

Dependency Injection

... oder kurz: DI

Informatik – 3. Semester

1 Warm-up & Kontext

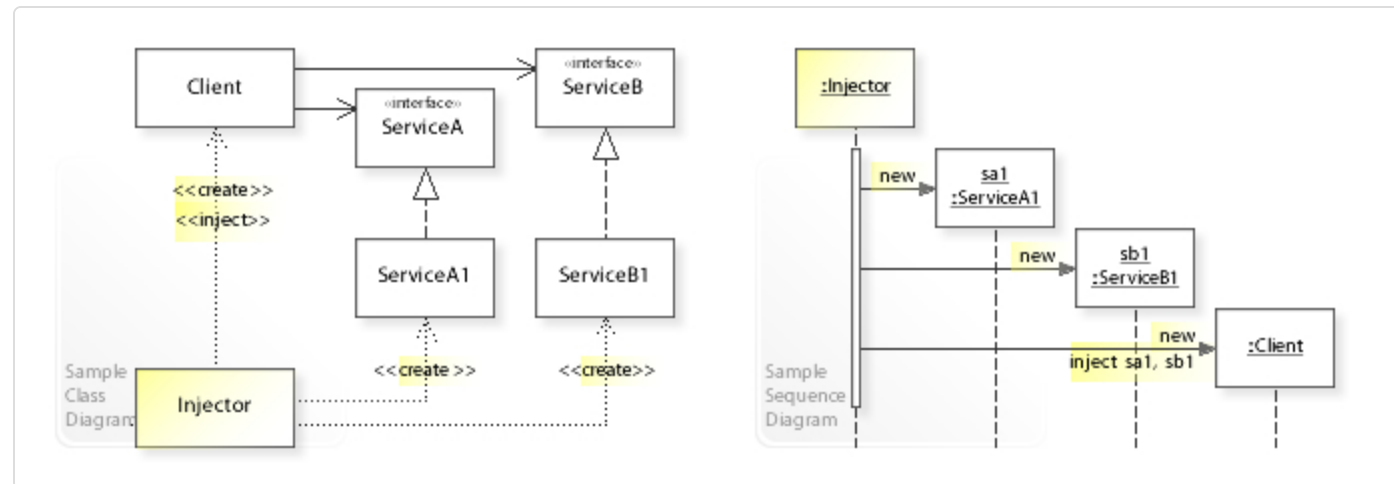
- 3 Begriffe & Grundlagen
- 5 Lebenszyklen & Scopes
- 7 Best Practices & Anti-Patterns
- 9 Spotlight: Spring Boot
- 2 Motivation & Problemstellung
- 4 Formen der Injection
- 6 DI-Container
- 8 Mini-Übungen
- 10 Ausblick & Takeaways

Warum DI?

- Blick zurück: HTML/CSS/JS/TS – was könnt ihr schon einsetzen?
- Semesterziel: MVP einer Webanwendung mit HTML, CSS, JS/TS und Spring Boot im Backend
- Heute legen wir das Fundament für sauberes Backend-Design



- Spontane Implementierungen führen oft zu enger Kopplung
- Testen wird schwer, wenn Services sich gegenseitig bauen
- Teamarbeit leidet, weil jeder andere Initialisierungspfade nutzt
- DI adressiert genau diese Probleme und schafft klare Verträge



- Habt ihr bereits DI in Projekten eingesetzt? (z.B. Angular, Nest, Spring)
- Wer hatte schon Probleme mit Mocking/Testing wegen harter Kopplung?
- Welche Sprache setzt ihr aktuell in Backends ein?

- Eure Projekt-Backends entstehen mit Spring Boot, und Spring lebt von DI
- Gemeinsame Patterns erleichtern Abstimmung zwischen Frontend/Backend-Teams
- Frühzeitige DI-Disziplin verhindert spätere Refactor-Aufwände im Projekt

- 1 Warm-up & Kontext
- 3 Begriffe & Grundlagen
- 5 Lebenszyklen & Scopes
- 7 Best Practices & Anti-Patterns
- 9 Spotlight: Spring Boot

2 Motivation & Problemstellung

- 4 Formen der Injection
- 6 DI-Container
- 8 Mini-Übungen
- 10 Ausblick & Takeaways

Motivation und Problemstellung

- Typische spontane Lösung: alles in einer Datei / Klasse
- Klassen erzeugen ihre Abhängigkeiten direkt mit `new`
- Globale Singletons / statische Helper überall verteilt
- Funktioniert am Anfang – wird mit wachsender Codebasis schnell unübersichtlich

