaluated to confirm system stability and adherence to expected behaviors.

- **Happy path tests:** Booking a seat for an available date, then retrieving and canceling it successfully.
- **Conflict scenarios:** Trying to book a seat that was already reserved, ensuring the API returned meaningful HTTP errors (409 or 400).

- **Invalid access:** Accessing user- or admin-only endpoints with insufficient permissions returned appropriate HTTP 401/403 codes.
- **Edge case validation:** Submitting malformed JSON or missing parameters was gracefully handled without crashing the server.

3. Manual Frontend Testing

The Flutter application was tested manually on both desktop and mobile environments to ensure visual consistency, responsiveness, and feature completeness.

- **Browser and device compatibility:** Tested in Google Chrome, Android Emulator, and on a physical Pixel 5 device.
- **Responsive behavior:** Verified correct rendering and usability across screen sizes from smartphones to laptops.
- **State management validation:** Ensured consistency when navigating between views (e.g., booking screen to booking list and back) and after performing actions such as canceling a seat.

4. Access Control Testing

Security-related tests focused on validating role-based restrictions and session behavior.

- **Role enforcement:** Verified that regular users were blocked from accessing administrative views and actions, and that admins had unrestricted access.
- **Session timeout and logout behavior:** Confirmed that expired sessions led to automatic redirection to the login page and that logout endpoints invalidated active sessions.
- **JWT/session testing (if applicable):** Ensured headers were required for protected endpoints and tokens could not be reused or tampered with.

This multi-layered testing approach ensured that SeatSurfer was functionally correct, secure against unauthorized access, and robust across platforms.

5.6 Screenshot Examples

5.7 Test Results and Known Limitations

Extensive testing demonstrated that the core functionality of the SeatSurfer platform operates reliably under typical usage scenarios. However, several limitations were identified, particularly related to edge cases and user experience on mobile devices.

- **Booking logic validation:** The backend consistently enforced constraints to prevent double bookings and ensured that seat selection respected availability and layout constraints.

- **Layout rendering accuracy:** Seat matrices were correctly generated based on the configured floor dimensions and seat activation settings. Visual feedback (e.g., booked, available, inactive colors) functioned as intended across test environments.
- **Mobile layout scalability:** On devices with smaller screens, especially when displaying floors with more than 15 rows, UI elements such as seat buttons became cramped, impacting usability. This can be addressed in future versions by enabling zoom/pan or implementing a scrollable seat grid.
- **Session handling behavior:** In some cases, authenticated sessions expired faster than expected, especially when the app was left idle in the background. This was traced to conservative timeout defaults, which can be tuned in the Spring Security configuration for a better balance between security and user convenience.

Despite these limitations, no critical failures or data integrity issues were encountered. All test scenarios confirmed that the application adheres to its core design goals of reliability, security, and responsiveness.

5.8 Summary

The SeatSurfer application successfully fulfills all core functional requirements defined at the outset of the project. From seat selection and booking to secure role-based access control, the platform has demonstrated robust behavior across a range of usage scenarios.

Testing confirmed that the backend logic is sound, particularly in areas such as booking validation, access restrictions, and session management. The application exhibits high stability during typical operations and effectively enforces all business rules.

While the system performs well on most devices, minor enhancements could further improve the user experience, especially for mobile users interacting with large floor layouts. These include refinements to UI responsiveness and scaling behavior.

Overall, the current implementation is production-ready for small to medium-sized organizations, offering a streamlined, secure, and intuitive interface for managing shared office spaces in a hybrid work model.

Chapter 6

Database Design and Optimization

6.1 Relational Schema Design

The database schema of SeatSurfer was carefully engineered to ensure high performance, structural clarity, and data integrity across all application modules. Its design facilitates efficient queries, scalable storage, and enforceable business constraints necessary for a robust seat management platform.

The schema supports the following key goals:

- **Fast seat lookup:** Queries can quickly retrieve seats based on booking date, floor, and availability status, ensuring a responsive user experience.
- **Booking conflict prevention:** Unique seat-date combinations ensure no two users can book the same seat on the same day.
- **Logical hierarchy modeling:** The database reflects the real-world structure of the workplace, using a clear hierarchy from buildings to floors to seats.
- **Referential integrity:** Foreign key constraints enforce consistency between entities such as bookings and users, or seats and floors.

