

Spring Boot

ein Java DI Framework

Informatik – 3. Semester

1 Motivation & Rahmen

- 3 Dependency Injection & Beans
- 5 Spring MVC Controller
- 7 Datenbankzugriff über ORMs
- 9 Spring Security & Sessions
- 2 Spring Ökosystem
- 4 Toolchain & Quickstart
- 6 Thymeleaf Templates
- 8 JPA & Spring Data

Warum Spring?

- Ihr kennt bereits HTML, CSS, JS, TS & DI
- Letzte Woche: REST und Web-APIs, also die Schnittstelle zwischen Front- und Back-End
- Fokus verschiebt sich nun auf serverseitige Umsetzung für das Teamprojekt
- Gemeinsames Vokabular für Backend-Architektur schafft die Basis für die Umsetzung in Phase 4

- Geht gut mit Java-Kenntnissen aus INF1 / INF2
- Bietet meinungsstarkes Setup (Konvention vor Konfiguration) und minimiert Boilerplate
- Riesiges Ökosystem mit Starter-Modulen, Guides und Community-Support

- Welche Kriterien habt ihr bei bisherigen Projekten genutzt?
- Passt ein monolithischer Ansatz für das kommende Teamprojekt?
- Welche Risiken seht ihr, wenn jedes Team Mitglied ein anderes Framework nutzt?

- 1 Motivation & Rahmen
- 3 Dependency Injection & Beans
- 5 Spring MVC Controller
- 7 Datenbankzugriff über ORMs
- 9 Spring Security & Sessions

2 Spring Ökosystem

- 4 Toolchain & Quickstart
- 6 Thymeleaf Templates
- 8 JPA & Spring Data

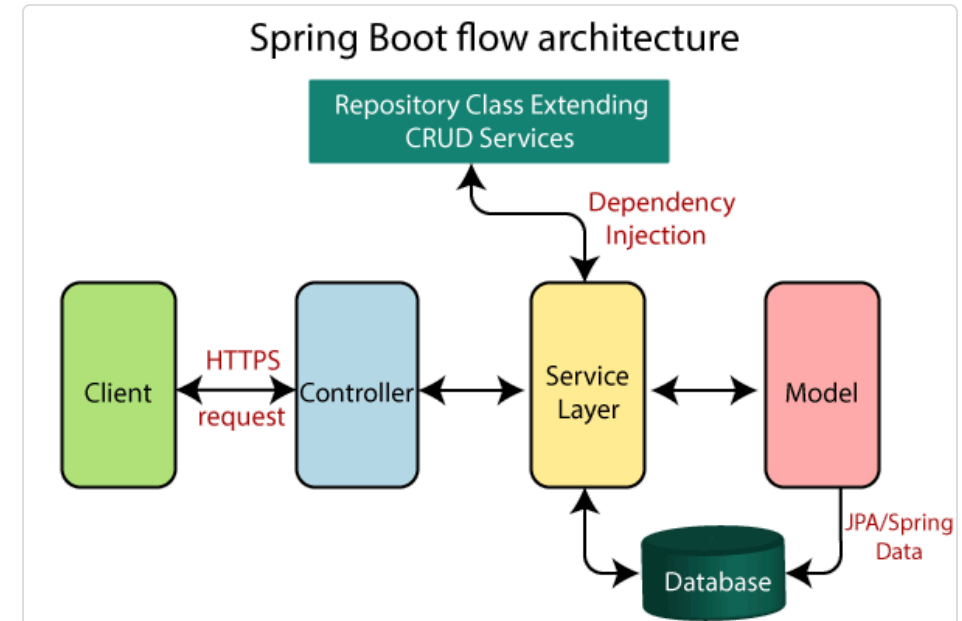
Was gibt es alles so?

- Alternative Frameworks: Jakarta EE, Micronaut, Quarkus, Play, Vaadin
- Spring ist de-facto-Standard im Enterprise/Cloud Umfeld
- Stärke: Modularer Aufbau + grosse Auswahl an Community Projekten

- Spring Core liefert IoC-Container, Bean-Verwaltung, SpEL
- Zusatzmodule für Data Access (JDBC, JPA, Hibernate), Web (MVC), AOP
- Spring Boot als Glue-Schicht für Konfiguration, Auto-Setup, Starter-Abhängigkeiten

