Projektdokumentation

Terminkalender / Reservierungssystem

Modul: 223 - Multiuser-Applikationen objektorientiert realisieren
Projektname: Terminkalender
Version: 1.0
Datum: November 2025

Team:

- Rigo Erisk Reyes
- Denis Perdomo

Repository: https://github.com/EriskReyes/Reservationssystem.git

Inhaltsverzeichnis

1. Einleitung und Projektziel
2. Anforderungsanalyse
3. Systemarchitektur
4. UML-Diagramme
5. Technologien und Tools
6. Projektstruktur
7. Implementierung
8. Datenbank
9. Benutzerhandbuch
10. Testing
11. Deployment
12. Probleme und Lösungen
13. Fazit und Ausblick

1. Einleitung und Projektziel

1.1 Ausgangslage

Dieses Projekt wurde im Rahmen des Moduls 223 "Multiuser-Applikationen objektorientiert realisieren" entwickelt. In einem Gebäude eines Unternehmens sind verschiedene Räume verfügbar, die für Sitzungen und Veranstaltungen genutzt werden. Eine Webapplikation soll die Verwaltung der Räume und der Termine bzw. Reservationen unterstützen.

Das Unternehmen verfügt über **5 Besprechungsräume** (Zimmer 101 bis 105), die von Mitarbeitern für verschiedene Zwecke gebucht werden können.

1.2 Projektziel

Entwicklung einer Webanwendung zur Raumreservierung mit folgenden Hauptzielen:

- **Einfache Reservierung** von Besprechungsräumen ohne Benutzerkonto
- **Verwaltung** von Reservierungen über eindeutige Schlüssel (Private/Public Keys)
- **Automatische Überprüfung** der Raumverfügbarkeit
- **Persistente Speicherung** aller Reservierungen in einer PostgreSQL-Datenbank
- **Intuitive Benutzeroberfläche** mit responsive Design

1.3 Projektumfang

Das Projekt umfasst:

- Vollständige Webanwendung mit Spring Boot
- PostgreSQL-Datenbank
- Thymeleaf-Templates für die Benutzeroberfläche
- Docker-basiertes Deployment
- Umfassende Projektdokumentation
- UML-Diagramme (Zustands-, Klassen- und ERM-Diagramm)

2. Anforderungsanalyse

2.1 Funktionale Anforderungen

2.1.1 Reservierung erstellen

Die Anwendung bietet auf der Hauptseite (index.html) die Möglichkeit, einen Raum zu reservieren. **Kein Benutzerkonto erforderlich.**

Erforderliche Eingabefelder:

Feld	Datentyp	Format/Validierung	Pflicht
Datum	Date	TT.MM.JJJJ	Ja, muss in der Zukunft liegen
Von	Time	HH:MM	Ja
Bis	Time	HH:MM	Ja
Zimmer	Integer	101, 102, 103, 104, 105	Ja
Bemerkung	String	10-200 Zeichen	Ja, alphanumerisch
Teilnehmer	String	Vorname Nachname	Ja, nur Buchstaben [A-Z][a-z]

Validierungsregeln:

- Alle Felder sind Pflichtfelder (kein Feld darf leer bleiben)
- Datum muss in der Zukunft liegen
- Startzeit (Von) muss vor Endzeit (Bis) liegen
- Mindestdauer: 15 Minuten
- Raum muss zum gewählten Zeitpunkt verfügbar sein
- Teilnehmernamen: nur Buchstaben, Format "Vorname Nachname, Vorname Nachname"
- Trennzeichen zwischen Teilnehmern: Komma

2.1.2 Schlüsselsystem

Nach erfolgreicher Reservierung erhält der Benutzer **zwei eindeutige Schlüssel**:

- **Private Key (Privater Schlüssel)**:
  - Ermöglicht Bearbeitung der Reservierung
  - Ermöglicht Löschung der Reservierung
  - Sollte geheim gehalten werden
- **Public Key (Öffentlicher Schlüssel)**:
  - Ermöglicht nur Einsicht der Reservierung
  - Kann mit Teilnehmern geteilt werden (z.B. per E-Mail)
  - Keine Bearbeitungsrechte

2.1.3 Verfügbarkeitsprüfung

Vor dem Erstellen einer Reservierung wird automatisch geprüft:

- Ist das Zimmer zum gewünschten Zeitpunkt noch frei?
- Sind Datum und Zeitangaben sinnvoll?
- Gibt es zeitliche Überschneidungen mit anderen Reservierungen?

Während der Eingabe (Echtzeit):

- AJAX-basierte Verfügbarkeitsprüfung
- Sofortiges Feedback für den Benutzer
- Visueller Indikator (Verfügbar ✓ / Nicht verfügbar ✗)

2.1.4 Navigation und Zugriff

Auf der Startseite kann ein Schlüssel eingegeben werden:

- System erkennt automatisch den Schlüsseltyp
- **Private Key** → Weiterleitung zur Bearbeitungsseite
- **Public Key** → Weiterleitung zur Ansichtsseite (nur Leserechte)
- Ungültiger Schlüssel → Fehlermeldung

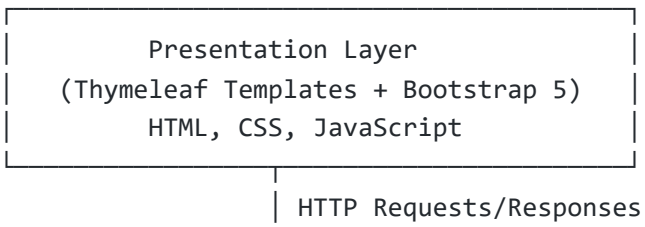
2.2 Nicht-funktionale Anforderungen

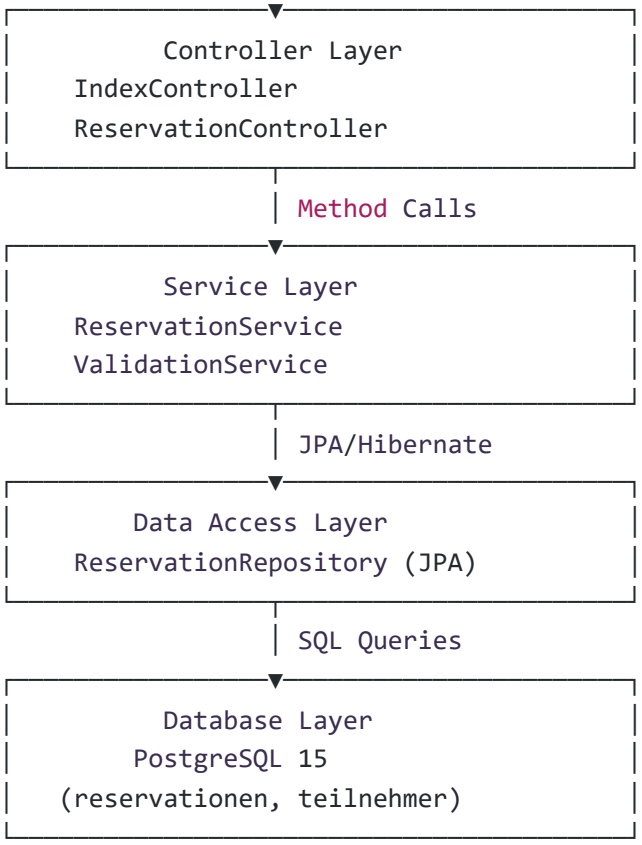
Kategorie	Anforderung
Benutzerfreundlichkeit	Intuitive Navigation, klare Fehlermeldungen, responsive Design
Performance	Verfügbarkeitsprüfung < 1 Sekunde, Seitenaufbau < 2 Sekunden
Sicherheit	UUID-basierte Schlüssel (praktisch nicht zu erraten), SQL-Injection-Schutz durch JPA
Zuverlässigkeit	Datenbankintegrität durch Constraints, Transaktionssicherheit
Wartbarkeit	Klare Code-Struktur nach MVC-Pattern, umfassende Dokumentation
Skalierbarkeit	Docker-basiertes Deployment, einfache Erweiterbarkeit

3. Systemarchitektur

3.1 Architekturüberblick

Das System folgt einer klassischen **3-Schichten-Architektur** nach dem **MVC-Pattern** (Model-View-Controller):





3.2 Schichtenverantwortlichkeiten

Presentation Layer (View):

- Darstellung der Benutzeroberfläche
- Formulareingaben
- Client-seitige Validierung (HTML5)
- AJAX-Aufrufe für Verfügbarkeitsprüfung

Controller Layer:

- Empfang von HTTP-Requests
- Request-Routing
- Aufruf der Service-Schicht
- Vorbereitung der View-Daten
- Exception-Handling

Service Layer:

- Business-Logik
- Validierung
- Transaktionsverwaltung
- Koordination zwischen Controllern und Repositories

Data Access Layer:

- Datenbankoperationen (CRUD)
- Custom Queries
- Datenbank-Abstraktionsschicht

Database Layer:

- Persistente Datenspeicherung
- Datenintegrität
- Constraints und Trigger

3.3 Technologie-Stack

```
┌─────────────────────────────────────┐
│         Frontend                    │
│  • Thymeleaf 3.1                    │
│  • Bootstrap 5.3                    │
│  • JavaScript (ES6)                 │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│         Backend                     │
│  • Java 21 (LTS)                    │
│  • Spring Boot 3.2.0                │
│  • Spring MVC                       │
│  • Spring Data JPA                  │
│  • Hibernate (JPA Provider)         │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│         Datenbank                   │
│  • PostgreSQL 15                    │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│         Build & Deployment          │
│  • Maven 3.9                        │
│  • Docker & Docker Compose          │
└─────────────────────────────────────┘
```

4. UML-Diagramme

4.1 UML-Zustandsdiagramm

Das Zustandsdiagramm zeigt alle möglichen Zustände (Seiten) der Webapplikation und die Übergänge zwischen ihnen.

Datei: `Doku/UML-Zustandsdiagramm.puml`

Hauptzustände:

1. **Index (Startseite)**
   - Einstiegspunkt der Anwendung
   - Zeigt "Neue Reservierung" Button
   - Zeigt Suchfeld für Schlüssel
2. **Reservierung Formular**
   - Eingabeformular für neue Reservierung
   - Echtzeit-Verfügbarkeitsprüfung
   - Client- und Server-Validierung
3. **Bestätigung**
   - Anzeige der generierten Schlüssel
   - Private Key und Public Key
   - Links zur Ansicht und Bearbeitung
4. **Ansicht (Public Key)**
   - Nur-Lese-Ansicht der Reservierung
   - Anzeige aller Details
   - Keine Bearbeitungsmöglichkeit
5. **Bearbeiten (Private Key)**
   - Vollständiges Bearbeitungsformular
   - Möglichkeit zum Aktualisieren
   - Möglichkeit zum Löschen

Übergänge (Signale):

- `[Neue Reservierung]` → Index → Formular
- `[Reservierung erstellen]` → Formular → Bestätigung (bei Erfolg)
- `[Validierungsfehler]` → Formular → Formular (mit Fehlermeldung)
- `[Private Key eingeben]` → Index → Bearbeiten
- `[Public Key eingeben]` → Index → Ansicht
- `[Aktualisieren]` → Bearbeiten → Bearbeiten (mit Erfolgsmeldung)

- `[Löschen]` → Bearbeiten → Index (mit Bestätigung)
- `[Zurück]` → Jede Seite → Index

Entscheidungen:

- Ist der Schlüssel ein Private Key oder Public Key?
- Ist das Formular valide?
- Ist der Raum verfügbar?
- Bestätigung beim Löschen?

4.2 ERM/ERD Datenbankdiagramm

Das Entity-Relationship-Diagramm zeigt die Datenbankstruktur mit allen Tabellen, Beziehungen und Attributen.

Datei: `Doku/ERM-Diagramm.puml`

Entitäten:

RESERVATIONEN

- **id** (PK, BIGSERIAL, AUTO_INCREMENT)
- **datum** (DATE, NOT NULL) - Reservierungsdatum
- **von_zeit** (TIME, NOT NULL) - Startzeit
- **bis_zeit** (TIME, NOT NULL) - Endzeit
- **zimmer** (INTEGER, NOT NULL, CHECK 101-105) - Raumnummer
- **bemerkung** (VARCHAR(200), NOT NULL) - Beschreibung
- **private_schluessel** (VARCHAR(36), UNIQUE, NOT NULL) - Private Key (UUID)
- **public_schluessel** (VARCHAR(36), UNIQUE, NOT NULL) - Public Key (UUID)
- **erstellt_am** (TIMESTAMP, DEFAULT NOW()) - Erstellungszeitpunkt
- **aktualisiert_am** (TIMESTAMP, DEFAULT NOW()) - Letzter Update

TEILNEHMER

- **id** (PK, BIGSERIAL, AUTO_INCREMENT)
- **vorname** (VARCHAR(50), NOT NULL) - Vorname
- **nachname** (VARCHAR(50), NOT NULL) - Nachname
- **reservation_id** (FK → reservationen.id, NOT NULL) - Foreign Key

Beziehung:

- Eine Reservierung hat viele Teilnehmer (**1:N**)
- Ein Teilnehmer gehört zu genau einer Reservierung

- **ON DELETE CASCADE**: Beim Löschen einer Reservierung werden alle Teilnehmer automatisch gelöscht
- **ON UPDATE CASCADE**: Bei Änderung der Reservierungs-ID werden Teilnehmer mitgeführt

Indizes:

- `idx_datum_zimmer` auf (datum, zimmer) - für Verfügbarkeitssuche
- `idx_private_key` auf (private_schluessel) - UNIQUE
- `idx_public_key` auf (public_schluessel) - UNIQUE
- `idx_reservation_id` auf (reservation_id) - für Joins

Constraints:

- CHECK (zimmer BETWEEN 101 AND 105)
- CHECK (von_zeit < bis_zeit)
- CHECK (LENGTH(bemerkung) BETWEEN 10 AND 200)

4.3 UML-Klassendiagramm

Das Klassendiagramm zeigt alle Controller-, Service-, Repository- und Model-Klassen mit ihren Beziehungen.

Datei: `Doku/UML-Klassendiagramm.puml`

Packages und Klassen:

Package: com.terminkalender

TerminkalenderApplication

- `+ main(String[] args): void` - Haupteinstiegspunkt

Package: com.terminkalender.controller

IndexController

- `+ index(): String` - GET / - Startseite anzeigen
- `+ schluesselSuche(String, Model): String` - POST /schluessel-suche

ReservationController

- `+ neueReservation(Model): String` - GET /reservation/neu
- `+ erstelleReservation(ReservationDTO, BindingResult, Model): String` - POST /reservation/erstellen
- `+ bestaetigung(Long, Model): String` - GET /reservation/bestaetigung/{id}
- `+ ansicht(String, Model): String` - GET /reservation/ansicht/{publicKey}

- `+ bearbeiten(String, Model): String` - GET /reservation/bearbeiten/{privateKey}
- `+ aktualisieren(String, ReservationDTO, BindingResult, Model): String` - POST /reservation/aktualisieren/{privateKey}
- `+ loeschen(String): String` - POST /reservation/loeschen/{privateKey}
- `+ pruefeVerfuegbarkeit(...): Map` - GET /reservation/verfuegbarkeit

Package: com.terminkalender.service

ReservationService

- `+ erstelleReservation(ReservationDTO): Reservation`
- `+ findeByPrivateKey(String): Optional<Reservation>`
- `+ findeByPublicKey(String): Optional<Reservation>`
- `+ aktualisiereReservation(String, ReservationDTO): Reservation`
- `+ loescheReservation(String): void`
- `+ istZimmerVerfuegbar(...): boolean`
- `- parseTeilnehmer(String, Reservation): void`

ValidationService

- `+ validiereReservation(ReservationDTO): void`
- `+ validiereZimmer(Integer): void`
- `+ validiereDatumUndZeit(LocalDate, LocalTime, LocalTime): void`
- `+ validiereBemerkung(String): void`
- `+ validiereUndParseTeilnehmer(String): List<String[]>`

Package: com.terminkalender.repository

ReservationRepository (Interface, extends JpaRepository)

- `+ findByPrivateKey(String): Optional<Reservation>`
- `+ findByPublicKey(String): Optional<Reservation>`
- `+ findOverlappingReservations(...): List<Reservation>`
- `+ findOverlappingReservationsExcludingId(...): List<Reservation>`

Package: com.terminkalender.model

Reservation (Entity)

- `- id: Long`
- `- datum: LocalDate`
- `- vonZeit: LocalTime`
- `- bisZeit: LocalTime`

- - zimmer: Integer
- - bemerkung: String
- - privateKey: String
- - publicKey: String
- - teilnehmer: List<Teilnehmer>
- - erstelltAm: LocalDateTime
- - aktualisiertAm: LocalDateTime

Teilnehmer (Entity)