Nicht zu empfehlen!

```
class EmailService {  
    sendWelcomeMail(address: string) {  
        console.log(`Sende Mail an ${address}`);  
    }  
}  
  
class UserRegistration {  
    private emailService = new EmailService();  
  
    register(email: string) {  
        // ... User in DB speichern  
        this.emailService.sendWelcomeMail(email);  
    }  
}
```

- `UserRegistration` entscheidet selbst, welche Implementierung von `EmailService` genutzt wird
- Austausch (z.B. Fake-/Testservice) ist nur über Codeänderung möglich

- **Wartbarkeit:** Jede neue Variante (z.B. anderer Mailanbieter) erfordert Änderungen an vielen Stellen
- **Testbarkeit:** Unit-Tests brauchen „echte“ Abhängigkeiten oder aufwändige Workarounds
- **Wiederverwendung:** Klassen sind schwer in anderen Kontexten nutzbar

Versteckte Abhängigkeiten: Konstruktion steckt im Code, nicht in der Schnittstelle

- Wie würdet ihr `UserRegistration` testen, ohne wirklich E-Mails zu versenden?
- Wo müsstet ihr aktuell Änderungen vornehmen, um einen Fake-Service zu benutzen?
- Welche Risiken seht ihr, wenn „echte“ Infrastruktur immer mitläuft (Mail, DB)?

```
// Pseudocode für einen Test ohne DI
it("soll Willkommen-Mail versenden", () => {
  const registration = new UserRegistration();
  registration.register("test@example.com");
  // Wie prüfen wir, ob eine Mail verschickt wurde?
  // Logfile parsen? DB abfragen? externen Service mocken?
});
```

- Test kennt keine saubere Möglichkeit, Verhalten zu beobachten
- Oft werden globale Flags, statische Mocks oder komplizierte Test-Setups gebaut
- Ergebnis: Tests sind brüchig und werden „nervig“ → weniger Tests

- Abhängigkeiten explizit machen statt versteckt erzeugen
- Erlaubt Austauschbarkeit (z.B. Fake vs. echter Service) ohne Codeänderungen an der Fachlogik
- Unterstützt saubere Schichtenarchitektur und klarere Verantwortlichkeiten
- Grundlage für gute Testbarkeit, Wartbarkeit und Teamarbeit im Projekt

1 Warm-up & Kontext

3 Begriffe & Grundlagen

5 Lebenszyklen & Scopes

7 Best Practices & Anti-Patterns

9 Spotlight: Spring Boot

2 Motivation & Problemstellung

4 Formen der Injection

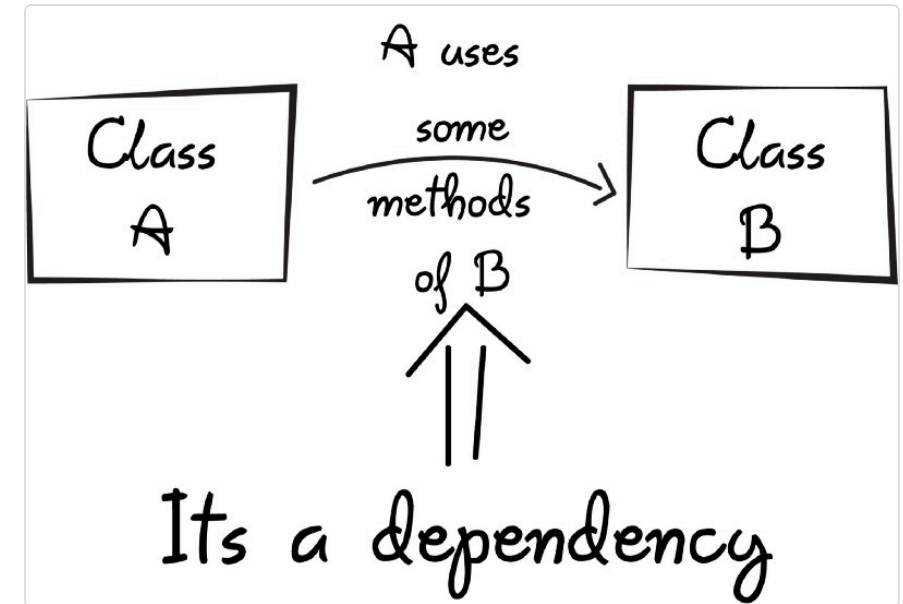
6 DI-Container

8 Mini-Übungen

10 Ausblick & Takeaways

Dependency, IoC, DIP, ...