- ApplicationContext verwaltet Beans inklusive Instanziierung, Scopes und Lifecycle-Hooks
- Bean-Definitionen lassen sich über Annotationen konfigurieren und vom Container auflösen
- SpEL (Spring Expression Language) ermöglicht dynamische Werte in Konfigurationen und Annotationen
- Kernmodul bildet die Basis für spätere Erweiterungen wie Data, Web oder Security

- Web-Schicht: Controller, Views, REST-Endpoints
- Service-Schicht: Business-Logik, Transaktionen
- Data-Schicht: Repositories, Persistence-Abstraktionen



- Viele Anwendungen brauchen Logging, Tracing, Security Checks
- Aspektorientierung kapselt solche Funktionen in Wiederverwendungsbausteinen
- Pointcuts beschreiben, wo Aspekte greifen; Advice (before/after) definiert Verhalten

1 Motivation & Rahmen

3 Dependency Injection & Beans

5 Spring MVC Controller

7 Datenbankzugriff über ORMs

9 Spring Security & Sessions

2 Spring Ökosystem

4 Toolchain & Quickstart

6 Thymeleaf Templates

8 JPA & Spring Data

Dependency Injection & Beans

- Objekte erzeugen ihre Abhängigkeiten nicht selbst; Container uebernimmt das
- Führt zu lose gekoppelten Komponenten und erleichtert Unit Tests
- Ermöglicht austauschbare Implementierungen (z.B. Mock vs echte Datenbank)

Statt „ich hole mir, was ich brauche“ -> „ich bekomme, was ich brauche“

- Constructor Injection erzwingt komplette Abhängigkeiten, ideal für Immutability
- Setter Injection erlaubt optionale Abhängigkeiten und späte Bindungen
- In Spring können beide gemischt auftreten, Constructor Injection ist bevorzugt

```
class UserRegistrationService {  
    private emailPort!: EmailPort;  
  
    setEmailPort(emailPort: EmailPort) {  
        this.emailPort = emailPort;  
    }  
  
    register(email: string) {  
        // ... User speichern  
        this.emailPort.send(email, 'Willkommen!');  
    }  
}
```

- Definition via Annotationen (@Component, @Service, @Repository) oder Config-Klassen
- Container instanziiert, verwaltet Scope und sorgt für Cleanup
- Hooks wie @PostConstruct / @PreDestroy für Initialisierung und Aufräumen

```
@Service
public class GreetingService {
    private final GreetingRepository repo;

    public GreetingService(GreetingRepository repo) {
        this.repo = repo; // Constructor Injection
    }

    public String greet(String name) {
        return repo.findTemplate().formatted(name);
    }
}
```

- Welche Komponenten in eurem aktuellen Projekt sollten als Interface modelliert werden?
- Gebt 2 Beispiele, bei denen ein Austausch (z.B. andere Datenquelle) realistisch ist
- Diskutiert in 2er-Teams, wie ihr diese Abhängigkeiten injizieren würdet

2 Min

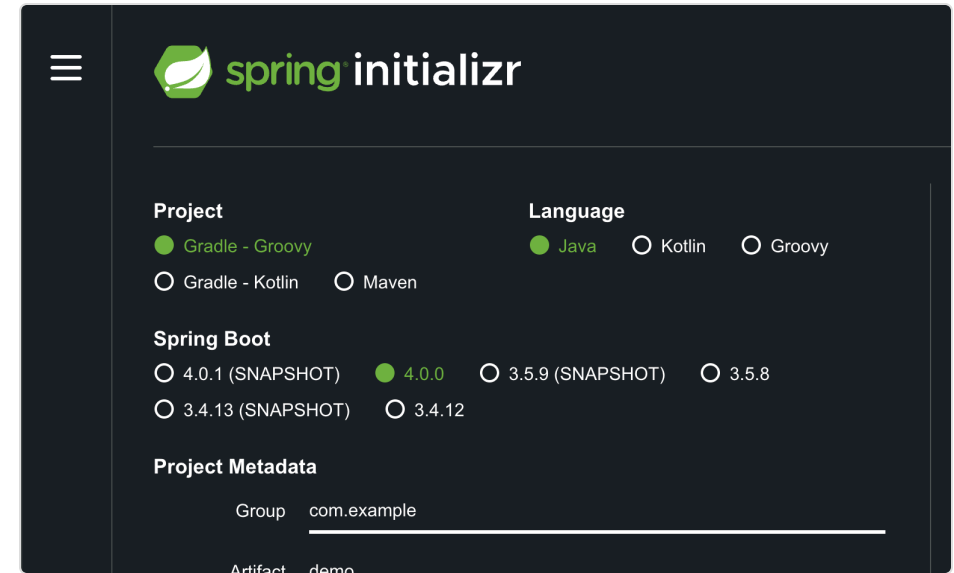
- 1 Motivation & Rahmen
- 3 Dependency Injection & Beans
- 5 Spring MVC Controller
- 7 Datenbankzugriff über ORMs
- 9 Spring Security & Sessions
- 2 Spring Ökosystem
- 4 Toolchain & Quickstart**
- 6 Thymeleaf Templates
- 8 JPA & Spring Data