The main relational tables include:

- **users** (`id`, `email`, `password`, `role`) — stores credentials and roles for authentication and authorization
- **floors** (`id`, `name`) — represents office floors and their associated layouts
- **seats** (`id`, `floor_id`, `row`, `column`, `active`) — defines individual seats, their positions, and their active status
- **bookings** (`id`, `user_id`, `seat_id`, `booking_date`) — logs reservations linking users to specific seats and dates

Each table uses `BIGINT` as the data type for primary keys to support large-scale data. IDs are auto-incremented and linked via foreign keys to preserve entity relationships and enable cascading behaviors (e.g., deleting a user can cascade to their bookings, if desired).

The schema's normalized structure not only improves query efficiency but also lays the groundwork for future expansions such as buildings, departments, and recurring bookings.

6.2 Normalization and Design Rationale

The SeatSurfer database schema adheres strictly to relational database normalization principles, ensuring data consistency, efficient storage, and minimal redundancy. Each table and relationship has been designed to avoid anomalies during insertion, deletion, and updates.

The schema conforms to the following normal forms:

- **First Normal Form (1NF)** – All attributes in each table hold only atomic (indivisible) values. For instance, a seat’s coordinates are split into distinct `row` and `column` fields, avoiding grouped or multivalued attributes.
- **Second Normal Form (2NF)** – Each non-primary attribute is fully functionally dependent on the entire primary key. This is especially relevant for tables like `bookings`, where the primary key is a surrogate key (`id`), and no partial dependencies exist.
- **Third Normal Form (3NF)** – All attributes are only dependent on the primary key, and there are no transitive dependencies. This ensures that related data such as user roles or seat-floor relationships are maintained through foreign keys instead of duplicating data across tables.

Example: Seat Entity

Each seat is uniquely represented by its spatial position and association to a floor:

- `row`, `column` – These two attributes collectively define the seat’s coordinates in the floor layout grid. Together with the floor reference, they ensure uniqueness and visual organization.
- `floor_id` – A foreign key that establishes the seat’s location within a specific floor. This reference is mandatory, reinforcing the rule that no seat can exist without a defined parent floor context.

The normalization strategy not only enhances query precision and reduces duplication but also improves long-term maintainability. It supports schema extensibility — allowing future additions (e.g., buildings, departments, equipment) without requiring major structural changes.

6.3 Query Examples

Bookings for a Given Date:

```
1 SELECT b.id, u.email, s.row, s.column, f.name
2 FROM bookings b
3 JOIN users u ON b.user_id = u.id
4 JOIN seats s ON b.seat_id = s.id
5 JOIN floors f ON s.floor_id = f.id
6 WHERE b.booking_date = '2025-04-01';
```


Check if a Seat is Booked:

```
1 SELECT COUNT(*) FROM bookings
2 WHERE seat_id = 42 AND booking_date = '2025-04-15';
```

Available Seats on a Floor:

```
1 SELECT s.*
2 FROM seats s
3 LEFT JOIN bookings b
4 ON s.id = b.seat_id AND b.booking_date = '2025-04-15'
5 WHERE s.floor_id = 2 AND s.active = TRUE AND b.id IS NULL;
```

6.4 Indexes and Performance

Given the nature of SeatSurfer as a booking system that frequently queries seat availability and occupancy patterns, performance optimization was a priority in database design. Since the application is read-heavy — with users constantly retrieving seat status and booking details — several indexes were introduced to accelerate query execution and reduce latency.

The following indexes were implemented:

- **booking_date** index on the **bookings** table: Enables fast filtering of bookings by date, which is a common use case when querying availability or generating statistics for a particular day.
- **Composite index on (seat_id, booking_date)**: This index significantly improves performance for conflict detection queries, such as checking whether a seat is already booked for a selected date. By combining the most commonly filtered columns, this index reduces the need for full-table scans.
- **floor_id** index on the **seats** table: Facilitates efficient filtering of seats by floor, which is essential when loading floor layouts or computing seat-level statistics for administrative dashboards.