- Eine Dependency ist alles, was eine Klasse/Funktion zum Arbeiten braucht
- Beispiele: Logger, Datenbankzugriff, E-Mail-Service, Konfiguration
- Wichtig: Fachlogik soll nicht wissen, wie die Dependency gebaut wird



Klassisch

- Eine Klasse kontrolliert selbst, welche konkreten Abhängigkeiten sie erstellt

IoC

- Die Kontrolle wird „umgedreht“ – etwas anderes liefert ihr die Abhängigkeiten
- DI ist eine konkrete Art, IoC umzusetzen

Statt „ich hole mir, was ich brauche“ → „ich bekomme, was ich brauche“

„High-level modules should not depend on low-level modules. Both should depend on abstractions.“

„Abstractions should not depend on details. Details should depend on abstractions.“

- Fachlogik hängt von Interfaces/Typen ab, nicht von konkreten Implementierungen
- DI-Container helfen, diese Abstraktionen zur Laufzeit mit konkreten Objekten zu verknüpfen

```
interface EmailPort {  
    send(address: string, body: string): void;  
}  
  
class UserRegistrationService {  
    constructor(private emailPort: EmailPort) {}  
  
    register(email: string) {  
        // ... User in DB speichern  
        this.emailPort.send(email, "Willkommen!");  
    }  
}
```

- `UserRegistrationService` kennt nur das Interface `EmailPort`, nicht die konkrete Implementierung
- Später können wir z.B. `SmtEmailService` oder `FakeEmailService` injizieren

Service Locator

- Klasse fragt zentralen „Locator“ nach Abhängigkeiten
(`ServiceLocator.get('EmailService')`)
- Problem: versteckte Abhängigkeiten, schwerer zu testen

Dependency Injection

- Abhängigkeiten werden explizit über Konstruktor/Methoden übergeben
- Vorteil: Signatur zeigt klar, was zum Arbeiten benötigt wird

Welchen Usecase könnte ein Service Locator haben?

- Welche Vorteile hat es, wenn eine Klasse nur von Interfaces (Abstraktionen) abhängt?
- Woran erkenne ich im Code, ob etwas eher Service Locator oder DI ist?
- Wo habt ihr in eigenen Projekten bereits IoC unbewusst eingesetzt?

- 1 Warm-up & Kontext
- 3 Begriffe & Grundlagen
- 5 Lebenszyklen & Scopes
- 7 Best Practices & Anti-Patterns
- 9 Spotlight: Spring Boot
- 2 Motivation & Problemstellung
- 4 Formen der Injection**
- 6 DI-Container
- 8 Mini-Übungen
- 10 Ausblick & Takeaways

Konstruktor-, Setter-, Interface-Injection

Constructor Injection

Setter Injection

Interface (oder Method) Injection

- Alle verfolgen dasselbe Ziel: Abhängigkeiten von außen zuführen
- Unterschied: Wann und wie wird die Abhängigkeit gesetzt?

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

- Abhängigkeiten sind im Konstruktor sofort sichtbar
- Objekt ist nach Konstruktion vollständig benutzbar („fully initialized“)
- Gut für Pflicht-Abhängigkeiten

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

- Abhängigkeit kann nachträglich gesetzt/gewechselt werden
- Geeignet für optionale oder zur Laufzeit wechselbare Dependencies
- Risiko: Objekt kann in ungültigem Zustand sein, wenn Setter nicht aufgerufen wird

```
interface EmailAware {  
    injectEmailPort(emailPort: EmailPort): void;  
}  
  
class UserRegistrationService implements EmailAware {  
    private emailPort!: EmailPort;  
  
    injectEmailPort(emailPort: EmailPort) {  
        this.emailPort = emailPort;  
    }  
}
```

- Klasse definiert explizit eine „Injection-Methode“ über ein Interface
- Wird in manchen Frameworks genutzt, um bestimmte Hooks zu markieren
- Im reinen Anwendungscode eher selten nötig

	Vorteile	Nachteile
Konstruktor Injection	Klarheit, Immutabilität, Testbarkeit	mehr Parameter bei vielen Dependencies
Setter Injection	Flexibilität	Gefahr unvollständiger Objekte
Interface Injection	explizite Markierung	zusätzliche Komplexität, Framework-Lastigkeit

In vielen Projekten: Constructor Injection als Standard, Setter/Interface nur bei speziellen Fällen

- Welche Injection-Form würdet ihr für einen `Logger` wählen? Warum?
- Und für einen `FeatureToggleService`, dessen Implementierung zur Laufzeit wechseln können soll?