Toolchain & Quickstart

- Java 21 (oder aktuelles LTS), Git für Versionskontrolle
- Maven oder Gradle als Build-Werkzeuge für Compile/Test/Package
- IDE oder Editor (IntelliJ, VS Code) plus Spring Boot DevTools für Hot Reload



- <https://start.spring.io/> aufrufen, Projektmetadaten ausfüllen
- Wichtige Starter: Spring Web, Spring Data JDBC/JPA, Thymeleaf, H2, DevTools
- Projekt als ZIP laden, entpacken, ins Repo importieren



The screenshot shows the Spring Initializr web interface. It features a dark theme with a sidebar on the left containing a hamburger menu icon. The main content area has the 'spring initializr' logo at the top. Below the logo, there are three sections: 'Project', 'Language', and 'Spring Boot'. The 'Project' section has radio buttons for 'Gradle - Groovy' (selected), 'Gradle - Kotlin', and 'Maven'. The 'Language' section has radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'. The 'Spring Boot' section has radio buttons for '4.0.1 (SNAPSHOT)', '4.0.0' (selected), '3.5.9 (SNAPSHOT)', '3.5.8', '3.4.13 (SNAPSHOT)', and '3.4.12'. Below these sections is the 'Project Metadata' section, which includes a 'Group' field with the value 'com.example' and an 'Artifact' field with the value 'demo'.

```
./mvnw spring-boot:run
```

```
# oder Gradle
```

```
./gradlew bootRun
```

- Anwendung startet lokal auf <http://localhost:8080>
- Live-Reload ueber DevTools spart Neustarts

- Gerne auch im Internet recherchieren!
- Welcher Starter deckt Template-Rendering ab?
- Welche DB nutzt ihr für Prototypen? (H2, PostgreSQL, ...)
- Plant zwei Profil-Konfigurationen (dev vs prod) für später

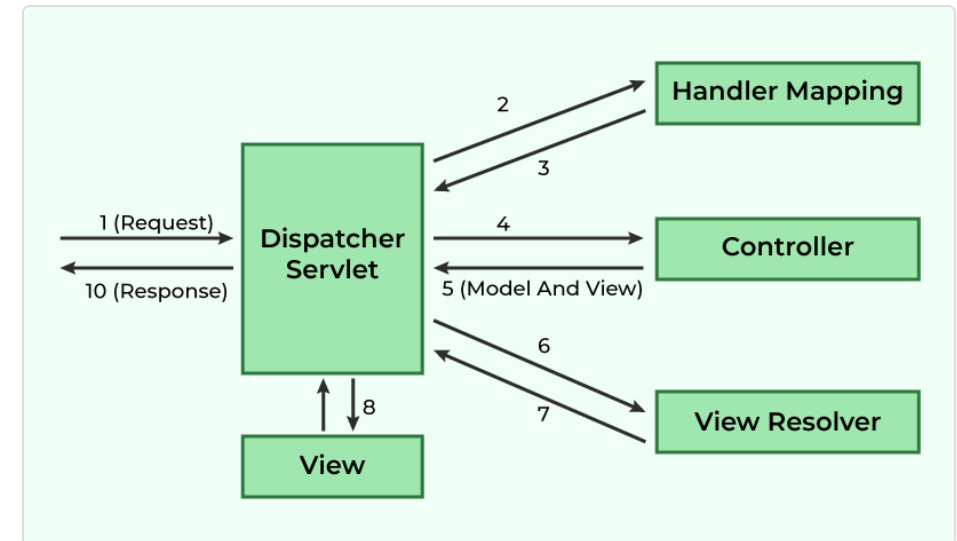
3 Min

- 1 Motivation & Rahmen
- 3 Dependency Injection & Beans
- 5 Spring MVC Controller**
- 7 Datenbankzugriff über ORMs
- 9 Spring Security & Sessions
- 2 Spring Ökosystem
- 4 Toolchain & Quickstart
- 6 Thymeleaf Templates
- 8 JPA & Spring Data