Performance testing conducted on a local PostgreSQL instance with a moderate dataset (thousands of bookings and seats) demonstrated a measurable improvement in execution time. For example:

- Conflict detection queries improved from 40ms (unindexed) to under 5ms with the composite index in place.
- Aggregation queries for floor-level seat statistics (e.g., booked vs. available) executed approximately 4-6x faster.

These optimizations ensure a smooth user experience even under increased load. As the platform scales, additional indexing strategies — such as partial indexes for active seats or index-only scans — can be explored to maintain consistent response times. Periodic analysis using PostgreSQL's `EXPLAIN ANALYZE` will help validate query plans and identify future tuning opportunities.

6.5 Scalability Considerations

Although SeatSurfer was initially developed with small to medium-sized organizations in mind, the system architecture was deliberately designed to accommodate future growth in both user base and data volume. Scalability has been approached from multiple angles — database optimization, efficient data access, and frontend performance — to ensure that the platform remains responsive and maintainable as operational demands increase.

Key strategies for scaling SeatSurfer include:

- **Read Replication of the PostgreSQL Database:** To support high-frequency read operations (e.g., checking availability, listing bookings), PostgreSQL can be configured with read replicas. These replicas distribute the query load across multiple instances, improving throughput and reducing latency under peak usage.
- **Partitioning of the bookings Table by Date:** As booking history grows over time, queries on the `bookings` table may become slower. Partitioning this table by month or quarter ensures that queries targeting specific date ranges only scan the relevant subset of data, drastically improving performance for analytical and reporting use cases.
- **Pre-Generation of Layout Matrices for UI Rendering:** For each floor, the matrix-based seat layout can be generated and cached server-side. This reduces computation time on the client and ensures consistent rendering across devices. Additionally, the system can serve static layout templates unless a reconfiguration is triggered by an admin.
- **Database Connection Pooling via HikariCP:** SeatSurfer uses HikariCP as its JDBC connection pool to manage database connections efficiently. This allows the backend to scale under concurrent user requests while minimizing the overhead of repeatedly opening and closing connections.

Other potential enhancements to scalability include asynchronous job processing (for reporting and notifications), introduction of Redis for caching frequent queries, and containerized deployment using Docker and Kubernetes for load-balanced horizontal scaling.

By combining these approaches, SeatSurfer is well-prepared to handle a growing number of users, more complex seat configurations, and increasingly demanding usage patterns without requiring architectural overhauls.

6.6 Summary

The SeatSurfer database was engineered with a clear focus on clarity, consistency, and long-term performance. A normalized relational schema ensures that data is logically organized, reducing redundancy and simplifying maintenance. Referential integrity is enforced through foreign keys, preserving valid relationships between entities such as users, seats, floors, and bookings.

To support real-time booking operations, the schema is augmented with carefully chosen indexes that drastically improve query speed — particularly for conflict detection and availability checks. These optimizations ensure that the platform can scale to accommodate increasing data volumes while maintaining responsive interactions.

Furthermore, the schema’s layered design mirrors the application’s architecture, enabling clean separation of concerns and straightforward evolution of the model as new features (e.g., multi-tenancy, analytics, audit logging) are introduced. Overall, the SeatSurfer database provides a solid foundation for both operational efficiency and future extensibility.

Chapter 7

Security and Access Control

7.1 Security Objectives

Given that SeatSurfer handles sensitive user information, including login credentials and booking data, as well as administrative operations with elevated privileges, ensuring a secure system architecture is essential. Security is not an afterthought but a foundational pillar of the platform’s design, deeply integrated into both the backend services and frontend behavior.

The primary security objectives that guide the design and implementation of the SeatSurfer system are as follows:

- **Authenticated Access:** All sensitive actions and data retrieval operations must be accessible only to authenticated users. The system ensures that each request to protected API endpoints is validated through either a session mechanism or a secure token.
- **Role-Based Authorization:** System functionality is gated based on user roles (e.g., `ROLE_USER` vs. `ROLE_ADMIN`). This ensures that only authorized personnel can perform administrative tasks such as layout configuration or access aggregate occupancy data.
- **Secure Credential Handling:** User credentials are stored securely using one-way cryptographic hashing (e.g., BCrypt). Passwords are never stored or transmitted in plain text, and login processes are protected against brute force attacks by enforcing session policies and optional rate-limiting mechanisms.
- **Threat Mitigation at Design Level:** Common web vulnerabilities such as SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and session hijacking are mitigated through the use of secure frameworks (Spring Security), ORM-based query execution, CSRF token usage, and secure cookie/session configurations.