2 Min

- 1 Warm-up & Kontext
- 3 Begriffe & Grundlagen
- 5 Lebenszyklen & Scopes**
- 7 Best Practices & Anti-Patterns
- 9 Spotlight: Spring Boot
- 2 Motivation & Problemstellung
- 4 Formen der Injection
- 6 DI-Container
- 8 Mini-Übungen
- 10 Ausblick & Takeaways

Lebenszyklen und Scopes

- Gleiche Klasse kann in verschiedenen Kontexten unterschiedliche Lebensdauer haben
 - Konfiguration (eher einmalig)
 - HTTP-Request (kurzlebig)
 - Datenbankverbindung (wiederverwendet)
- Falscher Scope führt zu Bugs
 - geteilte Zustände
 - Memory-Leaks
 - Performance-Probleme

Transient

Neue Instanz bei jeder Anforderung
(Injection)

Singleton

Eine Instanz pro Anwendung / Container

Request Scoped

Eine Instanz pro HTTP-Request

Weitere mögliche Scopes: Session, Thread, Job, Modul etc. (frameworkabhängig)

```
class ConfigService {  
    constructor(private readonly config: Record<string, string>) {}  
  
    get(key: string) {  
        return this.config[key];  
    }  
}
```

// Idee: ConfigService einmal erstellen und überall wiederverwenden

- Sinnvoll für zustandsarme, teure oder globale Ressourcen (z.B. Konfiguration, Logger)
- Achtung bei mutablem Zustand: geteilte Daten über Threads/Requests hinweg

Transient

- Jedes Mal eine neue Instanz → gut für kurzlebige, zustandsbehaftete Objekte
- Beispiel: Builder-Objekte, temporäre Berechnungen

Request Scoped

- Eine Instanz pro HTTP-Request → teilt sich Zustand innerhalb der Request-Verarbeitung
- Beispiel: `currentUserContext` , `RequestMetrics`

- Welche Scope-Wahl (Singleton / Request / Transient / Anderes) erscheint sinnvoll für:
 - `DatabaseConnectionPool`
 - `ShoppingCart` in einem klassischen Webshop
 - `FeatureFlagCache` , das periodisch aktualisiert wird

2 Min

- 1 Warm-up & Kontext
- 3 Begriffe & Grundlagen
- 5 Lebenszyklen & Scopes
- 7 Best Practices & Anti-Patterns
- 9 Spotlight: Spring Boot
- 2 Motivation & Problemstellung
- 4 Formen der Injection
- 6 DI-Container**
- 8 Mini-Übungen
- 10 Ausblick & Takeaways

Allgemeine Definition

- Verwaltet Registrierungen von „welcher Typ liefert welche Abhängigkeit?“
- Erstellt Objekte (inkl. ihrer Dependencies) nach definierten Regeln
- Beachtet Scopes/Lebenszyklen (Singleton, Request, ...)
- Löst Abhängigkeiten auf, ohne dass Aufrufer `new` schreiben müssen

- **Registrierung:** „Wenn jemand `EmailPort` braucht, nimm `SmtplibService`“
- **Auflösung:** „Gib mir eine Instanz von `UserRegistrationService`“
- **Konfiguration:** Wie sind Scopes, Parameter, Umgebungen definiert?

Oft zusätzlich: Module/Packages, die zusammen registriert werden

```
type Token<T> = string; // Vereinfachung

class Container {
  private singletons = new Map<Token<any>, any>();
  private factories = new Map<Token<any>, () => any>();

  registerSingleton<T>(token: Token<T>, factory: () => T) {
    this.factories.set(token, factory);
  }

  resolve<T>(token: Token<T>): T {
    if (!this.singletons.has(token)) {
      const instance = this.factories.get(token)!();
      this.singletons.set(token, instance);
    }
    return this.singletons.get(token) as T;
  }
}
```

```
const container = new Container();

container.registerSingleton<EmailPort>(
  "EmailPort",
  () => new SmtplibEmailService(),
);
container.registerSingleton<UserRegistrationService>(
  "UserRegistrationService",
  () => new UserRegistrationService(container.resolve("EmailPort")),
);

const registrationService = container.resolve<UserRegistrationService>(
  "UserRegistrationService",
);
```

- Container kennt, wie `UserRegistrationService` gebaut wird
- Aufrufer brauchen nur noch `resolve` – keine `new`-Ketten mehr im Anwendungscode