MVC = Model View Control

- Model enthält Attribute, die in der View verfügbar sind
- Rückgabe ist String (Template-Name) oder View-Objekt bzw. ResponseEntity
- Achtet auf Namensgleichheit zwischen Rückgabewert und Template-Datei

- DispatcherServlet routet jede HTTP-Anfrage an passende Handler
- Handler gibt View-Namen oder ResponseEntity zurück



- @GetMapping, @PostMapping, @RequestMapping für Pfade und HTTP-Methoden
- @RequestParam für Query-Parameter, optional mit defaultValue
- @PathVariable für Segmente im Pfad, @RequestBody für JSON/Form-Daten

```
@Controller
public class GreetingController {
    @GetMapping("/greeting")
    public String greeting(@RequestParam(defaultValue = "World") String name,
        Model model) {
        model.addAttribute("name", name);
        return "greeting"; // verweist auf templates/greeting.html
    }
}
```

- Spring Web kann klassische HTML-Views (JSP, Thymeleaf) oder reine JSON-Antworten liefern
- Gleiche Controller lassen sich über `@ResponseBody` oder `@RestController` zu REST-APIs ausbauen
- Sessions, Rechte und Rollenverwaltung sind über Spring MVC + Security kombinierbar
- Entscheidet früh, ob ihr Templates, APIs oder hybride Szenarien braucht, um UI-Team zu entkoppeln

- Definiert die Pfade für "Team-Board anzeigen" (GET) und "Task anlegen" (POST)
- Welche Parameter braucht ihr jeweils?
- Bonus: Wie würdet ihr Validierungsfehler zurückgeben?

3 Min

```
@Controller
@RequestMapping("/team-board")
class TeamBoardController {
    @GetMapping
    String showBoard(Model model) {
        model.addAttribute("members", teamService.loadMembers());
        model.addAttribute("taskForm", new TaskForm());
        return "team-board";
    }

    @PostMapping("/tasks")
    String addTask(@Valid TaskForm taskForm, BindingResult errors, RedirectAttributes flash) {
        if (errors.hasErrors()) {
            return "team-board";
        }
        boardService.createTask(taskForm);
        flash.addFlashAttribute("message", "Task angelegt");
        return "redirect:/team-board";
    }
}
```

- 1 Motivation & Rahmen
- 3 Dependency Injection & Beans
- 5 Spring MVC Controller
- 7 Datenbankzugriff über ORMs
- 9 Spring Security & Sessions
- 2 Spring Ökosystem
- 4 Toolchain & Quickstart
- 6 Thymeleaf Templates**
- 8 JPA & Spring Data

HTML Templating

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <body>
    <p th:text="'Hello, ' + ${name} + '!'" />
  </body>
</html>
```

- Templates liegen unter src/main/resources/templates
- Server rendert HTML vor dem Senden an den Client

- `${book.author.name}` greift auf verschachtelte Eigenschaften zu
- `th:object` definiert Kontext, `*{title}` referenziert aktuelles Objekt
- Eigene Variablen via `th:with` oder Schleifenvariable in `th:each`

- `#{main.title}` nutzt MessageSource für i18n-Texte
- `@{/order/details(id=${orderId})}` erzeugt interne Links inkl. Parameter
- `~{fragments :: header}` bindet wiederverwendbare Markup-Snippets ein

```
<!-- fragments.html -->
<div th:fragment="main">
  <header>Team Board</header>
  <div th:insert="~{::body}"></div>
</div>

<!-- general.html -->
<body>
  <div th:insert="~{fragments :: main}">Fallback Content</div>
</body>
```

- Fragmente helfen, Header/Footer konsistent zu halten und Duplikate zu vermeiden
- th:insert oder th:replace entscheidet, ob bestehende Knoten erhalten bleiben
- Übergeben Variablen an Fragmente via th:with, um dynamische Inhalte zu ermöglichen

- Unterstützt Strings in ", Zahlen, true/false, null
- Operatoren: Text (+, | |), arithmetisch (+,-,*,/,%), boolsch (and, or, not)
- Ternary: condition ? valueIfTrue : valueIfFalse

- `th:text` ersetzt Textknoten, `th:utext` ermöglicht HTML
- `th:each` iteriert Listen, `th:if` / `th:unless` für Bedingungen
- Formulare: `th:action`, `th:value`, `th:field` für Bindung an Model-Attribute

- Modell liefert `List<TeamMember> members`
- Schreibe ein `<ul>` mit `th:each="member : ${members}"`
- Gib Name und Rolle je Person aus; optional Link auf Profil

