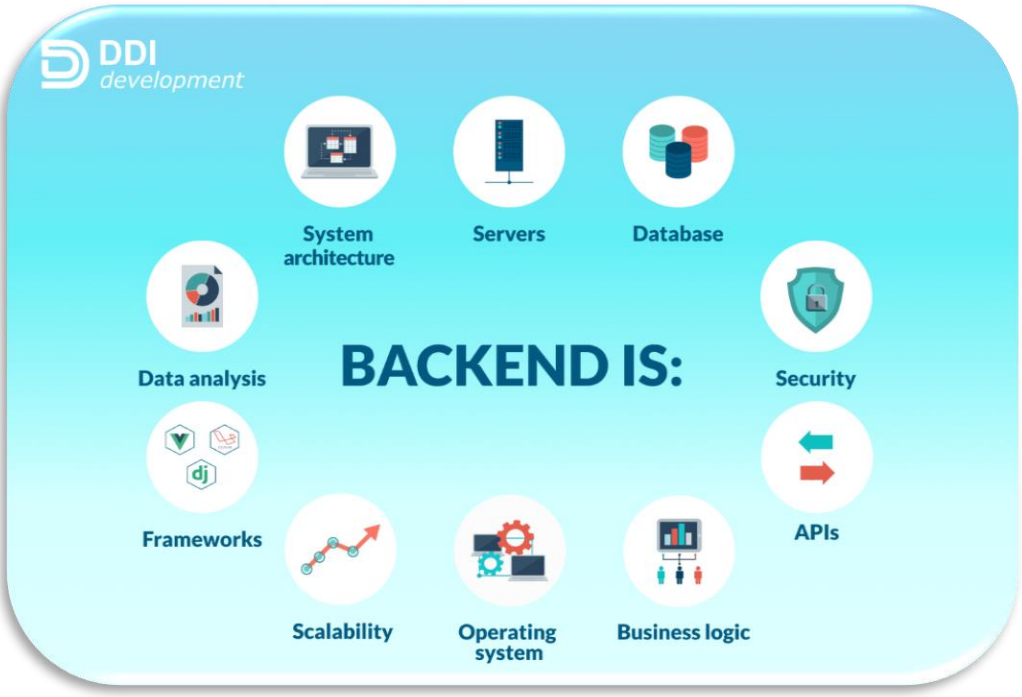
MODUL 295

Backend-Skiservice



FEBRUARY 17, 2025

IBZ AARAU
David Lindörfer

1. Inhaltsverzeichnis

1. Informieren

1.1. Projektziel

Die Firma Jetstream-Service ist ein kleines Unternehmen und bietet im Winter Skiservice an. Jetzt soll die Verwaltung der Aufträge komplett digital werden. Dafür wird eine neue Web- und Datenbank-Anwendung entwickelt.

Die bisherige Online-Anmeldung bleibt bestehen, wird aber erweitert. In der Hochsaison arbeiten bis zu zehn Mitarbeiter am Skiservice. Sie sollen sich mit einem Passwort einloggen können, um Aufträge einzusehen, zu übernehmen und zu bearbeiten.

1.1.1. Anforderungen

- **Funktionen**:
  - Login mit Benutzername und Passwort.
  - Anzeige und Mutation von Serviceaufträgen (Status: Offen, InArbeit, Abgeschlossen).
  - Löschen von Aufträgen (z. B. bei Stornierung).
  - Verwaltung von Dienstleistungen (z. B. «Kleiner Service», «Großer Service»).
- **Technische Anforderungen**:
  - Web-API mit Authentifikation.
  - Datenbankdesign und Implementierung (Code First oder Database First).
  - Testprojekt und Testplan (Unit-Tests, Postman-Tests).
  - OpenAPI-Dokumentation.