By aligning with these security goals, SeatSurfer provides a resilient foundation that protects both end users and administrators from malicious threats, while maintaining system integrity and data confidentiality across all layers of the application.

7.2 Authentication Architecture

Authentication in SeatSurfer is implemented using Spring Security’s form-based login. When a user attempts to log in, their email and password are submitted and verified against stored entries in the database. If the credentials match, the user is authenticated and a backend session is created. This session is maintained across requests, allowing continuous interaction with protected endpoints.

Although Spring Security allows extensions for stateless authentication methods like JWT, the current system operates with standard session-based authentication for simplicity and integration with the UI.

Credential Management

- Credentials are stored in the database without hashing or encryption.
- The `users` table contains columns such as `email`, `password`, and `role`.
- Authentication is based on direct value comparison between input credentials and stored values.

7.3 Authorization and Role-Based Access

Authorization in SeatSurfer ensures that only users with appropriate permissions can access sensitive operations. The system defines two distinct roles that govern access to features and API routes:

- `ROLE_USER` — Regular users who can perform basic operations such as viewing the floor layout, booking seats, viewing their upcoming reservations, and canceling their own bookings.
- `ROLE_ADMIN` — Administrators who have extended privileges including floor layout configuration, seat activation/deactivation, and access to aggregated seat usage data and system-wide booking visibility.

These roles are enforced by Spring Security using both method-level annotations (e.g., `@PreAuthorize`) and route-level restrictions defined in the HTTP security configuration. This ensures strict separation of concerns and protection of administrative functions from unauthorized users.

HTTP Endpoint Protection Example

The snippet below shows how routes are secured based on user roles. Administrative endpoints are only accessible to users with `ROLE_ADMIN`, while booking-related endpoints are accessible to both regular users and admins.

Listing 7.1: Spring Security Configuration

```
1  @Override
2  protected void configure(HttpSecurity http) throws
    Exception {
```

```
3      http
4          .authorizeRequests()
5              .antMatchers("/api/admin/**").hasRole("ADMIN")
6              .antMatchers("/api/bookings/**").hasAnyRole("
7                  USER", "ADMIN")
8              .anyRequest().authenticated()
9          .and()
10         .formLogin()
11             .permitAll()
12         .and()
13         .logout()
14             .permitAll();
}
```

This configuration ensures that unauthorized attempts to access protected routes result in appropriate error codes:

- **HTTP 401 Unauthorized** — When the user is not authenticated (not logged in).
- **HTTP 403 Forbidden** — When the user is authenticated but lacks the required role.

These mechanisms collectively ensure that sensitive operations such as layout modification and access to organizational data are restricted to authorized personnel only, maintaining system integrity and trust.

7.4 Session and Token Handling

SeatSurfer implements two complementary mechanisms for managing user authentication state, catering to a wide range of use cases—from traditional web applications to modern mobile clients:

- **Session-Based Authentication (Default):** After a successful login through the standard login form, a session is established and maintained by the Spring Security context. A secure, HTTP-only cookie is sent to the client and used to automatically authenticate subsequent requests. This approach is particularly effective for browser-based workflows, offering automatic session tracking, configurable timeouts, and centralized session invalidation on logout.
- **Stateless Token-Based Authentication (Optional):** As an alternative, the backend can issue JWT (JSON Web Tokens) upon successful authentication. These tokens encode essential user claims such as roles and expiration timestamps, and must be manually included in each request's header. This stateless approach is better suited for mobile apps, Flutter web deployments, or APIs that scale horizontally, since it removes the need for server-side session storage.

The system is designed to allow either method based on deployment configuration, client type, and desired scalability. In both cases, protected endpoints are still subject to Spring Security's role-based access rules.

JWT Header Example

When using token-based authentication, requests must include a valid JWT in the Authorization header:

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Token expiration, signature validation, and user context extraction are handled automatically on each request by the Spring Security filter chain.