```
<ul>
  <li th:each="member : ${members}">
    <a th:href="@{/profiles/{id}(id=${member.id})}" th:text="${member.name}"></a>
    <span th:text="${member.role}"></span>
  </li>
</ul>
```

- 1 Motivation & Rahmen
- 3 Dependency Injection & Beans
- 5 Spring MVC Controller
- 7 Datenbankzugriff über ORMs**
- 9 Spring Security & Sessions
- 2 Spring Ökosystem
- 4 Toolchain & Quickstart
- 6 Thymeleaf Templates
- 8 JPA & Spring Data

Datenbankzugriff über ORMs

- Relationale DBs speichern Tabellen + Fremdschlüssel, OO-Code kennt Vererbung/Assoziationen
- Mapping zwischen beiden Welten führt zu Boilerplate, wenn man reines JDBC nutzt
- Strategien: Table per Class, Single Table, Joined Subclass

- Direkter JDBC-Zugriff erfordert SQL-Strings, Connection/Statement-Handling und Fehlerbehandlung
- ResultSet -> Objekt Mapping muss für jede Tabelle von Hand gepflegt werden
- Transaktionen, Ressourcen-Freigabe und Schema-änderungen führen schnell zu Fragilität

- SQL wird hart codiert; Schema-änderungen erzeugen manuellen Aufwand
- Mapping von ResultSet -> Objekt ist wiederkehrend und fehleranfällig
- Welche Probleme habt ihr bereits erlebt?

- DAO kapselt DB-spezifische Zugriffe, stellt CRUD-Methoden bereit
- Kennt Verbindung, Queries, Transaktionen
- Bsp. `accountDao.create(account)` aus ORMLite Code

```
String databaseUrl = "jdbc:h2:mem:account";
ConnectionSource connectionSource = new JdbcConnectionSource(databaseUrl);
Dao<Account, String> accountDao = DaoManager.createDao(connectionSource, Account.class);

Account account = new Account();
account.setName("Jim Coakley");
accountDao.create(account);
```

- Library übernimmt SQL-Generierung und Mapping, dennoch bleibt man nah an JDBC dran
- Zeigt, warum vollwertige ORM-Frameworks wie JPA/Hibernate noch mehr Komfort liefern

- Repository repräsentiert Sammlung von Domain-Objekten
- Fokus auf Fachsprache (findActiveOrders) statt technische Details
- Spring Data stellt Standardmethoden automatisch bereit

- Welche Aufgaben gehören in Service, welche in Repository?
- Identifiziert 2 Methoden, die eher Business-Logik als DB-Zugriff sind

2 Min

- 1 Motivation & Rahmen
- 3 Dependency Injection & Beans
- 5 Spring MVC Controller
- 7 Datenbankzugriff über ORMs
- 9 Spring Security & Sessions
- 2 Spring Ökosystem
- 4 Toolchain & Quickstart
- 6 Thymeleaf Templates
- 8 JPA & Spring Data**

Java Persistence API

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    @OneToMany(mappedBy = "customer")
    private List<Address> addresses;
}
```

- @Entity mappt Klasse auf Tabelle, Felder auf Spalten
- @Transient schliesst Felder vom Mapping aus

- `interface CustomerRepository extends JpaRepository<Customer, Long>`
- Erbt CRUD-Methoden (save, findAll, delete)
- PagingAndSortingRepository oder CrudRepository für spezielle Fälle

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
    <S extends T> S save(S entity);  
    Optional<T> findById(ID id);  
    Iterable<T> findAll();  
    void deleteById(ID id);  
    long count();  
}
```

- Generische Signaturen vermeiden Casting und geben klar vor, was ein Repository leisten muss
- Spezialisierte Interfaces wie JpaRepository bauen darauf auf und ergänzen Paging/Sorting

- findByUsername, findByLastnameOrderByCreatedAtAsc usw.
- Kombiniert Operatoren (And, Or, Between, Like, IgnoreCase)
- Spart Boilerplate, solange Query simpel ist

```
@Query("SELECT u FROM User u WHERE u.status = 1")  
List<User> findAllActiveUsers();  
  
@Query(value = "SELECT * FROM users u WHERE u.status = 1", nativeQuery = true)  
List<User> findAllActiveUsersNative();
```

- JPQL bevorzugt, native SQL für Spezialfälle