- - id: Long
- - vorname: String
- - nachname: String
- - reservation: Reservation

Package: com.terminkalender.dto

ReservationDTO

- - datum: LocalDate
- - vonZeit: LocalTime
- - bisZeit: LocalTime
- - zimmer: Integer
- - bemerkung: String
- - teilnehmerListe: String

Package: com.terminkalender.config

DataInitializer (implements CommandLineRunner)

- + run(String... args): void - Erstellt Testdaten beim Start

Beziehungen:

- Controller → Service (Dependency)
- Service → Repository (Dependency)
- Repository → Model (verwaltet)
- Controller ↔ DTO (verwendet)
- Service ↔ Model (verwendet)
- Reservation ↔ Teilnehmer (1:N)

5. Technologien und Tools

5.1 Backend-Technologien

Spring Boot 3.2.0

Begründung der Wahl:

- De-facto Standard für Java-Webanwendungen
- "Convention over Configuration" - minimaler Konfigurationsaufwand
- Eingebetteter Tomcat-Server
- Auto-Konfiguration und Dependency Injection
- Große Community und umfangreiche Dokumentation
- Einfaches Dependency Management

Verwendete Module:

- **Spring MVC**: Web-Framework, Request-Handling
- **Spring Data JPA**: Vereinfachter Datenbankzugriff
- **Spring Boot Validation**: Bean-Validierung (JSR-380)
- **Spring Boot DevTools**: Hot-Reload während Entwicklung

Java 21

Vorteile:

- LTS-Version (Long Term Support)
- Verbesserte Performance gegenüber älteren Versionen
- Pattern Matching
- Records (für DTOs)
- Bessere Garbage Collection
- Text Blocks für SQL-Queries

5.2 Frontend-Technologien

Thymeleaf 3.1

Begründung:

- Server-seitiges Rendering (keine separate REST-API nötig)
- Natürliche HTML5-Templates (können im Browser geöffnet werden)
- Nahtlose Spring-Integration
- Einfachere Fehlerbehandlung als bei SPAs

- SEO-freundlich (vollständiges HTML vom Server)

Alternative erwogen: React

- Wurde in der Aufgabenstellung als Option erwähnt
- Nicht gewählt, da serverseitiges Rendering für diesen Use Case ausreichend

Bootstrap 5.3

Begründung:

- Responsive Design out-of-the-box
- Mobile-first Ansatz
- Vorgefertigte UI-Komponenten
- Grid-System für flexible Layouts
- Konsistentes Erscheinungsbild
- Große Community, viele Ressourcen

5.3 Datenbank

PostgreSQL 15

Begründung gegenüber Alternativen:

Kriterium	PostgreSQL	MySQL	H2 (In-Memory)
Datum/Zeit-Handling	⭐ ⭐ ⭐ Exzellent	⭐ ⭐ Gut	⭐ ⭐ Gut
Constraints	⭐ ⭐ ⭐ Sehr umfangreich	⭐ ⭐ Gut	⭐ Eingeschränkt
ACID-Konformität	⭐ ⭐ ⭐ Vollständig	⭐ ⭐ ⭐ Vollständig	⭐ ⭐ Teilweise
Performance	⭐ ⭐ ⭐ Sehr gut	⭐ ⭐ ⭐ Sehr gut	⭐ ⭐ ⭐ Sehr schnell
Persistenz	⭐ ⭐ ⭐ Ja	⭐ ⭐ ⭐ Ja	⭐ Nein (Memory)
Produktionsreife	⭐ ⭐ ⭐ Ja	⭐ ⭐ ⭐ Ja	⭐ Nur für Tests

Entscheidung: PostgreSQL

- Robuste Handhabung von Datum/Zeit (kritisch für Reservierungssystem)
- Exzellente Constraint-Unterstützung
- Open Source und kostenlos

- Perfekt für produktionsnahe Entwicklung

5.4 Build und Deployment

Maven 3.9

Begründung:

- Standard Build-Tool für Java
- Klare Projektstruktur
- Zentrales Dependency Management
- Integration in alle IDEs
- Umfangreiches Plugin-Ökosystem

Docker & Docker Compose

Vorteile:

- Einfaches Setup ohne manuelle Datenbankinstallation
- Reproduzierbare Umgebungen
- Isolation von Abhängigkeiten
- Einfaches Deployment
- Plattformunabhängig

5.5 Entwicklungstools

Tool	Zweck
IntelliJ IDEA Ultimate	IDE für Java-Entwicklung
Git / GitHub	Versionskontrolle und Zusammenarbeit
PlantUML	UML-Diagramme (Zustands-, Klassen-, ERM-Diagramm)
Postman	API-Testing (optional)
DBeaver	Datenbank-Verwaltung
Chrome DevTools	Frontend-Debugging

6. Projektstruktur

6.1 Verzeichnisstruktur

```
terminkalender/
│
├── Doku/                                    📁 Projektdokumentation
│   ├── ERM-Diagramm.puml
│   ├── UML-Klassendiagramm.puml
│   ├── UML-Zustandsdiagramm.puml
│   └── Projektdokumentation.pdf             ← Diese Datei
│
├── src/
│   ├── main/
│   │   ├── java/com/terminkalender/
│   │   │   ├── TerminkalenderApplication.java
│   │   │   │
│   │   │   ├── config/
│   │   │   │   └── DataInitializer.java     🔧 Testdaten-Generator
│   │   │   │
│   │   │   ├── controller/
│   │   │   │   ├── IndexController.java     🎮 Startseite
│   │   │   │   └── ReservationController.java
│   │   │   │
│   │   │   ├── dto/
│   │   │   │   └── ReservationDTO.java      📋 Formular-Daten
│   │   │   │
│   │   │   ├── model/
│   │   │   │   ├── Reservation.java         📊 Entität
│   │   │   │   └── Teilnehmer.java
│   │   │   │
│   │   │   ├── repository/
│   │   │   │   └── ReservationRepository.java 💾 Datenzugriff
│   │   │   │
│   │   │   └── service/
│   │   │       ├── ReservationService.java  ⚙️ Business-Logik
│   │   │       └── ValidationService.java   ✅ Validierung
│   │   │
│   │   └── resources/
│   │       ├── templates/                   📄 HTML-Templates
│   │       │   ├── index.html
│   │       │   ├── reservation-formular.html
│   │       │   ├── reservation-bestaetigung.html
│   │       │   ├── reservation-ansicht.html
│   │       │   └── reservation-bearbeiten.html
│   │       │
│   │       ├── static/
│   │       │   └── css/
│   │       │       └── style.css            🎨 Custom Styles
│   │       │
│   │       └── application*.properties      ⚙️ Konfigurationen
│   │
│   └── test/                                🧪 Unit-Tests
│
├── docker-compose.yml                       🐳 Docker-Orchestrierung
├── Dockerfile
```

```
├── init.sql                          DB-Schema
├── pom.xml                           Maven-Dependencies
│
├── *.cmd                             Windows-Scripts
└── START-ANLEITUNG.txt               Benutzerhandbuch
```

6.2 Maven-Konfiguration (pom.xml)

Projekt-Koordinaten:

```
<groupId>com.terminkalender</groupId>
<artifactId>terminkalender</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>Terminkalender</name>
<description>Reservierungssystem für Besprechungsräume</description>
```

Java-Version:

```
<properties>
    <java.version>21</java.version>
</properties>
```

Wichtigste Dependencies:

- spring-boot-starter-web (Spring MVC)
- spring-boot-starter-data-jpa
- spring-boot-starter-thymeleaf
- spring-boot-starter-validation
- postgresql (JDBC Driver)
- spring-boot-devtools (optional)

7. Implementierung

7.1 Model-Klassen (Entitäten)

7.1.1 Reservation.java

```
@Entity
@Table(name = "reservationen")
public class Reservation {

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull(message = "Datum darf nicht leer sein")
    @Future(message = "Datum muss in der Zukunft liegen")
    private LocalDate datum;

    @NotNull(message = "Startzeit darf nicht leer sein")
    private LocalTime vonZeit;

    @NotNull(message = "Endzeit darf nicht leer sein")
    private LocalTime bisZeit;

    @NotNull(message = "Zimmer muss ausgewählt werden")
    @Min(value = 101, message = "Zimmer muss zwischen 101 und 105 liegen")
    @Max(value = 105, message = "Zimmer muss zwischen 101 und 105 liegen")
    private Integer zimmer;

    @NotNull(message = "Bemerkung darf nicht leer sein")
    @Size(min = 10, max = 200,
          message = "Bemerkung muss zwischen 10 und 200 Zeichen haben")
    private String bemerkung;

    @Column(name = "private_schluessel", unique = true, nullable = false, length = 36)
    private String privateKey;

    @Column(name = "public_schluessel", unique = true, nullable = false, length = 36)
    private String publicKey;

    @OneToMany(mappedBy = "reservation",
               cascade = CascadeType.ALL,
               orphanRemoval = true)
    private List<Teilnehmer> teilnehmer = new ArrayList<>();

    @CreationTimestamp
    @Column(name = "erstellt_am", updatable = false)
    private LocalDateTime erstelltAm;

    @UpdateTimestamp
    @Column(name = "aktualisiert_am")
    private LocalDateTime aktualisiertAm;

    // Konstruktoren, Getter und Setter
}
```