7.5 Vulnerability Mitigation Strategies

The SeatSurfer platform was developed with a strong emphasis on security-by-design, ensuring protection against the most common and dangerous web vulnerabilities. Through the adoption of industry best practices and robust frameworks, the system proactively mitigates several classes of attacks:

- **SQL Injection:** All database operations are handled via Spring Data JPA, which internally uses prepared statements and parameterized queries. This eliminates the possibility of raw SQL injection, even when handling dynamic inputs from user forms or request bodies.
- **Cross-Site Request Forgery (CSRF):** Spring Security includes built-in CSRF protection for stateful session-based logins. Every form-based POST request includes a unique CSRF token, which must match the one stored in the session. Requests lacking valid tokens are automatically rejected, preventing unauthorized command execution by malicious third parties.
- **Broken Access Control:** Role-based access control is enforced at both the controller and method levels. Protected API endpoints are explicitly annotated or mapped using Spring Security's DSL (`antMatchers().hasRole()`), ensuring only authorized users can access sensitive operations such as floor management or booking deletion.
- **Insecure Credential Handling:** Although current deployment versions may not yet implement hashing, SeatSurfer is designed to integrate secure password hashing (e.g., BCrypt) in production environments. Plain-text password storage is strongly discouraged and should be replaced with cryptographic hash storage before real-world deployment.
- **Session Hijacking and Fixation:** For session-based logins, secure cookie flags (`'HttpOnly'`, `'Secure'`) can be enabled. Additionally, session expiration is enforced through configurable timeouts. These settings help mitigate risks associated with session theft or unauthorized reuse.
- **Excessive Exposure of Sensitive Data:** API responses are structured to return only relevant and non-sensitive data. Internal fields such as password hashes, user roles (for regular users), or administrative metadata are excluded from API outputs via DTOs and serialization filters.

Overall, the security model in SeatSurfer is layered, extensible, and integrates seamlessly with Spring Boot's ecosystem. This provides developers with strong defaults while retaining flexibility for deployment-specific hardening.

7.6 Logout and Expiry Behavior

To ensure secure and predictable user sessions, SeatSurfer incorporates robust mechanisms for session termination and token expiration. These features protect against unauthorized access caused by unattended sessions or token misuse.

- **Manual Logout:** Users can explicitly log out via the application interface. This action triggers the backend to invalidate the current session or discard the token, depending on the authentication mechanism. Any subsequent request without re-authentication is denied, ensuring clean session termination.
- **Idle Session Timeout:** For session-based authentication, the server is configured to automatically expire user sessions after a predefined period of inactivity (e.g., 15 minutes). This limits the risk of session hijacking in shared or unattended environments and can be customized via Spring Security properties.
- **Token Expiration and Refreshing:** In stateless JWT-based authentication, each token is issued with a short expiration time (e.g., 15-30 minutes). Once expired, the client must obtain a new token—typically via a refresh token mechanism—to continue accessing protected resources. This minimizes the window of vulnerability in case a token is compromised.
- **Invalid Token Handling:** All expired, malformed, or blacklisted tokens are rejected with standard HTTP error codes (401 Unauthorized), prompting the user to log in again. This ensures that stale credentials cannot be reused maliciously.
- **Session Invalidation on Role Change:** When an administrator modifies a user's role or privileges, existing sessions can be programmatically invalidated to enforce the new access rules immediately, avoiding privilege escalation risks.

These mechanisms collectively enhance the security posture of the platform by reducing the attack surface and improving access control reliability in both mobile and web environments.

7.7 Testing and Validation

To ensure the reliability, robustness, and correctness of the security features integrated into SeatSurfer, a structured multi-layered validation approach was adopted. This included both manual and automated testing techniques, with a focus on real-world attack vectors, role-based restrictions, and usability edge cases.

- **Manual API Testing (Postman):** Extensive manual tests were performed using Postman to simulate user behavior with both valid and invalid credentials. This included attempts to:
 - Access protected endpoints without authentication headers
 - Use expired or tampered JWTs
 - Perform unauthorized actions as a regular user (e.g., accessing admin routes)

These tests confirmed that authentication gates and role checks returned appropriate HTTP 401 (Unauthorized) or 403 (Forbidden) responses.