A1	Login Dialog mit Passwort für den autorisierten Zugang der Mitarbeiter (Datenänderungen).
A2	In der Datenbank müssen die Informationen des Serviceauftrags und die Login Daten der Mitarbeiter verwaltet sein.
A3	Erfasste Serviceaufträge abrufbar sein.
A4	Die erfassten Serviceaufträge müssen selektiv nach Priorität abrufbar sein.
A5	Mitarbeiter können eine Statusänderung eines Auftrages vornehmen.
A6	Mitarbeiter können Aufträge löschen (z.B. bei Stornierungen)
A7	Die aufgerufenen API-Endpoints müssen zwecks Fehlerlokalisierung protokolliert sein (DB oder Protokolldatei).
A8	Datenbankstruktur muss normalisiert in der 3.NF sein inkl. referenzieller Integrität
A9	Für die Web-API Applikation muss ein eigener Datenbankbenutzerzugang mit eingeschränkter Berechtigung (DML) zur Verfügung gestellt werden (Benutzer root bzw. sa ist verboten).
A10	Das Web-API muss vollständig nach Open-API (Swagger) dokumentiert sein.
A11	Das Softwareprojekt ist über ein Git-Repository zu verwalten.
A12	Ganzes Projektmanagement muss nach IPERKA dokumentiert sein

1.1.2. Technologien

Für mein Projekt nutze ich verschiedene Technologien, um eine sichere und gut funktionierende Anwendung zu entwickeln.

Spring Boot: Ich verwende Spring Boot, um das Backend zu programmieren. Damit kann ich schnell und einfach Webanwendungen mit Java erstellen.

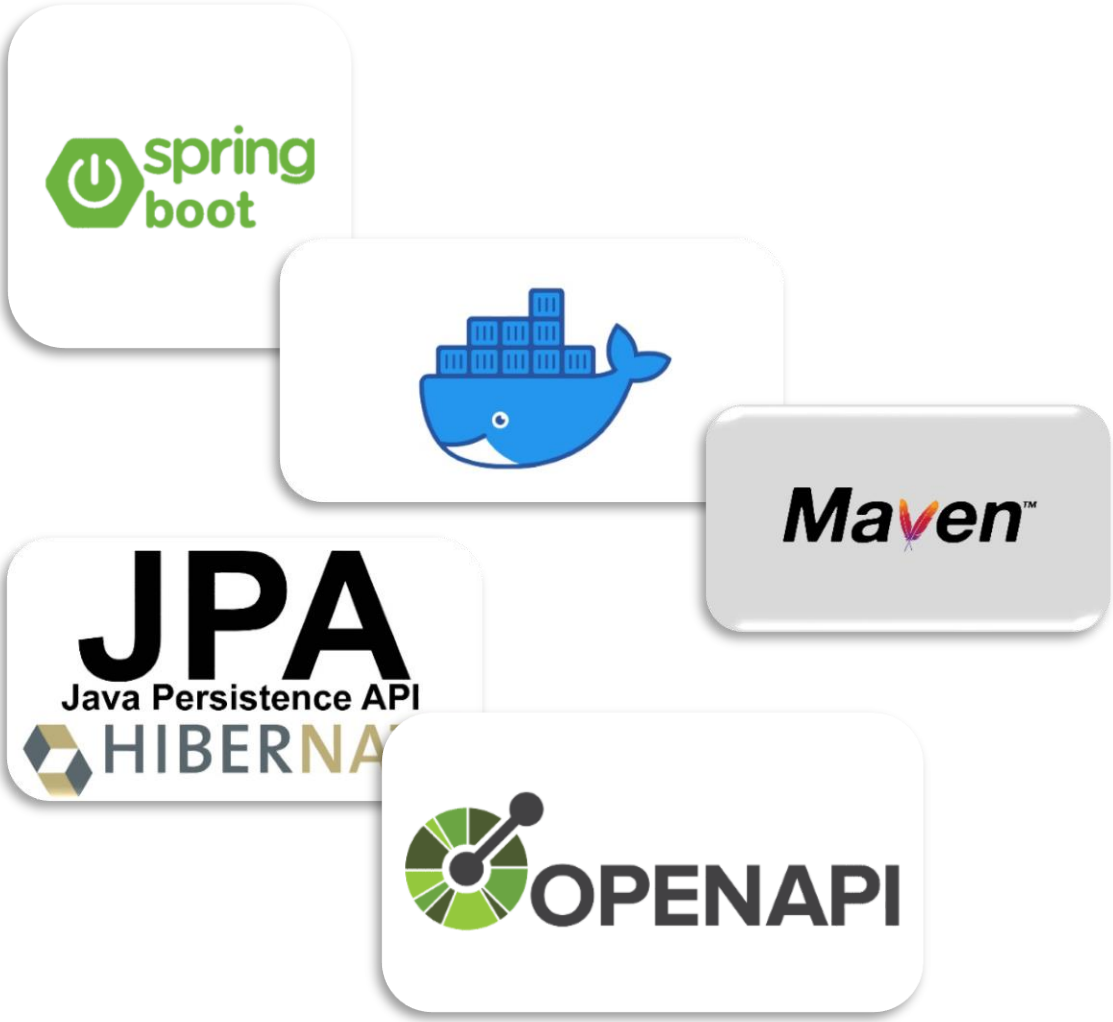
Docker: Mit Docker verpacke ich meine Anwendung und die Datenbank in Containern.