Wichtige JPA-Annotationen erklärt:

- `@Entity` : Markiert die Klasse als JPA-Entität (Tabelle)
- `@Table(name = "reservationen")` : Definiert den Tabellennamen
- `@Id` + `@GeneratedValue` : Primary Key mit Auto-Increment
- `@NotNull` , `@Size` , `@Min` , `@Max` : Bean-Validierung (JSR-380)

- `@OneToMany` : 1:N-Beziehung zu Teilnehmern
- `cascade = CascadeType.ALL` : Alle Operationen werden weitergegeben
- `orphanRemoval = true` : Teilnehmer ohne Reservierung werden gelöscht
- `@CreationTimestamp / @UpdateTimestamp` : Automatische Zeitstempel

7.1.2 Teilnehmer.java

```
@Entity
@Table(name = "teilnehmer")
public class Teilnehmer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull(message = "Vorname darf nicht leer sein")
    @Size(min = 1, max = 50)
    @Pattern(regexp = "[a-zA-ZäöüÄÖÜß]+",
             message = "Vorname darf nur Buchstaben enthalten")
    private String vorname;

    @NotNull(message = "Nachname darf nicht leer sein")
    @Size(min = 1, max = 50)
    @Pattern(regexp = "[a-zA-ZäöüÄÖÜß]+",
             message = "Nachname darf nur Buchstaben enthalten")
    private String nachname;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "reservation_id", nullable = false)
    private Reservation reservation;

    // Konstruktoren, Getter und Setter
}
```

Wichtige Punkte:

- `@Pattern` : Regex-Validierung (nur Buchstaben)
- `@ManyToOne` : N:1-Beziehung zur Reservierung
- `FetchType.LAZY` : Lädt Reservierung nur bei Bedarf (Performance)
- `@JoinColumn` : Definiert Foreign Key Spalte

7.1.3 ReservationDTO.java

```
public class ReservationDTO {

    @NotNull(message = "Datum darf nicht leer sein")
    @Future(message = "Datum muss in der Zukunft liegen")
```

```
    private LocalDate datum;

    @NotNull(message = "Startzeit darf nicht leer sein")
    private LocalTime vonZeit;

    @NotNull(message = "Endzeit darf nicht leer sein")
    private LocalTime bisZeit;

    @NotNull(message = "Zimmer muss ausgewählt werden")
    @Min(value = 101)
    @Max(value = 105)
    private Integer zimmer;

    @NotNull(message = "Bemerkung darf nicht leer sein")
    @Size(min = 10, max = 200)
    private String bemerkung;

    @NotNull(message = "Mindestens ein Teilnehmer erforderlich")
    private String teilnehmerListe;

    // Konstruktoren, Getter und Setter
}
```

Zweck des DTO:

- Trennung zwischen Formular-Daten und Datenbank-Entität
- Teilnehmerliste als String (aus Formular)
- Wird später zu Teilnehmer-Objekten geparst

7.2 Repository-Schicht

```
@Repository
public interface ReservationRepository extends JpaRepository<Reservation, Long> {

    // Finde Reservierung über Private Key
    Optional<Reservation> findByPrivateKey(String privateKey);

    // Finde Reservierung über Public Key
    Optional<Reservation> findByPublicKey(String publicKey);

    // Custom Query: Finde überschneidende Reservierungen
    @Query("SELECT r FROM Reservation r WHERE r.zimmer = :zimmer " +
           "AND r.datum = :datum " +
           "AND NOT (r.bisZeit <= :vonZeit OR r.vonZeit >= :bisZeit)")
    List<Reservation> findOverlappingReservations(
        @Param("zimmer") Integer zimmer,
        @Param("datum") LocalDate datum,
        @Param("vonZeit") LocalTime vonZeit,
        @Param("bisZeit") LocalTime bisZeit
    );
```

```
    // Überschneidungen außer eigener Reservierung (für Updates)
    @Query("SELECT r FROM Reservation r WHERE r.zimmer = :zimmer " +
           "AND r.datum = :datum " +
           "AND r.id != :excludeId " +
           "AND NOT (r.bisZeit <= :vonZeit OR r.vonZeit >= :bisZeit)")
    List<Reservation> findOverlappingReservationsExcludingId(
        @Param("zimmer") Integer zimmer,
        @Param("datum") LocalDate datum,
        @Param("vonZeit") LocalTime vonZeit,
        @Param("bisZeit") LocalTime bisZeit,
        @Param("excludeId") Long excludeId
    );
}
```

Erklärung der Überschneidungs-Logik:

Zwei Zeiträume überschneiden sich **NICHT**, wenn:

- Der erste endet, bevor der zweite beginnt: `bisZeit <= vonZeit`
- ODER der erste beginnt, nachdem der zweite endet: `vonZeit >= bisZeit`

Die Negation `NOT (...)` findet alle überschneidenden Reservierungen.

Beispiel:

- Bestehende Reservierung: 10:00 - 12:00
- Neue Reservierung: 11:00 - 13:00
- Überschneidung? **Ja** (11:00 liegt zwischen 10:00 und 12:00)

7.3 Service-Schicht

7.3.1 ReservationService.java (Hauptmethoden)

```
@Service
public class ReservationService {

    @Autowired
    private ReservationRepository repository;

    @Autowired
    private ValidationService validationService;

    /**
     * Erstellt eine neue Reservierung
     * 1. Validierung aller Eingaben
     * 2. Verfügbarkeitsprüfung
     * 3. Generierung eindeutiger Schlüssel (UUIDs)
     * 4. Parsen der Teilnehmerliste
```

```
     * 5. Speichern in Datenbank
     */
    @Transactional
    public Reservation erstelleReservation(ReservationDTO dto)
            throws ValidationException {

        // 1. Validierung
        validationService.validiereReservation(dto);

        // 2. Verfügbarkeitsprüfung
        if (!istZimmerVerfuegbar(dto.getZimmer(), dto.getDatum(),
                                 dto.getVonZeit(), dto.getBisZeit(), null)) {
            throw new ValidationException(
                "Der Raum ist zu diesem Zeitpunkt bereits belegt");
        }

        // 3. Reservierung erstellen
        Reservation reservation = new Reservation();
        reservation.setDatum(dto.getDatum());
        reservation.setVonZeit(dto.getVonZeit());
        reservation.setBisZeit(dto.getBisZeit());
        reservation.setZimmer(dto.getZimmer());
        reservation.setBemerkung(dto.getBemerkung());

        // 4. Eindeutige Schlüssel generieren
        reservation.setPrivateKey(UUID.randomUUID().toString());
        reservation.setPublicKey(UUID.randomUUID().toString());

        // 5. Teilnehmer parsen und hinzufügen
        parseTeilnehmer(dto.getTeilnehmerListe(), reservation);

        // 6. Speichern (Cascade speichert auch Teilnehmer)
        return repository.save(reservation);
    }

    /**
     * Prüft ob ein Zimmer verfügbar ist
     * @param excludeId Bei Update: ID der eigenen Reservierung ausschließen
     */
    public boolean istZimmerVerfuegbar(Integer zimmer, LocalDate datum,
                                       LocalTime von, LocalTime bis,
                                       Long excludeId) {
        List<Reservation> overlapping;

        if (excludeId == null) {
            // Neue Reservierung
            overlapping = repository.findOverlappingReservations(
                zimmer, datum, von, bis);
        } else {
            // Update: Eigene Reservierung ausschließen
            overlapping = repository.findOverlappingReservationsExcludingId(
                zimmer, datum, von, bis, excludeId);
        }
```

```
        return overlapping.isEmpty();
    }

    /**
     * Parst Teilnehmerliste und erstellt Objekte
     * Format: "Max Mustermann, Anna Schmidt, ..."
     */
    private void parseTeilnehmer(String teilnehmerString,
                                 Reservation reservation)
            throws ValidationException {

        List<String[]> teilnehmerDaten =
            validationService.validiereUndParseTeilnehmer(teilnehmerString);

        for (String[] nameParts : teilnehmerDaten) {
            Teilnehmer teilnehmer = new Teilnehmer();
            teilnehmer.setVorname(nameParts[0]);
            teilnehmer.setNachname(nameParts[1]);
            teilnehmer.setReservation(reservation);
            reservation.getTeilnehmer().add(teilnehmer);
        }
    }
}
```

Warum UUID für Schlüssel?