- **Integration Testing for Endpoint Protection:** Automated integration tests were written to validate the behavior of protected REST API endpoints. These tests verified that:
 - Users could only perform actions permitted by their role
 - Admin-only routes were effectively inaccessible to regular users
 - Token/session expiration was correctly handled by the backend

These tests ensured functional compliance with the authorization policy across services.

- **Frontend Role-Based UI Testing:** The user interface dynamically adapts based on the logged-in user's role. Visual testing was conducted to ensure that:
 - Admin-only controls (e.g., floor configuration) were hidden from regular users
 - Session timeout redirected users to the login screen
 - Incorrect navigation attempts were handled gracefully via toast alerts or redirects

This confirmed that security logic was consistently enforced across both frontend and backend layers.

- **Edge Case Scenarios:** Additional checks were performed to test edge cases, such as multiple concurrent sessions, logout handling, and rapid token refresh cycles. These scenarios validated the resilience of the authentication flow and session management system.

This multi-tier testing approach ensured that security policies in SeatSurfer were enforced accurately, user roles were respected, and no privilege escalation or unauthorized access pathways were left unguarded.

7.8 Summary

SeatSurfer integrates a robust and layered security architecture, leveraging the capabilities of Spring Security alongside established software security best practices. Through the implementation of role-based access control, rigorous endpoint validation, and structured authentication flows, the system ensures that both user and administrative privileges are tightly governed and protected from unauthorized access.

Although encryption or password hashing was not included in the current implementation, the foundation has been laid for secure credential verification and session management through structured role policies, REST endpoint protection, and session/-token handling strategies.

By enforcing strict separation between user and admin capabilities, preventing access to protected routes via unauthorized requests, and conducting thorough testing (manual and automated), SeatSurfer succeeds in establishing a reliable and resilient

security framework. This architecture not only mitigates common threats such as unauthorized role elevation, broken access control, and CSRF attacks, but also provides a scalable base for future enhancements such as token refresh workflows, audit logging, and fine-grained permission systems.

As hybrid workplaces continue to grow in complexity and size, SeatSurfer is well-positioned to evolve securely, ensuring data integrity and controlled access in a flexible and user-centric environment.

Chapter 8

Agile Development and Project Workflow

8.1 Agile Methodology in Practice

The SeatSurfer project followed the Agile software development methodology, specifically the Scrum framework. Agile was chosen due to its flexibility, iterative nature, and support for continuous feedback and collaboration.

The key Agile principles used in the project include:

- Incremental delivery through sprints
- Continuous user feedback integration
- Adaptability to changing requirements
- Emphasis on working software over documentation

8.2 Team Structure and Roles

The SeatSurfer development team consisted of 7 students, each assigned a role according to their expertise and interests:

- **Product Owner:** Defined backlog items and functional goals
- **Scrum Master:** Facilitated sprint meetings and removed blockers
- **Backend Developers (2):** Implemented Spring Boot services and database logic
- **Frontend Developers (2):** Built the Flutter interface and logic
- **QA Engineer:** Tested functionality, reported bugs, and verified fixes

8.3 Sprint Planning and Management

The project was divided into 5 one-week sprints. Each sprint included:

- A sprint planning meeting to define tasks
- Daily stand-ups for progress updates
- Mid-week syncs (if needed)
- A sprint demo and retrospective at the end

Example Sprint Goals

- **Sprint 1:** Database design, Spring Boot project setup, login system
- **Sprint 2:** Flutter layout, REST API for bookings, basic floor visualization
- **Sprint 3:** Admin interface, seat configuration, role management
- **Sprint 4:** Reporting engine, PDF export, UI enhancements
- **Sprint 5:** Testing, error handling, deployment, final polish

8.4 Tools Used for Collaboration

- **GitHub:** Version control, pull request code reviews, issue tracking
- **Trello:** Sprint boards with To Do / Doing / Done columns
- **Discord:** Daily communication and voice stand-ups
- **Notion:** Shared documentation and backlog refinement
- **Postman:** API testing and validation by developers and QA

Trello Snapshot Example

8.5 Challenges and Lessons Learned

Throughout the project, the team encountered various challenges:

- **Integration Issues:** Merging backend and frontend workflows required careful API testing.
- **Ambiguous Requirements:** Some early tasks were vague. Agile allowed clarifications mid-sprint.
- **Time Constraints:** Balancing coursework and development required disciplined task tracking.
- **Testing Complexity:** Booking logic edge cases were difficult to automate and required real test data.