Maven: Dieses Tool hilft mir, wichtige Bibliotheken zu verwalten und mein Projekt zu bauen. So stelle ich sicher, dass alle benötigten Abhängigkeiten vorhanden sind.

JPA/Hibernate: Ich nutze JPA mit Hibernate, um mein Java-Backend mit der Datenbank zu verbinden. Dadurch kann ich Daten speichern und abrufen, ohne komplizierte SQL-Befehle schreiben zu müssen.

Spring Security: Um meine Anwendung sicher zu machen, verwende ich Spring Security. Damit kann ich Benutzer verwalten, Passwörter schützen und den Zugriff auf bestimmte Bereiche einschränken.

OpenAPI: Mit OpenAPI dokumentiere ich meine API. So kann ich genau festhalten, wie das Frontend mit dem Backend kommuniziert.



2. Planen

2.1. Projektstruktur

- **Ordnerstruktur**:
    - Mvnw: Maven Wrapper für den Build-Prozess.
    - Docker-compose.yml: Docker-Konfiguration für die Containerisierung.
    - Src/main/java: Enthält den gesamten Java-Code, aufgeteilt in verschiedene Ordner für Konfiguration, Controller, Entities, Models, Repositories und Service (Java-Code).
        - Config: Konfigurationsdateien (z. B. SecurityConfig).
        - Controller: API-Endpunkte, damit zwischen dem Benutzer vermittelt werden kann.
        - Entity: JPA-Entities.
        - Models: Datenmodelle und Enums, das Enum damit man den Status definieren kann z.B. «Offen», «InArbeit», «Abgeschlossen».
        - Repository: JPA-Repositories, damit man die Daten aus der Datenbank lesen, speichern und löschen kann, ohne direkt in SQL zu schreiben zu müssen.
        - Service: Geschäftslogik.
- **Datenbankdesign**:

  Entities:
    - ActionLog: Protokolliert Aktionen, die Benutzer durchführen (z. B. «Auftrag aktualisiert»).
    - ServiceAngebot: Verwaltet die Dienstleistungen (z. B. «Kleiner Service», «Großer Service»).
    - ServiceAuftrag: Verwaltet die Serviceaufträge (z. B. Aufträge mit Status und Priorität).
    - User: Verwaltet die Benutzerdaten (z. B. Name, E-Mail, Passwort).

  Beziehungen:

  Jeder Serviceauftrag ist mit genau einer Dienstleistung verknüpft. Zum Beispiel:
    - Ein Auftrag für einen «Kleinen Service» ist mit dem ServiceAngebot «Kleiner Service» verknüpft.
    - Diese Beziehung ermöglicht es, die Dienstleistung eines Auftrags einfach zu verwalten.

2.2. Zeitplan

1. **Woche 1**: Projekt-Setup, Datenbankdesign, Entities erstellen.
2. **Woche 2**: Implementierung der Service-Schicht und Controller.
3. **Woche 3**: Integration von Spring Security und Authentifikation.
4. **Woche 3**: Tests schreiben (Unit-Tests, Postman-Tests).
5. **Woche 4**: Dokumentation und Feinschliff.

3. Entscheiden

3.1. Technologische Entscheidungen

- **Spring Boot**: Als Framework für die Backend-Entwicklung, da es viele integrierte Funktionen bietet.
- **Docker**: Mit Docker packe ich meine Anwendung und die Datenbank, damit sie überall gleich funktionieren.
- **JPA/Hibernate**: Für die objektrelationale Abbildung (ORM) und die einfache Datenbankanbindung.
- **Spring Security**: Für die sichere Authentifikation und Autorisierung.

3.2. Architekturentscheidungen

- **Layered Architecture**: Trennung in Controller, Service und Repository für eine klare Struktur.
- **Code First**: Die Datenbank wird aus den Entities generiert, um Flexibilität zu gewährleisten.

