sepp.med

DAS SPRING FRAMEWORK

Von Teams geschätzt - Von Entwicklern geliebt

Wir machen **DIGITALISIERUNG** – aber **SICHER!**

www.seppmed.de

1. Abstract

Das Spring Framework hat sich als führendes Framework für die Java-Entwicklung etabliert, indem es umfassende Unterstützung für die Erstellung von robusten, skalierbaren und effizienten Anwendungen bietet. Dieser Artikel analysiert die Schlüsselkomponenten und Konzepte von Spring, darunter der Spring Container, *Inversion of Control*, *Dependency Injection* und die Verwaltung von *Spring Managed Beans*. Zudem wird die Bedeutung von Konfigurationsklassen und die Steuerung des *Bean*-Lebenszyklus durch benutzerdefinierte *Init*- und *Destroy*-Methoden beleuchtet. Ziel des Artikels ist es, sowohl erfahrenen Entwicklern als auch Einsteigern ein tiefes Verständnis des Frameworks zu vermitteln und die praktische Anwendung durch veranschaulichende Beispiele zu fördern. Der Artikel erörtert weiterhin die Vorteile des Einsatzes des Spring Containers, wie zentrale Verwaltung von *Beans*, vereinfachte Konfiguration und erhöhte Flexibilität, die insgesamt zu einer saubereren Codebasis und verbesserten Testbarkeit führen.

Inhaltsverzeichnis

2. Einleitung

Das Spring Framework hat sich als eines der führenden Frameworks in der Java-Entwicklung etabliert, und bietet umfangreiche Unterstützung für die Erstellung von robusten, skalierbaren und effizienten Anwendungen. Mit seiner umfassenden Infrastruktur für Konfigurations- und Dependency-Management vereinfacht Spring die Entwicklung komplexer Unternehmensanwendungen und fördert dabei bewährte Programmierpraktiken.

In diesem Artikel werden wir uns eingehend mit den fundamentalen Komponenten und Konzepten des Spring Frameworks beschäftigen. Dazu gehören der Spring Container, der Mechanismus der *Inversion of Control*, die Verwaltung und das Management von *Spring Managed Beans*, die Prinzipien und die Umsetzung von *Dependency Injection*, sowie die Herausforderungen und Lösungsansätze bei zyklischen Abhängigkeiten. Darüber hinaus erörtern wir die Bedeutung der Konfigurationsklassen und die Steuerung des Bean-Lebenszyklus durch benutzerdefinierte Init- und Destroy-Methoden. Diese Themen sind entscheidend für ein tiefes Verständnis des Frameworks und seine effektive Anwendung in der Praxis.

Ziel dieses Artikels ist es, sowohl erfahrenen Entwicklern als auch Einsteigern eine klare und umfassende Darstellung der Kernaspekte von Spring zu bieten. Dabei werden die technischen Details durch praktische Beispiele veranschaulicht, die helfen sollen, die Konzepte in realen Anwendungsszenarien effektiv zu nutzen.

3. Vorteile von Spring Boot

In diesem Kapitel werden die wesentlichen Vorteile von Spring Boot, einem der führenden Java-Frameworks für die schnelle und effiziente Entwicklung von modernen Webanwendungen und Microservices, beleuchtet. Spring Boot zeichnet sich durch seine Fähigkeit aus, den Entwicklungsprozess durch eine Reihe von leistungsstarken Features und Automatisierungen erheblich zu beschleunigen. Von der vereinfachten Konfiguration bis hin zur Unterstützung für Microservices bietet Spring Boot eine robuste Grundlage, die Entwickler nutzen können, um hochwertige, skalierbare und wartbare Anwendungen zu erstellen. Im Folgenden stellen wir die fünf wichtigsten Vorteile vor, die Spring Boot gegenüber anderen Frameworks in der Java-Landschaft auszeichnen.

- **Automatische Konfiguration**: Spring Boot minimiert den Konfigurationsaufwand, indem es auf der Basis der hinzugefügten Bibliotheken automatische Konfigurationen vornimmt. Dies erleichtert den Einstieg und reduziert Boilerplate-Code.
- **Eingebettete Webserver**: Die Möglichkeit, Webserver direkt in die Anwendung einzubetten, vereinfacht das Deployment erheblich, da keine separate Serverkonfiguration benötigt wird.
- **Unabhängigkeit bei der Bereitstellung**: Anwendungen können als eigenständige JAR- oder WAR-Dateien gepackt werden, die alle notwendigen Abhängigkeiten enthalten. Das macht die Bereitstellung und das Ausführen der Anwendungen plattformunabhängig und einfach.
- **Unterstützung für Microservices**: Spring Boot ist besonders gut geeignet für die Entwicklung von Microservices, da es die Entwicklung, das Testing und das Skalieren von unabhängigen Servicekomponenten unterstützt.
- **Umfassende Community und Ressourcen**: Dank der großen und aktiven Community rund um Spring Boot und das gesamte Spring-Ökosystem gibt es eine reiche Auswahl an Ressourcen, Tutorials und Unterstützung, was die Problembehebung und Weiterbildung fördert.