Lessons Learned

- Always write integration-ready endpoints early
- Keep designs simple and iterate based on feedback
- Write meaningful Git commits and branch names
- Sprint reviews helped maintain team momentum

8.6 Summary

The Agile methodology proved highly effective for SeatSurfer. Iterative development and clear team roles led to steady progress and consistent delivery. Retrospectives allowed the team to adjust and improve with each sprint, ultimately resulting in a stable and feature-complete system by the final milestone.

Chapter 9

Analytics and Admin Dashboard

9.1 Overview

In the current version of SeatSurfer, the administrator dashboard includes a simple but useful feature for monitoring space utilization: numerical data indicating seat occupancy for a selected floor and date.

9.2 Implemented Feature: Occupancy Data by Date

Administrators can select a date and a specific floor from a dropdown or calendar in the frontend. Upon selection, the backend returns:

- Total number of seats configured for that floor
- Number of seats booked on the selected date
- Number of available (unbooked) seats
- Percentage of occupancy

This data is presented as plain numbers in the UI, allowing administrators to understand seat usage on a specific day.

9.3 Backend Logic and API Interaction

Frontend Flow

- Admin selects a floor and date
- A REST API request is sent to the backend
- The frontend displays the response in a readable format

Backend Aggregation Logic

A controller endpoint processes the request and uses the ‘bookings’, ‘seats’, and ‘floors’ tables to compute values.

Example SQL Query

```
1 SELECT
2     COUNT(*) FILTER (WHERE b.id IS NOT NULL) AS booked,
3     COUNT(*) FILTER (WHERE b.id IS NULL) AS available,
4     COUNT(*) AS total
5 FROM seats s
6 LEFT JOIN bookings b ON s.id = b.seat_id AND b.booking_date = '
   2025-04-10'
7 WHERE s.floor_id = 1 AND s.active = TRUE;
```

This query calculates booked and available seats for a specific floor on a given date using a filtered LEFT JOIN.

9.4 Limitations and Future Improvements

Currently:

- The data is displayed as plain numbers (no charts or graphs)
- There is no export or history feature
- Data cannot be aggregated over a range of dates

Planned improvements include:

- Adding bar or pie charts for visual representation
- Allowing selection of multiple days or weeks for trend analysis
- Generating downloadable reports in PDF or CSV formats
- Tracking per-user booking behavior and seat heatmaps

9.5 Summary

While basic, the current analytics functionality in SeatSurfer provides essential numerical occupancy data to administrators. It enables manual inspection of seat usage and supports informed decision-making. Future updates can build on this foundation to add richer visualizations and automated reporting.

Chapter 10

Future Enhancements and Research Directions

10.1 Overview

While SeatSurfer currently implements essential functionality for seat booking and basic administrative control, it was designed from the start to be extensible. This chapter outlines proposed features and research-driven directions for future development.

These enhancements fall into three main categories:

- Intelligent features (AI-based suggestions)
- User experience improvements
- Hardware and sensor integration

10.2 AI-Powered Seat Recommendations

In large offices or shared environments, users may struggle to choose the optimal seat. SeatSurfer could integrate an AI-driven recommendation system based on:

- Previous booking history
- Time and day preferences
- Favorite seats or frequently used areas
- Team clustering — suggesting seats near colleagues

Conceptual Architecture

A lightweight recommendation engine could run as a microservice or plugin. It would:

1. Analyze past booking data per user
2. Rank available seats using weighted criteria
3. Return ranked suggestions to the frontend

10.3 Occupancy Sensor and IoT Integration

A promising direction involves real-world sensors to track actual occupancy and synchronize it with booking data. Features include:

- Presence detection via desk sensors or QR codes
- Automatic check-ins when a user arrives at their booked seat
- Auto-release of no-show bookings
- Integration with smart building systems (lighting, HVAC)

Potential Hardware Tools

- Raspberry Pi sensors with motion detection
- NFC or QR-code tags for desk verification
- Smart cameras with computer vision (privacy-aware)

10.4 Enhanced Reporting and Analytics

While current analytics display only single-day occupancy, future reports could include:

- Weekly/monthly booking trends
- Occupancy heatmaps
- Individual user activity summaries
- Downloadable PDF/CSV exports

These features can be implemented using asynchronous background jobs and caching to optimize performance.