4. Realisieren

4.1. Projekt-Setup

- **Spring Boot-Projekt erstellen**: Mit Spring Initializer und den notwendigen Abhängigkeiten (Spring Web, Spring Data JPA, Spring Security, Lombok).
- **Docker einrichten**: Docker-compose.yml für die Containerisierung der Anwendung und Datenbank.

```yaml
version: "3.1"
services:

  db:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: "password"
      MYSQL_DATABASE: "testdb"
      MYSQL_USER: "user"
      MYSQL_PASSWORD: "password"
    ports:
      - 3306:3306
```

4.2. Datenbankdesign und Entities

- **Entities definieren**:
    - ActionLog: Protokollierung von Aktionen.
    - ServiceAngebot: Verwaltung der Dienstleistungen.
    - ServiceAuftrag: Verwaltung der Serviceaufträge.
    - User: Benutzerverwaltung.

  Die Entity-Klasse hat die Aufgabe, eine Tabelle in einer Datenbank zu repräsentieren. Das bedeutet sie beschreibt, wie die Daten in der Datenbank strukturiert sind und wie sie mit der Anwendung interagieren.

```
@NonNull
private String vorname;
private String nachname;
private String email;
private String telefon;
private LocalDate abholdatum;
private LocalDate erstelldatum;
```

- **Repositories erstellen**: JPA-Repositories für den Datenbankzugriff.

4.3. Implementierung der Service-Schicht

- **Service-Klassen**:
    - ActionLogService: Logik für die Protokollierung von Aktionen.
    - ServiceAuftragService: Logik für die Verwaltung von Serviceaufträgen.
    - UserService: Logik für die Benutzerverwaltung und Authentifikation.

  Der Service ist für die spezifische Geschäftslogik oder Funktionalitäten zuständig.

4.4. Implementierung der Controller

- **API-Endpunkte**:
    - ServiceAngebotController: Verwaltung der Dienstleistungen.
    - ServiceAuftragController: Verwaltung der Serviceaufträge.
    - SignupController: Registrierung von Benutzern.
    - StaticHtmlController: Bereitstellung von statischen HTML-Seiten.

  Der **Controller** ist dafür zuständig, HTTP-Anfragen (Requests) zu empfangen, zu verarbeiten und eine Antwort (Response) zurückzugeben. Er vermittelt zwischen dem Client und dem Model (Anwendungslogik).

4.5. Authentifikation und Sicherheit

- **Spring Security konfigurieren**:
    - SecurityConfig für die Authentifikation und Autorisierung.
    - Passwortgeschützter Zugang für Mitarbeiter.

```
public SecurityFilterChain securityFilterChain(HttpSecurity
                                               httpSecurity) throws Exception {
    return httpSecurity
            .csrf().disable()
            .csrf(AbstractHttpConfigurer::disable)
            .formLogin(httpForm -> {
                httpForm.loginPage("/login");
                httpForm.defaultSuccessUrl("/index");
            })
```

- Ich deaktiviere CSRF-Schutz, weil ich das in meiner Anwendung nicht brauche.
- Ich konfiguriere die Anmeldeseite:
- Die Login-Seite ist unter /login erreichbar.
- Nach erfolgreicher Anmeldung werde ich zur /index-Seite weitergeleitet.

```
.logout(logout -> logout
        .logoutUrl("/logout") // Logout-URL
        .logoutSuccessUrl("/login?logout") // Weiterleitung nach erfolgreichem Logout
        .invalidateHttpSession(true) // Sitzung ungültig machen
        .deleteCookies( ...cookieNamesToClear: "JSESSIONID") // Cookies löschen
        .permitAll()
)
```

- Ich konfiguriere den Logout:
- Der Logout wird über die URL /logout ausgelöst.
- Nach dem Logout werde ich zur Login-Seite (/login?logout) weitergeleitet.
- Die aktuelle Sitzung wird beendet, und das Session-Cookie (JSESSIONID) wird gelöscht.

```
.authorizeHttpRequests(registry -> {
    registry.requestMatchers("/login", "/signup", "/css/**", "/js/**", "/media/**").permitAll();
    registry.anyRequest().authenticated();
})
.build();
```