- Module bündeln thematisch zusammengehörige Registrierungen (z.B. „User“, „Billing“, „Auth“)
- Konfigurationsquellen:
 - Umgebungsvariablen (`NODE_ENV` , DB-URL, ...)
 - Konfigurationsdateien (JSON, YAML, .env)
 - Feature Flags
- Container/Framework liest diese Quellen und entscheidet, welche Implementierungen registriert werden

- Wo seht ihr Vorteile eines zentralen DI-Containers gegenüber „einfach überall `new`“?
- Welche Risiken entstehen, wenn der Container zu „magisch“ wird?
- Wie könnte man Module für euer Semesterprojekt zuschneiden (z.B. User, Board, Notifications)?

- 1 Warm-up & Kontext
- 3 Begriffe & Grundlagen
- 5 Lebenszyklen & Scopes
- 7 Best Practices & Anti-Patterns**
- 9 Spotlight: Spring Boot
- 2 Motivation & Problemstellung
- 4 Formen der Injection
- 6 DI-Container
- 8 Mini-Übungen
- 10 Ausblick & Takeaways

Best Practices und Anti-Patterns

- Abhängigkeiten explizit machen (Konstruktorparameter, Interfaces)
- Constructor Injection als Standard wählen
- Wenige, klar geschnittene Services statt „God Objects/Services“
- DI nutzen, um Testbarkeit zu verbessern, nicht zu verstecken

- Ein Service kennt „alles“: User, Bestellungen, Mails, Logging, ...
- Symptome:
 - Riesige Klasse, viele hundert Zeilen
 - Viele Dependencies im Konstruktor
 - Änderungen an einer Fachlichkeit brechen andere

Besser: Nach Bounded Contexts / Modulen schneiden und aufteilen

- Zu viele Dependencies in einer Klasse (z.B. $> 4 - 5$ Services)
- Hinweis auf fehlende Verantwortlichkeits-Trennung
- Folge: Schwer zu testen, hoher kognitiver Load
- Gegenmittel: Klasse in kleinere Komponenten zerlegen, Schnittstellen nachschärfen

- Statische Singletons / globale Variablen statt sauberer DI
- Erschwert Tests (Zustand bleibt zwischen Tests erhalten)
- Erhöht Kopplung und macht Reihenfolge/Timing relevant
- Empfehlung: Globals vermeiden, explizite Injection bevorzugen

Probleme

- Container kann Graph nicht mehr sauber auflösen
- Fachlich oft ein Hinweis auf schlechte Schnittstellen

Lösung

- Gemeinsame Logik in neuen Service auslagern (z.B. `UserOrderFacade`)
- Events oder Ports/Adapter nutzen

Beispiel: `UserService` braucht `OrderService` , und `OrderService` braucht `UserService`

- Bei optionalen Dependencies: Null Object Pattern statt `null / undefined` prüfen
- Beispiel: `NoopMetricsService`, der nichts tut, aber Interface erfüllt
- Vorteil: Weniger `if (metrics)`-Abfragen, klarere Logik

- 1 Warm-up & Kontext
- 3 Begriffe & Grundlagen
- 5 Lebenszyklen & Scopes
- 7 Best Practices & Anti-Patterns
- 9 Spotlight: Spring Boot
- 2 Motivation & Problemstellung
- 4 Formen der Injection
- 6 DI-Container
- 8 Mini-Übungen**
- 10 Ausblick & Takeaways

Mini-Übungen mit Diskussionen

```
class OrderController {  
  constructor(private readonly db: DbClient) {}  
  
  async createOrder(req: any, res: any) {  
    const userId = req.headers["x-user-id"];  
    const body = req.body;  
    const logger = new ConsoleLogger();  
  
    logger.info("Creating order for", userId);  
    const orderId = await this.db.insert("orders", { userId, body });  
    res.json({ orderId });  
  }  
}
```

- Markiert alle Abhängigkeiten (explizit und implizit)
- Überlegt, welche davon sinnvoll per DI injiziert werden sollten

2 Min

- `ConsoleLogger` soll per DI injiziert werden
 - Interface `Logger` definieren
 - `OrderController` über Konstruktor mit `Logger` versorgen
 - `ConsoleLogger` als konkrete Implementierung im Container/Setup registrieren
- Wie würdet ihr einen `FakeLogger` für Tests einhängen?

2 Min

- Ordnet gemeinsam zu
 - ConfigService → eher Singleton / Transient / Request?
 - RequestContext (aktuell eingeloggter User) → ?
 - DiscountCalculator ohne eigenen Zustand → ?