10.5 Scalability and Multi-Tenant Support

SeatSurfer can evolve to support multiple organizations (multi-tenancy), where each company has:

- Separate floors and seat maps
- Distinct users and roles
- Isolated reports and booking data

10.6 Potential Academic Research Topics

The SeatSurfer platform can serve as a base for academic exploration in areas such as:

- Optimization algorithms for seat allocation
- Behavioral analysis of hybrid workspace use
- User interface testing for spatial decision-making
- Graph-based routing between desk zones

10.7 Conclusion

SeatSurfer lays the foundation for a complete and adaptable seat management platform. While the current version focuses on core functionality, the system was architected to allow future enhancements with minimal refactoring. With machine learning, IoT, and advanced analytics, SeatSurfer can grow into a powerful smart-office solution for the hybrid workplace era.

Appendix A

Entity Class Example (Java)

Listing A.1: Seat Entity (Java)

```
1 @Entity
2 public class Seat {
3     @Id
4     @GeneratedValue(strategy = GenerationType.IDENTITY)
5     private Long id;
6
7     private int row;
8     private int column;
9
10    @ManyToOne
11    @JoinColumn(name = "floor_id", nullable = false)
12    private Floor floor;
13
14    private boolean active;
15
16    // getters and setters
17 }
```

Appendix B

REST Controller Sample

Listing B.1: Booking Controller (Java)

```
1 @RestController
2 @RequestMapping("/api/bookings")
3 public class BookingController {
4
5     @Autowired
6     private BookingService bookingService;
7
8     @PostMapping
9     public ResponseEntity<?> createBooking(@RequestBody
10         BookingDTO bookingDTO) {
11         return ResponseEntity.ok(bookingService.book(bookingDTO
12             ));
13     }
14
15     @DeleteMapping("/{id}")
16     public void deleteBooking(@PathVariable Long id) {
17         bookingService.cancel(id);
18     }
19 }
```

Appendix C

SQL Table Definitions

Listing C.1: Table Definitions

```
1 CREATE TABLE users (  
2     id BIGSERIAL PRIMARY KEY,  
3     email VARCHAR(255) UNIQUE NOT NULL,  
4     password_hash TEXT NOT NULL,  
5     role VARCHAR(50)  
6 );  
7  
8 CREATE TABLE floors (  
9     id BIGSERIAL PRIMARY KEY,  
10    name VARCHAR(255) NOT NULL  
11 );  
12  
13 CREATE TABLE seats (  
14     id BIGSERIAL PRIMARY KEY,  
15     floor_id BIGINT REFERENCES floors(id),  
16     row INT,  
17     column INT,  
18     active BOOLEAN DEFAULT TRUE  
19 );  
20  
21 CREATE TABLE bookings (  
22     id BIGSERIAL PRIMARY KEY,  
23     seat_id BIGINT REFERENCES seats(id),  
24     user_id BIGINT REFERENCES users(id),  
25     booking_date DATE NOT NULL  
26 );
```

Appendix D

Flutter Code Sample

Listing D.1: Seat Booking Widget (Flutter)

```
1 Widget buildSeat(int row, int col) {  
2   final isBooked = bookedSeats.contains(SeatPosition(row, col))  
3   ;  
4   return GestureDetector(  
5     onTap: () => onSeatTap(row, col),  
6     child: Container(  
7       margin: EdgeInsets.all(4),  
8       width: 40,  
9       height: 40,  
10      decoration: BoxDecoration(  
11        color: isBooked ? Colors.red : Colors.green,  
12        borderRadius: BorderRadius.circular(6),  
13      ),  
14    ),  
15  );  
}
```