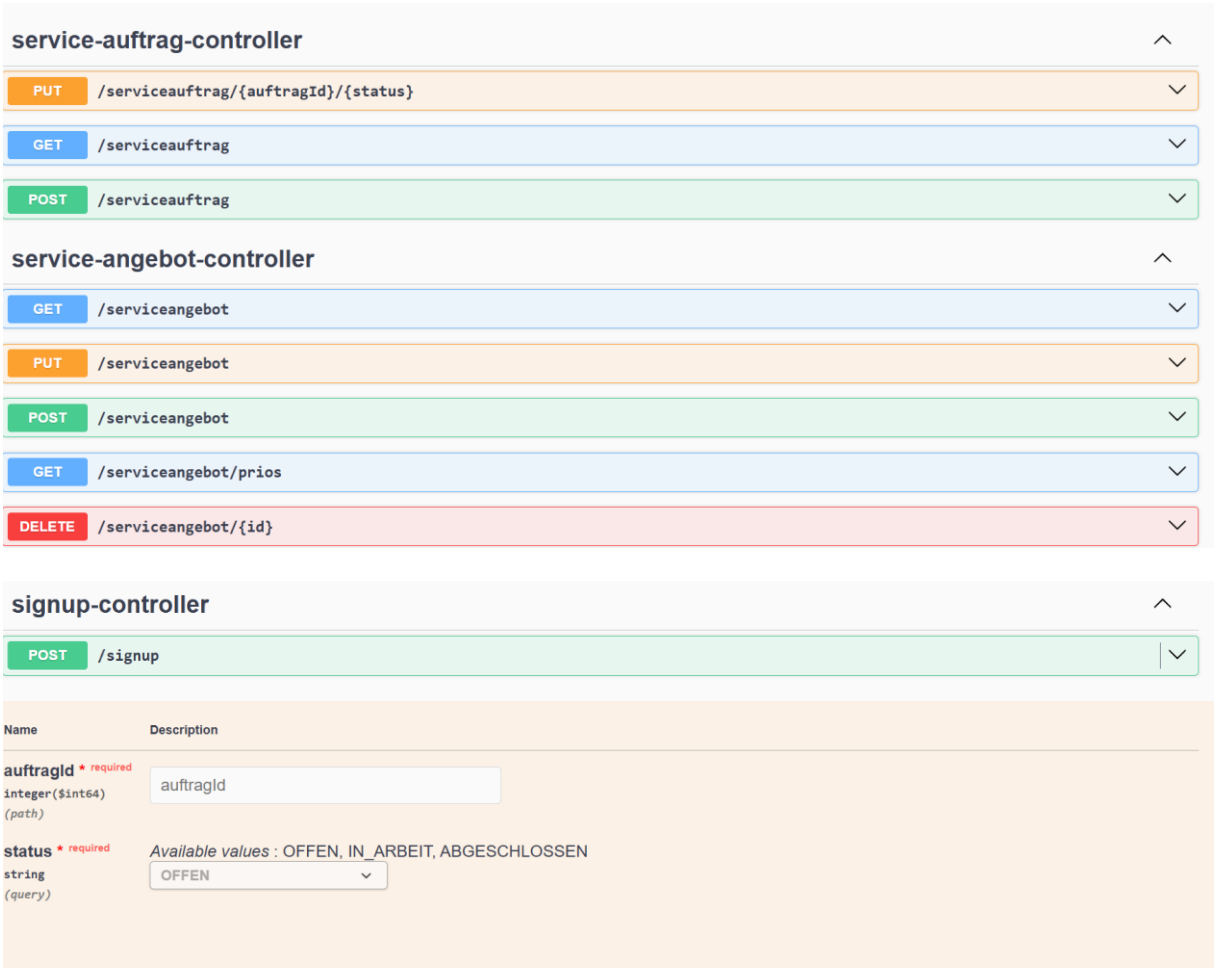
- Ich lege fest, welche Seiten öffentlich sind und welche geschützt werden:
- Seiten wie /login, /signup, CSS-, JS- und Medien-Dateien sind für alle zugänglich.
- Alle anderen Seiten sind nur für angemeldete Benutzer sichtbar.
- Ich mache das ganze mit @Bean, damit Spring diese Sicherheitskonfiguration automatisch übernimmt und verwaltet.

```java
@Bean  David L
public AuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
    provider.setUserDetailsService(appUserService);
    provider.setPasswordEncoder(passwordEncoder());
    return provider;
```

- Ich konfiguriere einen AuthenticationProvider
- Dieser ist dafür zuständig, Benutzer zu authentifizieren (also zu überprüfen, ob sie richtig sind).
- Ich verwende DaoAuthenticationProvider, der den UserService und ein Passwort-Verschlüsselungsverfahren (BCryptPasswordEncoder) nutzt.