1 Min

- Was ist der Unterschied zwischen IoC und DI?
- Nenne zwei Vorteile von Constructor Injection gegenüber Setter Injection
- Warum sind globale Singletons oft problematisch für Tests?
- Woran erkennst du im Code, dass ein Service eventuell ein „God Service“ geworden ist?
- Welche Teile eures Semesterprojekts profitieren am stärksten von sauberem DI-Design?
- Wo seht ihr aktuell potenzielle Problemstellen (z.B. enge Kopplung, globale States)?

- 1 Warm-up & Kontext
- 3 Begriffe & Grundlagen
- 5 Lebenszyklen & Scopes
- 7 Best Practices & Anti-Patterns
- 9 Spotlight: Spring Boot**
- 2 Motivation & Problemstellung
- 4 Formen der Injection
- 6 DI-Container
- 8 Mini-Übungen
- 10 Ausblick & Takeaways

Framework-Spotlight: Spring Boot

- Java-/JVM-Framework basierend auf dem Spring-Ökosystem
- Starker Fokus auf Konventionen, Auto-Konfiguration und DI
- Nutzt einen leistungsfähigen DI-Container (Spring Context)
- Auch hier: Nur ein erster Blick auf DI – Details folgen in eigener Spring-Vorlesung

- Spring findet Beans über Annotationen und Package-Scan
- Typische Annotationen:
 - `@Component` – generische Komponente
 - `@Service` – Service-Schicht
 - `@Repository` – Datenzugriffsschicht
 - `@Controller` / `@RestController` – Web-Schicht
- Container registriert diese Beans und macht sie injizierbar

```
// UserService.java
import org.springframework.stereotype.Service;

@Service
public class UserService {
    public List<String> findAll() {
        return List.of("alice", "bob");
    }
}
```

```
// UserController.java
import org.springframework.web.bind
    .annotation.GetMapping;
import org.springframework.web.bind
    .annotation.RestController;

@RestController
public class UserController {

    private final UserService userService;

    // Constructor Injection
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/users")
    public List<String> findAll() {
        return userService.findAll();
    }
}
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MailConfig {

    @Bean
    public MailClient mailClient() {
        return new SmtplibClient("smtp.example.com");
    }
}
```

- `@Configuration` -Klasse definiert Beans explizit
- `@Bean` -Methoden geben Objekte zurück, die im Container registriert werden
- Erinnert an unsere `Container.register(...)` -Beispiele, nur deklarativ

- Spring Profiles erlauben verschiedene Bean-Konfigurationen für unterschiedliche Umgebungen
 - z.B. `dev`, `test`, `prod`
- Beispiel:
 - In `dev`: `FakeMailClient`
 - In `prod`: `SmtplibClient`
- Aktivierung über Properties/Umwgebungsvariablen
- Passt direkt zu unserem Thema „Austauschbare Implementierungen per DI“

- Ebenfalls Constructor Injection als Best Practice
- Spring Context = DI-Container mit Registrierung, Auflösung, Scopes
- Komponenten-Scan und `@Configuration`-Klassen übernehmen die Container-Konfiguration

Detaillierte Themen (Scopes, Lifecycle, Testing mit Spring) folgen in der Spring-Vorlesung

- | | |
|----------------------------------|------------------------------------|
| 1 Warm-up & Kontext | 2 Motivation & Problemstellung |
| 3 Begriffe & Grundlagen | 4 Formen der Injection |
| 5 Lebenszyklen & Scopes | 6 DI-Container |
| 7 Best Practices & Anti-Patterns | 8 Mini-Übungen |
| 9 Spotlight: Spring Boot | 10 Ausblick & Takeaways |

Was in den nächsten Vorlesungen folgt

- DI löst Probleme von enger Kopplung, Testbarkeit und Wartbarkeit
- Kerngedanken: IoC, Dependency Inversion, explizite Abhängigkeiten
- Wichtige Bausteine: Injection-Formen, Scopes, DI-Container
- Spring Boot setzt diese Konzepte in der Praxis um, Angular oder NestJS ebenso

Als nächstes: REST und Web-APIs

- Wie Front-End und Back-End über HTTP und JSON miteinander reden
- Ressourcen, Methoden, Statuscodes, ganz ohne Framework

Danach: Spring Boot, Teil 1 und 2

- Spring Context, Konfiguration, Repositories, REST-Controller
- Testing mit Spring und DI