- 36 Zeichen lang (z.B. `550e8400-e29b-41d4-a716-446655440000`)
- Praktisch unmöglich zu erraten (2^122 Möglichkeiten)
- Sicherer als numerische IDs
- Keine sequentielle Nummerierung (Security by Obscurity)

7.3.2 ValidationService.java

```
@Service
public class ValidationService {

    /**
     * Validiert alle Felder einer Reservierung
     */
    public void validiereReservation(ReservationDTO dto)
            throws ValidationException {
        validiereZimmer(dto.getZimmer());
        validiereDatumUndZeit(dto.getDatum(),
                              dto.getVonZeit(),
                              dto.getBisZeit());
        validiereBemerkung(dto.getBemerkung());
        validiereUndParseTeilnehmer(dto.getTeilnehmerListe());
    }
```

```java
/**
 * Validiert Datum und Zeitangaben
 */
public void validiereDatumUndZeit(LocalDate datum,
                                  LocalTime von,
                                  LocalTime bis)
        throws ValidationException {

    // Datum in der Zukunft?
    if (datum == null || datum.isBefore(LocalDate.now())) {
        throw new ValidationException(
            "Das Datum muss in der Zukunft liegen");
    }

    // Zeiten vorhanden?
    if (von == null || bis == null) {
        throw new ValidationException(
            "Start- und Endzeit müssen angegeben werden");
    }

    // Startzeit vor Endzeit?
    if (!von.isBefore(bis)) {
        throw new ValidationException(
            "Die Startzeit muss vor der Endzeit liegen");
    }

    // Mindestdauer 15 Minuten
    long minutes = Duration.between(von, bis).toMinutes();
    if (minutes < 15) {
        throw new ValidationException(
            "Die Reservierung muss mindestens 15 Minuten dauern");
    }
}

/**
 * Validiert und parst Teilnehmerliste
 * Erwartet: "Vorname Nachname, Vorname Nachname, ..."
 * Gibt zurück: List<String[]> mit [Vorname, Nachname]
 */
public List<String[]> validiereUndParseTeilnehmer(String teilnehmerString)
        throws ValidationException {

    if (teilnehmerString == null || teilnehmerString.trim().isEmpty()) {
        throw new ValidationException(
            "Es muss mindestens ein Teilnehmer angegeben werden");
    }

    String[] teilnehmerArray = teilnehmerString.split(",");
    List<String[]> result = new ArrayList<>();

    for (String teilnehmer : teilnehmerArray) {
        String[] nameParts = teilnehmer.trim().split("\\s+");
```

```
            // Genau 2 Teile: Vorname Nachname
            if (nameParts.length != 2) {
                throw new ValidationException(
                    "Falsches Format. Erwartet: " +
                    "Vorname Nachname, Vorname Nachname");
            }

            String vorname = nameParts[0];
            String nachname = nameParts[1];

            // Nur Buchstaben erlaubt (inkl. Umlaute)
            if (!vorname.matches("[a-zA-ZäöüÄÖÜß]+")) {
                throw new ValidationException(
                    "Vorname darf nur Buchstaben enthalten: " + vorname);
            }
            if (!nachname.matches("[a-zA-ZäöüÄÖÜß]+")) {
                throw new ValidationException(
                    "Nachname darf nur Buchstaben enthalten: " + nachname);
            }

            result.add(new String[]{vorname, nachname});
        }

        return result;
    }
}
```

7.4 Controller-Schicht

7.4.1 IndexController.java

```
@Controller
public class IndexController {

    @Autowired
    private ReservationRepository repository;

    /**
     * GET / - Startseite anzeigen
     */
    @GetMapping("/")
    public String index() {
        return "index"; // templates/index.html
    }

    /**
     * POST /schluessel-suche
     * Sucht Reservierung nach Schlüssel und leitet weiter
     */
    @PostMapping("/schluessel-suche")
    public String schluesselSuche(@RequestParam("schluessel") String schluessel,
```

```
                                    Model model) {
        schluessel = schluessel.trim();

        // Zuerst in Private Keys suchen
        Optional<Reservation> byPrivate =
            repository.findByPrivateKey(schluessel);
        if (byPrivate.isPresent()) {
            return "redirect:/reservation/bearbeiten/" + schluessel;
        }

        // Dann in Public Keys suchen
        Optional<Reservation> byPublic =
            repository.findByPublicKey(schluessel);
        if (byPublic.isPresent()) {
            return "redirect:/reservation/ansicht/" + schluessel;
        }

        // Nicht gefunden
        model.addAttribute("error", "Schlüssel nicht gefunden");
        return "index";
    }
}
```

7.4.2 ReservationController.java (Auszug)

```
@Controller
@RequestMapping("/reservation")
public class ReservationController {

    @Autowired
    private ReservationService reservationService;

    @Autowired
    private ReservationRepository repository;

    /**
     * GET /reservation/neu - Formular anzeigen
     */
    @GetMapping("/neu")
    public String neueReservation(Model model) {
        model.addAttribute("reservationDTO", new ReservationDTO());
        return "reservation-formular";
    }

    /**
     * POST /reservation/erstellen
     * Erstellt neue Reservierung nach Validierung
     */
    @PostMapping("/erstellen")
    public String erstelleReservation(
            @Valid @ModelAttribute ReservationDTO dto,
```

```
            BindingResult result,
            Model model) {

        // Bean-Validierung fehlgeschlagen?
        if (result.hasErrors()) {
            return "reservation-formular";
        }

        try {
            Reservation created = reservationService.erstelleReservation(dto);
            return "redirect:/reservation/bestaetigung/" + created.getId();
        } catch (ValidationException e) {
            model.addAttribute("error", e.getMessage());
            return "reservation-formular";
        }
    }

    /**
     * GET /reservation/verfuegbarkeit
     * AJAX-Endpunkt für Echtzeit-Verfügbarkeitsprüfung
     */
    @GetMapping("/verfuegbarkeit")
    @ResponseBody
    public Map<String, Boolean> pruefeVerfuegbarkeit(
            @RequestParam Integer zimmer,
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
                LocalDate datum,
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.TIME)
                LocalTime vonZeit,
            @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.TIME)
                LocalTime bisZeit) {

        boolean verfuegbar = reservationService.istZimmerVerfuegbar(
            zimmer, datum, vonZeit, bisZeit, null);

        return Map.of("verfuegbar", verfuegbar);
    }
}
```

8. Datenbank

8.1 Datenbankschema (init.sql)

```
-- Datenbank erstellen (wenn nicht vorhanden)
CREATE DATABASE IF NOT EXISTS reservations_db;

-- Tabelle: reservationen
CREATE TABLE IF NOT EXISTS reservationen (
```

```
    id BIGSERIAL PRIMARY KEY,
    datum DATE NOT NULL,
    von_zeit TIME NOT NULL,
    bis_zeit TIME NOT NULL,
    zimmer INTEGER NOT NULL CHECK (zimmer BETWEEN 101 AND 105),
    bemerkung VARCHAR(200) NOT NULL CHECK (LENGTH(bemerkung) BETWEEN 10 AND 200),
    private_schluessel VARCHAR(36) UNIQUE NOT NULL,
    public_schluessel VARCHAR(36) UNIQUE NOT NULL,
    erstellt_am TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    aktualisiert_am TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Tabelle: teilnehmer
CREATE TABLE IF NOT EXISTS teilnehmer (
    id BIGSERIAL PRIMARY KEY,
    vorname VARCHAR(50) NOT NULL,
    nachname VARCHAR(50) NOT NULL,
    reservation_id BIGINT NOT NULL,
    CONSTRAINT fk_reservation
        FOREIGN KEY (reservation_id)
        REFERENCES reservationen(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);

-- Indizes für bessere Performance
CREATE INDEX IF NOT EXISTS idx_datum_zimmer
    ON reservationen(datum, zimmer);

CREATE INDEX IF NOT EXISTS idx_private_key
    ON reservationen(private_schluessel);

CREATE INDEX IF NOT EXISTS idx_public_key
    ON reservationen(public_schluessel);

CREATE INDEX IF NOT EXISTS idx_reservation_id
    ON teilnehmer(reservation_id);

-- Trigger für automatisches Update von aktualisiert_am
CREATE OR REPLACE FUNCTION update_aktualisiert_am()
RETURNS TRIGGER AS $
BEGIN
    NEW.aktualisiert_am = CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_update_aktualisiert_am
BEFORE UPDATE ON reservationen
FOR EACH ROW
EXECUTE FUNCTION update_aktualisiert_am();
```

8.2 Datenbankdesign-Entscheidungen

Warum zwei Tabellen?