4.6. Swagger

Swagger aufrufen URL: http://localhost:8085/swagger-ui/index.html



5. Kontrollieren

5.1. Unit-Tests

- **Service-Schicht testen**: Unit-Tests für ServiceAuftragService und UserService.
- **Controller testen**: Unit-Tests für die API-Endpunkte.

5.2. Integrationstests

- **Postman-Tests**: Testen der API-Endpunkte (z. B. Erstellen eines Auftrags, Ändern des Status).

5.3. Code-Reviews

- **Qualitätssicherung**: Überprüfung des Codes auf Fehler und Optimierungspotenzial.

6. Auswerten

6.1. Ergebnisse

- Die Anwendung erfüllt alle Anforderungen:
    - Mitarbeiter können sich authentifizieren und Aufträge verwalten.
    - Die API ist gut dokumentiert und getestet.
    - Die Datenbank ist korrekt implementiert und mit den Entities verknüpft.

6.2. Reflexion

- **Erfolge**:
    - Der Code ist klar strukturiert (Architektur).
    - Effiziente Implementierung der Authentifikation mit Spring Security.
- **Herausforderungen**:
    - Die Beziehung zwischen den Entities korrekt implementieren.
    - Die Integration von Docker und Spring Boot war anfangs komplex.
    - Spring Security (Fehler)

6.3. Verbesserungsvorschläge

- **Erweiterung der Funktionalität**: Hinzufügen von Benachrichtigungen (z. B. E-Mail bei Statusänderung). Mitarbeiter können zu einem Auftrag einen Kommentar hinterlegen. Gelöschte Aufträge werden nicht aus der Datenbank entfernt, sondern nur als gelöscht markiert.
- **Performance-Optimierung**: **Caching,** damit du die Daten nicht jedes Mal neu aus der Datenbank holen musst, sondern die häufigen abgerufenen Daten zwischenspeicherst.

7. Fazit

Im Laufe dieses Projekts habe ich gelernt, wie wichtig eine klare und gut durchdachte Struktur für die Entwicklung einer Anwendung ist. Durch die Trennung der Verantwortlichkeiten in Controller, Service und Repository konnte ich den Code übersichtlicher und wartbarer gestalten. Die Controller sind ausschliesslich für die Verwaltung der API-Endpunkte zuständig, während die Service-Schicht die gesamte Geschäftslogik enthält. Die Repository-Schicht kümmert sich um die Datenbankzugriffe. Diese Aufteilung hat nicht nur die Lesbarkeit des Codes verbessert, sondern auch das Testen und Debuggen erleichtert.

Ein weiterer wichtiger Aspekt war die Verwendung von Docker und Maven. Docker hat mir geholfen, die Anwendung in einer konsistenten Umgebung zu entwickeln und zu deployen. Durch die Definition der Container in der Docker-compose.yml-Datei konnte ich sicherstellen, dass die Anwendung lokal genauso läuft wie in der Live Umgebung. Maven hat den Build-Prozess vereinfacht.

Mit der OpenAPI-Dokumentation habe ich sichergestellt, dass die API gut beschrieben und fehlerfrei ist. Durch die automatisch generierte Dokumentation konnte ich die API-Endpunkte klar und verständlich darstellen.

Zusätzlich habe ich durch die Erstellung von Tests, sowohl für die Geschäftslogik als auch für die API-Endpunkte, die Zuverlässigkeit der Anwendung sichergestellt. Unit-Tests und Integration-Tests haben mir geholfen, Fehler frühzeitig zu erkennen und die Stabilität der Anwendung zu erhöhen.

Insgesamt hat mir dieses Projekt gezeigt, wie wichtig eine gute Planung, eine klare Struktur und die Nutzung moderner Tools wie Docker, Maven und OpenAPI sind. Es hat nicht nur meine technischen Fähigkeiten erweitert.

8. Quellen

- Spring Boot Dokumentation: https://spring.io/projects/spring-boot
- Docker Dokumentation: https://docs.docker.com/get-started/
- Spring Security Dokumentation: https://spring.io/projects/spring-security
- OpenAPI Dokumentation: https://swagger.io/specification/