4. Der Spring Container

Der *Spring Container* ist das Herzstück des Spring Frameworks und spielt eine zentrale Rolle bei der Verwaltung der Lebenszyklen von Java-Objekten die als *Beans* bezeichnet werden. Durch den Container werden Objekte instanziiert, konfiguriert und assembliert.

4.1 BeanFactory und ApplicationContext

Innerhalb des *Containers* gibt es zwei Haupttypen von Container Implementierungen, die *BeanFactory* und der *ApplicationContext*. Die *BeanFactory* ist die einfachste Schnittstelle, die für die Grundversorgung mit *Dependency Injection* zuständig ist. Er wird hauptsächlich in Szenarien verwendet, in denen der Ressourcenverbrauch minimal gehalten werden soll, wie bei eingebetteten Systemen oder kleinen Anwendungen.

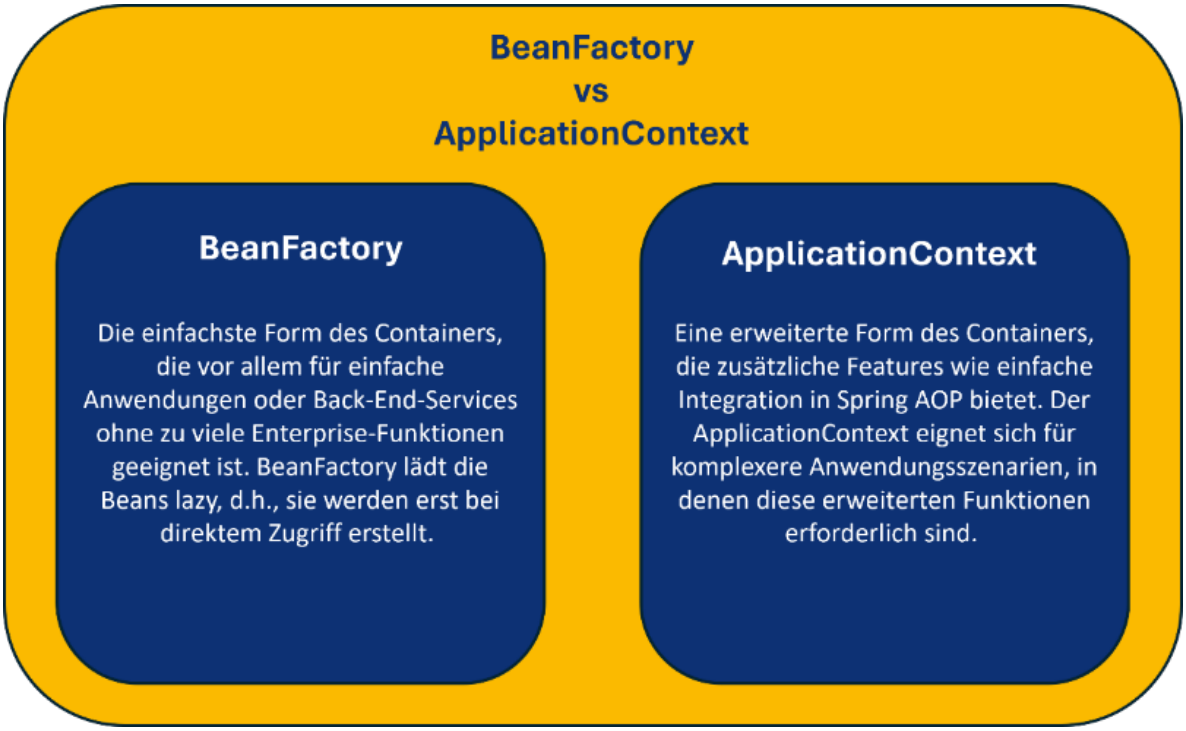

Abbildung 1 Unterschied zwischen der BeanFactory und dem ApplicationContext, selbst erstellt

Im Gegensatz dazu bietet der *ApplicationContext* eine erweiterte Funktionalität. Er ist eine Unterklasse der *BeanFactory* und stellt zusätzliche Features bereit, darunter internationale Nachrichtenunterstützung, Ereignisverbreitung und verschiedene Möglichkeiten zum Laden von Kontextkonfigurationen.

Das Starten eines *Spring Containers* ist typischerweise der erste Schritt beim Hochfahren einer Spring-Anwendung. Dies geschieht durch den Aufruf der *run()* Methode der *SpringApplication*-Klasse, die den *ApplicationContext* initialisiert und die Konfiguration der *Beans* sowie das Auflösen ihrer Abhängigkeiten übernimmt. Dieser Prozess bildet das Rückgrat der Anwendung und stellt sicher, dass alle Komponenten ordnungsgemäß konfiguriert und bereitgestellt sind, um ihre Aufgaben innerhalb der Anwendung zu erfüllen.

```
public static void main(String[] args) {
      ApplicationContext ctx = SpringApplication.run(….);
      for (String name: ctx.getBeanDefinitionNames() {
            System.out.println(name};
      )
}
```

Abbildung 2 Einstiegspunkt, selbst erstellt

Durch die Methode `getBeanDefinitionNames()` werden alle Namen der *Beans* ausgegeben, die im Container vorhanden sind.