Alternative 1: Teilnehmer als JSON in TEXT-Feld

```
teilnehmer_json TEXT  -- z.B. '[{"vorname":"Max","nachname":"Mustermann"}]'
```

❌ **Nachteile:**

- Keine Validierung auf Datenbankebene
- Schwierige Suche nach Teilnehmern
- Keine referentielle Integrität

Alternative 2: Separate Tabelle (gewählt) ✅

```
CREATE TABLE teilnehmer (...)
```

✅ **Vorteile:**

- Strukturierte Daten
- Suchbar und abfragbar
- Validierung möglich
- Normalisiert (3. Normalform)

Warum CASCADE?

```
ON DELETE CASCADE
ON UPDATE CASCADE
```

Szenario: Reservierung wird gelöscht

- **Mit CASCADE:** Teilnehmer werden automatisch gelöscht ✅
- **Ohne CASCADE:** Fehlermeldung, manuelle Löschung nötig ❌

Vorteil: Automatische Aufräumarbeit, keine "verwaisten" Teilnehmer

Warum UUIDs als Strings?

```
private_schluessel VARCHAR(36)
```

Alternative: BIGINT als numerische ID

```
private_schluessel BIGINT
```

Entscheidung für UUID:

- **Sicherheit:** 2^122 Möglichkeiten (praktisch nicht zu erraten)
- **Keine Sequenz:** Kein Rückschluss auf Anzahl der Reservierungen
- **Global eindeutig:** Keine Kollisionen

Nachteil: 36 Bytes vs. 8 Bytes (akzeptabler Trade-off)

8.3 Testdaten

Die Klasse `DataInitializer.java` erstellt beim Start automatisch 2-3 Testreservierungen:

```
@Component
public class DataInitializer implements CommandLineRunner {

    @Autowired
    private ReservationRepository repository;

    @Override
    public void run(String... args) {
        // Nur im dev-Profil Testdaten erstellen
        if (repository.count() == 0) {
            // Reservierung 1: Morgen, 10:00-12:00, Zimmer 101
            Reservation r1 = new Reservation();
            r1.setDatum(LocalDate.now().plusDays(1));
            r1.setVonZeit(LocalTime.of(10, 0));
            r1.setBisZeit(LocalTime.of(12, 0));
            r1.setZimmer(101);
            r1.setBemerkung("Team Meeting - Sprint Planning");
            r1.setPrivateKey(UUID.randomUUID().toString());
            r1.setPublicKey(UUID.randomUUID().toString());

            Teilnehmer t1 = new Teilnehmer();
            t1.setVorname("Max");
            t1.setNachname("Mustermann");
            t1.setReservation(r1);
            r1.getTeilnehmer().add(t1);

            repository.save(r1);

            // Weitere Testreservierungen...
        }
    }
}
```

9. Benutzerhandbuch

9.1 System starten

Option 1: Mit Docker (empfohlen)

1. **Docker Desktop starten**
   - Docker Desktop öffnen und warten bis es läuft (grünes Symbol)
2. **Projekt starten**
   - Doppelklick auf `ALLES-STARTEN.cmd`
   - Oder im Terminal: `docker-compose up`
3. **Warten**
   - Erste Start: ca. 2-3 Minuten (Downloads)
   - Folgende Starts: ca. 30 Sekunden
4. **Browser öffnen**
   - Automatisch: http://localhost:8080
   - Manuell: Im Browser eingeben

Option 2: Lokal (für Entwicklung)

1. **PostgreSQL starten**
   - Doppelklick auf `starten-postgres.cmd`
2. **Datenbank initialisieren** (nur beim ersten Mal)
   - Doppelklick auf `initialisieren-db.cmd`
3. **Anwendung starten**
   - In IntelliJ: `TerminkalenderApplication.java` ausführen
   - Oder Terminal: `mvn spring-boot:run`
4. **Browser öffnen**
   - http://localhost:8080

9.2 Reservierung erstellen

Schritt 1: Startseite

![Startseite]

- Button "Neue Reservierung" klicken

Schritt 2: Formular ausfüllen

Pflichtfelder:

1. **Datum**: Wählen Sie ein Datum in der Zukunft
2. **Von**: Startzeit (z.B. 10:00)
3. **Bis**: Endzeit (z.B. 12:00)
4. **Zimmer**: Raumnummer (101-105)
5. **Bemerkung**: Grund der Reservierung (10-200 Zeichen)
6. **Teilnehmer**: Namen im Format "Vorname Nachname, Vorname Nachname"

Beispiel Teilnehmer:

```
Max Mustermann, Anna Schmidt, Peter Müller
```

Echtzeit-Verfügbarkeitsprüfung:

- Während Sie tippen, sehen Sie:
  - ✓ **Verfügbar** (grün)
  - ✗ **Nicht verfügbar** (rot)

Schritt 3: Bestätigung

Nach erfolgreicher Erstellung erhalten Sie:

Private Key (Privater Schlüssel):

```
550e8400-e29b-41d4-a716-446655440000
```

→ **Speichern Sie diesen Schlüssel!** → Ermöglicht Bearbeitung und Löschung

Public Key (Öffentlicher Schlüssel):

```
7c9e6679-7425-40de-944b-e07fc1f90ae7
```

→ Zum Teilen mit Teilnehmern → Nur Ansichtsrecht

9.3 Reservierung ansehen

1. **Schlüssel eingeben**

- Auf Startseite: Schlüssel in Textfeld eingeben
- Button "Suchen" klicken

2. **Automatische Weiterleitung**
   - **Private Key** → Bearbeitungsseite
   - **Public Key** → Ansichtsseite

9.4 Reservierung bearbeiten

Nur mit Private Key möglich!

1. **Private Key eingeben** auf Startseite
2. **Auf Bearbeitungsseite:**
   - Alle Felder sind änderbar
   - Button "Aktualisieren" speichert Änderungen
   - Button "Löschen" entfernt Reservierung (mit Bestätigung)
3. **Nach Aktualisierung:**
   - Erfolgsmeldung
   - Schlüssel bleiben unverändert

9.5 Reservierung löschen

1. **Mit Private Key zur Bearbeitungsseite**
2. **Button "Löschen" klicken**
3. **JavaScript-Bestätigung:**

```
Möchten Sie diese Reservierung wirklich löschen?
Dies kann nicht rückgängig gemacht werden.
```

4. **Nach Bestätigung:**
   - Reservierung wird gelöscht
   - Teilnehmer werden automatisch mitgelöscht (CASCADE)
   - Weiterleitung zur Startseite

10. Testing

10.1 Testdaten

Beim Start im `dev` -Profil werden automatisch 2-3 Testreservierungen erstellt:

Datum	Zeit	Zimmer	Bemerkung	Teilnehmer
Morgen	10:00-12:00	101	Team Meeting	Max Mustermann, Anna Schmidt
Übermorgen	14:00-16:00	103	Client Präsentation	Peter Müller

Private/Public Keys werden in Console ausgegeben.