```
@Entity
@NamedQueries({
    @NamedQuery(name = "findAllRecipes", query = "SELECT r FROM Recipe r"),
    @NamedQuery(name = "findRecipeById", query = "SELECT r FROM Recipe r WHERE r.id = :id")
})
public class Recipe {}

List<Recipe> recipes = entityManager.createNamedQuery("findAllRecipes", Recipe.class).getResultList();
```

- Wiederverwendbare Abfragen bleiben zentral und lassen sich z.B. für Reporting nutzen
- Repository-Methoden können die Named Queries automatisch aufrufen, wenn der Name passt

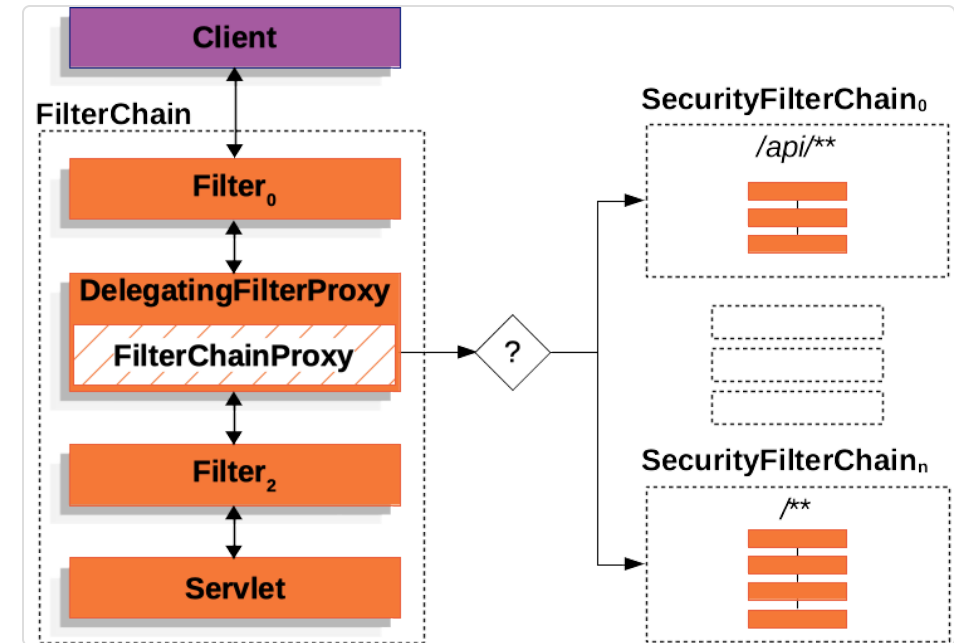
Erleichtert Code-Reviews, weil Query-Definitionen nicht über Klassen verteilt sind

- 1 Motivation & Rahmen
- 3 Dependency Injection & Beans
- 5 Spring MVC Controller
- 7 Datenbankzugriff über ORMs
- 9 Spring Security & Sessions**
- 2 Spring Ökosystem
- 4 Toolchain & Quickstart
- 6 Thymeleaf Templates
- 8 JPA & Spring Data

Spring Security & Sessions

- Schutzbedarf: Authentifizierung, Autorisierung, Session-Management
- Unterstützt HTTP Basic, Form Login, OAuth2, LDAP usw.
- Session-Handling, Token, Password Encoding integriert

- Filter verarbeiten Requests sequenziell; Chain entscheidet über Zugriff
- Anpassbar ueber SecurityFilterChain Bean



- CSRF-Schutz standardmässig aktiv (sollte nur für stateless APIs deaktiviert werden)
- Sessions benötigen sichere IDs (z.B. JSESSIONID) und Timeouts
- Best Practices: invalidate session on login/logout, erneuere Tokens nach Rollenwechsel

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(auth -> auth
            .requestMatchers("/css/**", "/images/**").permitAll()
            .anyRequest().authenticated())
            .formLogin().and()
            .logout();
        return http.build();
    }
}
```

- `@EnableGlobalMethodSecurity` oder `@EnableMethodSecurity` aktiviert `@Secured`, `@RolesAllowed`, `@PreAuthorize`
- Fein-granulare Kontrolle auf Service-Ebene, z.B. `@PreAuthorize("hasRole('ADMIN')")`
- Ideal für Use-Cases, bei denen URL-basierte Regeln nicht reichen

- Welche Endpunkte sollen öffentlich bleiben?
- Welche Rollen braucht ihr mindestens für euer Projekt?
- Skizziert eine SecurityFilterChain für eure API

3 Min