10.2 Manuelle Testszenarien

Test 1: Erfolgreiche Reservierung

Schritte:

1. Neue Reservierung klicken
2. Formular ausfüllen mit gültigen Daten
3. Erstellen klicken

Erwartetes Ergebnis:

- ✅ Bestätigungsseite mit zwei Schlüsseln
- ✅ Reservierung in Datenbank gespeichert

Test 2: Datum in der Vergangenheit

Schritte:

1. Neue Reservierung
2. Datum: Gestern
3. Erstellen klicken

Erwartetes Ergebnis:

- ❌ Fehlermeldung: "Das Datum muss in der Zukunft liegen"

Test 3: Zu kurze Dauer

Schritte:

1. Neue Reservierung
2. Von: 10:00, Bis: 10:10 (nur 10 Minuten)

3. Erstellen klicken

Erwartetes Ergebnis:

- ❌ Fehlermeldung: "Mindestens 15 Minuten erforderlich"

Test 4: Raum bereits belegt

Schritte:

1. Existierende Reservierung: Morgen, 10:00-12:00, Zimmer 101
2. Neue Reservierung: Morgen, 11:00-13:00, Zimmer 101
3. Erstellen klicken

Erwartetes Ergebnis:

- ❌ Fehlermeldung: "Der Raum ist bereits belegt"
- ✅ Echtzeit-Anzeige zeigt "Nicht verfügbar"

Test 5: Ungültiges Teilnehmerformat

Schritte:

1. Neue Reservierung
2. Teilnehmer: "Max123 Mustermann"
3. Erstellen klicken

Erwartetes Ergebnis:

- ❌ Fehlermeldung: "Namen dürfen nur Buchstaben enthalten"

Test 6: Schlüsselsuche

Test 6a - Private Key:

- ✅ Weiterleitung zur Bearbeitungsseite
- ✅ Alle Felder editierbar

Test 6b - Public Key:

- ✅ Weiterleitung zur Ansichtsseite
- ✅ Nur Leserechte

Test 6c - Ungültiger Schlüssel:

- ❌ Fehlermeldung: "Schlüssel nicht gefunden"

10.3 Unit-Tests (Beispiele)

```java
@SpringBootTest
class ReservationServiceTest {

    @Autowired
    private ReservationService service;

    @Test
    void testErstelleReservation_Success() {
        ReservationDTO dto = new ReservationDTO();
        dto.setDatum(LocalDate.now().plusDays(1));
        dto.setVonZeit(LocalTime.of(10, 0));
        dto.setBisZeit(LocalTime.of(12, 0));
        dto.setZimmer(101);
        dto.setBemerkung("Test Reservierung für Unit Test");
        dto.setTeilnehmerListe("Max Mustermann");

        Reservation result = service.erstelleReservation(dto);

        assertNotNull(result.getId());
        assertNotNull(result.getPrivateKey());
        assertNotNull(result.getPublicKey());
        assertEquals(1, result.getTeilnehmer().size());
    }

    @Test
    void testErstelleReservation_DatumInVergangenheit() {
        ReservationDTO dto = new ReservationDTO();
        dto.setDatum(LocalDate.now().minusDays(1)); // Gestern

        assertThrows(ValidationException.class,
            () -> service.erstelleReservation(dto));
    }
}
```

11. Deployment

11.1 Docker-Konfiguration

docker-compose.yml

```yaml
version: '3.8'

services:
  postgres:
    image: postgres:15-alpine
    container_name: postgres-terminkalender
    environment:
```

```
      POSTGRES_DB: reservations_db
      POSTGRES_USER: reservations_user
      POSTGRES_PASSWORD: reservations_pass
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - terminkalender-network

  app:
    build: .
    container_name: terminkalender-app
    depends_on:
      - postgres
    environment:
      SPRING_PROFILES_ACTIVE: docker
      DB_HOST: postgres
      DB_PORT: 5432
      DB_NAME: reservations_db
      DB_USERNAME: reservations_user
      DB_PASSWORD: reservations_pass
    ports:
      - "8080:8080"
    networks:
      - terminkalender-network

volumes:
  postgres_data:

networks:
  terminkalender-network:
    driver: bridge
```

Dockerfile

```
FROM eclipse-temurin:21-jdk-alpine AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN apk add --no-cache maven
RUN mvn clean package -DskipTests

FROM eclipse-temurin:21-jre-alpine
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

11.2 Konfigurationsprofile

application.properties (Standard)

```
# Server-Konfiguration
server.port=${SERVER_PORT:8080}

# Aktives Profil
spring.profiles.active=${SPRING_PROFILES:dev}

# Logging
logging.level.root=INFO
logging.level.com.terminkalender=DEBUG
```

application-dev.properties (Entwicklung)

```
# Datenbank
spring.datasource.url=jdbc:postgresql://localhost:5432/reservations_db
spring.datasource.username=reservations_user
spring.datasource.password=reservations_pass

# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Testdaten laden
data.initializer.enabled=true
```

application-docker.properties

```
# Datenbank (Docker Service Name)
spring.datasource.url=jdbc:postgresql://postgres:5432/reservations_db
spring.datasource.username=${DB_USERNAME:reservations_user}
spring.datasource.password=${DB_PASSWORD:reservations_pass}

# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=false

# Keine Testdaten
data.initializer.enabled=false
```

application-prod.properties (Produktion)

```
# Datenbank (von Umgebungsvariablen)
spring.datasource.url=${DATABASE_URL}
```

```
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}

# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=false

# Logging minimal
logging.level.root=WARN
logging.level.com.terminkalender=INFO
```

11.3 Deployment-Anleitung

Lokale Entwicklung

```
# 1. PostgreSQL starten
docker-compose up -d postgres

# 2. Anwendung starten
mvn spring-boot:run -Dspring-boot.run.profiles=dev
```

Docker (Komplett)

```
# Alles starten
docker-compose up -d

# Logs ansehen
docker-compose logs -f app

# Stoppen
docker-compose down

# Mit Datenbank löschen
docker-compose down -v
```

Produktion (Beispiel)

```
# 1. JAR bauen
mvn clean package -DskipTests

# 2. Auf Server kopieren
scp target/terminkalender-0.0.1-SNAPSHOT.jar user@server:/opt/app/

# 3. Auf Server ausführen
java -jar terminkalender-0.0.1-SNAPSHOT.jar \
```

```
  --spring.profiles.active=prod \
  --server.port=8080
```

12. Probleme und Lösungen

12.1 Häufige Probleme

Problem 1: "Connection refused" zur Datenbank

Symptome:

```
org.postgresql.util.PSQLException: Connection refused
```

Ursachen und Lösungen:

Ursache	Lösung
PostgreSQL läuft nicht	`docker-compose up -d postgres` oder `starten-postgres.cmd`
Falscher Port	In `application.properties` Port 5432 prüfen
Falscher Host	Bei Docker: `postgres`, lokal: `localhost`
Firewall blockiert	Port 5432 in Firewall freigeben

Prüfung:

```
# Docker
docker ps | grep postgres

# Direkte Verbindung testen
psql -h localhost -U reservations_user -d reservations_db
```

Problem 2: Port 8080 bereits belegt

Symptome:

```
Port 8080 is already in use
```

Lösungen:

Option A - Anderen Port verwenden:

```
mvn spring-boot:run -Dserver.port=8081
```

Option B - Prozess beenden (Windows):

```
netstat -ano | findstr :8080
taskkill /PID [PID] /F
```

Option C - Prozess beenden (Linux/Mac):

```
lsof -i :8080
kill -9 [PID]
```

Problem 3: Tabellen existieren nicht

Symptome:

```
ERROR: relation "reservationen" does not exist
```

Lösung im dev-Profil:

```
# application-dev.properties
spring.jpa.hibernate.ddl-auto=update
```

Lösung im docker/prod-Profil:

```
# init.sql manuell ausführen
docker exec -i postgres-terminkalender psql \
  -U reservations_user -d reservations_db < init.sql
```

Problem 4: Änderungen werden nicht übernommen

Symptome:

- Code geändert, aber alte Version läuft

Lösungen:

Bei lokaler Entwicklung:

```
mvn clean compile
# oder in IDE: Build → Rebuild Project
```

Bei Docker:

```
docker-compose down
docker-compose build --no-cache
docker-compose up
```

Problem 5: Teilnehmer werden nicht gelöscht

Symptom:

- Nach Update bleiben alte Teilnehmer erhalten

Ursache:

```
// FALSCH
reservation.getTeilnehmer().addAll(neueTeilnehmer);
```

Lösung:

```
// RICHTIG
reservation.getTeilnehmer().clear();
reservation.getTeilnehmer().addAll(neueTeilnehmer);
```

Problem 6: Echtzeit-Verfügbarkeit funktioniert nicht

Symptome:

- Indikator zeigt nichts an
- JavaScript-Fehler in Console

Prüfungen:

1. **Browser Console öffnen (F12)**
2. **Network Tab prüfen:**
   - Request zu `/reservation/verfuegbarkeit` ?
   - Status 200?
   - JSON-Response?

Häufige Fehler:

```
// FALSCH - ohne URL-Encoding
fetch(`/reservation/verfuegbarkeit?vonZeit=${vonZeit}`)
```

```
// RICHTIG
fetch(`/reservation/verfuegbarkeit?vonZeit=${encodeURIComponent(vonZeit)}`)
```

12.2 Performance-Optimierungen

Optimierung 1: Indizes

Ohne Index:

```
SELECT * FROM reservationen
WHERE datum = '2025-11-15' AND zimmer = 101;
-- Vollständiger Table Scan: ~100ms
```

Mit Index:

```
CREATE INDEX idx_datum_zimmer ON reservationen(datum, zimmer);
-- Index Scan: ~5ms
```

Optimierung 2: Lazy Loading

```
// EAGER (lädt immer alle Teilnehmer)
@ManyToOne(fetch = FetchType.EAGER)

// LAZY (lädt nur bei Bedarf) ✅
@ManyToOne(fetch = FetchType.LAZY)
```

Optimierung 3: Batch Processing

```
// Bei vielen Teilnehmern
spring.jpa.properties.hibernate.jdbc.batch_size=20
```

13. Fazit und Ausblick

13.1 Projektergebnisse

Erreichte Ziele ✅

- ✅ **Vollständige Webanwendung** mit allen geforderten Funktionen
- ✅ **Schlüsselsystem** für sichere Reservierungsverwaltung ohne Login
- ✅ **Echtzeit-Verfügbarkeitsprüfung** für bessere User Experience

- ✅ **Responsive Design** für Desktop und Mobile
- ✅ **Docker-Deployment** für einfache Installation
- ✅ **Umfassende Dokumentation** mit UML-Diagrammen

Technische Highlights

- **Saubere Architektur:** Klare Trennung nach MVC-Pattern
- **Robuste Validierung:** 3 Schichten (HTML5, Bean Validation, Business Logic)
- **Datenbankintegrität:** Constraints, Foreign Keys, CASCADE
- **UUID-basierte Sicherheit:** Praktisch unknackbare Schlüssel
- **Automatische Zeitstempel:** Keine manuelle Verwaltung nötig

13.2 Lessons Learned

Was hat gut funktioniert?

1. Spring Boot als Framework

- Schnelle Entwicklung durch Auto-Configuration
- Hervorragende IDE-Integration
- Umfangreiche Community-Unterstützung

2. Schlüsselsystem statt Login

- Einfacher für Benutzer (keine Registrierung)
- Weniger Code (keine Authentifizierung)
- Ausreichend sicher für internen Use Case

3. Docker für Deployment

- Einfaches Setup für Entwickler
- Reproduzierbare Umgebungen
- Keine manuellen Datenbankinstallationen

Herausforderungen

1. Überschneidungs-Logik

- Anfangs kompliziert zu verstehen
- Query mehrmals optimiert
- Wichtig: Eigene Reservierung bei Updates ausschließen

2. Teilnehmer-Parsing

- String zu Objekten konvertieren fehleranfällig

- Umfangreiche Validierung notwendig
- Alternative: Dynamische Formularfelder (mehr JavaScript)

3. Zeitstempel-Management

- Anfangs manuell mit `LocalDateTime.now()`
- Oft vergessen zu aktualisieren
- Lösung: `@CreationTimestamp` / `@UpdateTimestamp`

13.3 Mögliche Erweiterungen

Priorität 1: Must-Have für Produktion

1. E-Mail-Benachrichtigungen

```
@Service
public class EmailService {
    public void sendeSchluessel(String email, String privateKey, String publicKey) {
        // Spring Mail Integration
    }
}
```

- Schlüssel per E-Mail versenden
- Erinnerungen vor Meetings
- Bestätigung bei Änderungen

2. Kalender-Export (ICS)

```
public ICalendar exportiereAlsICS(Reservation reservation) {
    // ICS-Datei generieren
}
```

- In Outlook/Google Calendar importieren
- Automatische Meeting-Erinnerungen

3. Admin-Dashboard

- Übersicht aller Reservierungen
- Statistiken (meist genutzte Räume)
- Manuelle Verwaltung

Priorität 2: Nice-to-Have

4. Visueller Kalender

```
// FullCalendar.io Integration
$('#calendar').fullCalendar({
    events: '/api/reservations'
});
```

- Monats-/Wochenansicht
- Drag & Drop für neue Reservierungen
- Farbcodierung nach Räumen

5. Wiederkehrende Reservierungen

```
public class RecurringReservation {
    private RecurrencePattern pattern; // DAILY, WEEKLY, MONTHLY
    private LocalDate endDate;
}
```

- "Jeden Montag um 10:00"
- Bis zu einem Enddatum

6. Suche nach Teilnehmern

```
SELECT r.* FROM reservationen r
JOIN teilnehmer t ON r.id = t.reservation_id
WHERE t.vorname = 'Max' AND t.nachname = 'Mustermann';
```

- "Wann hat Max Meetings?"
- Verfügbarkeit von Personen prüfen

7. Raum-Equipment und Kapazität

```
public class Raum {
    private Integer nummer;
    private Integer kapazitaet;
    private List<String> ausstattung; // Beamer, Whiteboard, etc.
}
```

- Filtern nach Ausstattung
- Warnungen bei zu vielen Teilnehmern

8. Mobile App

- React Native oder Flutter
- Push-Benachrichtigungen
- Offline-Modus

9. Integration mit Slack/Teams

```
@Service
public class SlackService {
    public void posteReservierung(Reservation r) {
        // Webhook zu Slack
    }
}
```

- Automatische Benachrichtigungen
- Bot-Commands "/reserve Zimmer 101"

10. Mehrsprachigkeit (i18n)

```
# messages_de.properties
reservation.created=Reservierung erstellt

# messages_en.properties
reservation.created=Reservation created
```

13.4 Abschließende Bewertung

Stärken des Projekts

Aspekt	Bewertung	Kommentar
Funktionalität	⭐⭐⭐⭐⭐	Alle Anforderungen erfüllt
Code-Qualität	⭐⭐⭐⭐	Sauber strukturiert, gut dokumentiert
Benutzerfreundlichkeit	⭐⭐⭐⭐	Intuitiv, klare Fehlermeldungen
Sicherheit	⭐⭐⭐⭐	UUID-Schlüssel, SQL-Injection-Schutz
Performance	⭐⭐⭐⭐	Indizes, Lazy Loading
Wartbarkeit	⭐⭐⭐⭐⭐	MVC-Pattern, umfassende Doku

Projekterfolg

Das Projekt hat alle gesteckten Ziele erreicht:

✅ **Funktional vollständig:** Alle Anforderungen aus der Aufgabenstellung implementiert

✅ **Technisch solide:** Moderne Technologien, Best Practices befolgt

✅ **Gut dokumentiert:** Umfassende Dokumentation mit UML-Diagrammen

✅ **Einfach deploybar:** Docker-basiertes Setup

✅ **Erweiterbar:** Klare Architektur ermöglicht zukünftige Features

Das Reservierungssystem ist **produktionsreif** für den internen Einsatz und bietet eine solide Basis für zukünftige Erweiterungen.

Anhang

A. Glossar

Begriff	Erklärung
Bean Validation	Standard (JSR-380) für deklarative Validierung mit Annotationen
CASCADE	Automatisches Weitergeben von Datenbankoperationen auf verknüpfte Entitäten
DTO	Data Transfer Object - Objekt zum Datentransport zwischen Schichten
JPA	Java Persistence API - Standard für ORM in Java
MVC	Model-View-Controller - Architekturmuster
ORM	Object-Relational Mapping - Abbildung von Objekten auf Datenbanktabellen
UUID	Universally Unique Identifier - 128-Bit eindeutige Kennung

B. Verwendete Technologien (Versionen)

Technologie	Version	Lizenz
Java	21 (LTS)	GPL v2 + Classpath Exception
Spring Boot	3.2.0	Apache 2.0
PostgreSQL	15	PostgreSQL License
Thymeleaf	3.1	Apache 2.0
Bootstrap	5.3	MIT
Maven	3.9	Apache 2.0
Docker	Latest	Apache 2.0

C. Wichtige Links

- **Spring Boot Dokumentation:** https://spring.io/projects/spring-boot
- **PostgreSQL Dokumentation:** https://www.postgresql.org/docs/
- **Thymeleaf Dokumentation:** https://www.thymeleaf.org/documentation.html
- **Bootstrap Dokumentation:** https://getbootstrap.com/docs/
- **Projekt Repository:** https://github.com/EriskReyes/Reservationssystem.git

D. Kontakt und Support

Team-Kontakt:

- Rigo Erisk Reyes
- Denis Perdomo

Projektbetreuer:

- Andrea Casauro

Institution:

- [Schule/Institution]
- Modul 223 - Multiuser-Applikationen objektorientiert realisieren

Ende der Dokumentation

Datum: November 2025
Version: 1.0
Seitenzahl: [wird beim PDF-Export generiert]