4.2 Vorteile der Verwendung eines Spring Containers

Die Verwendung eines Containers bringt zahlreiche Vorteile mit sich. Zunächst ermöglicht er die zentrale Verwaltung von *Beans*, was zu einer einfacheren und effizienteren Codebasis führt. Entwickler können sich auf die Geschäftslogik konzentrieren, während der Container sich um die Erstellung und Verwaltung der Anwendungsobjekte kümmert. Durch diese Entkopplung wird die Anwendung leichter wartbar und skalierbar. Nachfolgend werden die Aufgaben des Containers zusammengefasst:

- **Bean-Erstellung und Management**: Der Container erstellt und verwaltet alle im Anwendungskontext definierten *Beans*. Die *Beans* werden entweder bei der Initialisierung der Anwendung oder bei Bedarf erstellt.
- **Abhängigkeitsinjektion**: Eine weitere Aufgabe des Containers ist die *Abhängigkeitsinjektion* (auch *Dependency Injection* oder *DI*), bei der Objektabhängigkeiten automatisch durch den Container bereitgestellt werden.
- **Bean Lebenszyklusmanagement**: Der Container übernimmt das vollständige Management des Lebenszyklus von Beans, von der Erstellung bis zur Garbage Collection, und verwaltet zudem Hooks, die es ermöglichen, benutzerdefinierte Logiken zu verschiedenen Zeitpunkten im Lebenszyklus einzubinden.
- **Ressourcenverwaltung**: Der Container unterstützt die Lokalisierung und das Laden von Ressourcen, wie z.B. das Bündeln von Nachrichten, was die Entwicklung von Anwendungen erleichtert, die mehrere Sprachen unterstützen.

Schließlich fördert der Einsatz des Containers das Prinzip der *Inversion of Control*, das den Code sauberer und die Modultests einfacher macht. Das ermöglicht es, Abhängigkeiten von außen in eine Klasse einzubringen, anstatt sie innerhalb der Klasse zu erstellen.

Nachdem wir die Grundlagen des Containers sowie die Unterschiede zwischen *BeanFactory* und *ApplicationContext* erörtert haben, konzentrieren wir uns nun auf die von Spring verwalteten *Beans*.

4.3 Inversion of Control

Inversion of Control (IoC) ist ein zentrales Konzept in der Softwareentwicklung, das eng mit dem Spring Framework verbunden ist. *IoC* bezeichnet eine Technik, bei der die Kontrolle über die Objekterstellung und -konfiguration von der Anwendung an einen externen Akteur, in diesem Fall den Container, übergeben wird. Dieser Paradigmenwechsel ist grundlegend für

das Verständnis moderner Anwendungsentwicklung und trägt zur Entkopplung und Modularisierung von Code bei.

4.4 Vorteile von IoC in Spring

- **Entkopplung**: Durch die Übertragung der Kontrolle über Abhängigkeiten an den Container werden Komponenten entkoppelt, was die Wartung und Erweiterung der Anwendung erleichtert.
- **Erhöhte Modulflexibilität**: Komponenten können unabhängig voneinander entwickelt und getestet werden, da sie nicht direkt voneinander abhängen.
- **Einfachere Konfiguration**: Änderungen in der Konfiguration der Abhängigkeiten können leicht durchgeführt werden, ohne den Code zu berühren, was insbesondere in großen Anwendungen und bei der Migration zwischen verschiedenen Umgebungen nützlich ist.
- **Verbesserte Testbarkeit**: Da Abhängigkeiten extern verwaltet und injiziert werden, können Komponenten isoliert und getestet werden, oft unter Verwendung von *Mock-Objekten*, die die realen Abhängigkeiten in Testumgebungen simulieren.

4.5 Praktische Anwendung von IoC

In der praktischen Anwendung ermöglicht *IoC* Entwicklern, sich mehr auf die Geschäftslogik zu konzentrieren, anstatt sich um die Lebenszyklusverwaltung von Objekten oder die Verwaltung von Abhängigkeiten zu kümmern. Ein typisches Beispiel für *IoC* in Aktion ist die Erstellung eines Webanwendungsservices, bei dem Spring automatisch Datenbankverbindungen, Transaktionsmanager und andere Dienste bereitstellt, ohne dass der Entwickler explizit Code zur Initialisierung dieser Komponenten schreiben muss.

5. Konfigurationsmanagement in Spring

Das Konfigurationsmanagement spielt eine entscheidende Rolle in der Architektur von Spring-Anwendungen. Es ermöglicht Entwicklern, die Erstellung und Verwaltung der Anwendungskomponenten präzise zu steuern. Die zentralen Elemente dieses Managements sind die `@Configuration` Klassen und `@Bean` Annotationen.

5.1 Verwendung von `@Configuration` Klassen

Die `@Configuration` Annotation markiert eine Klasse als Quelle für Bean-Definitionen. In einer solchen Klasse definierte Methoden, die mit der `@Bean` Annotation versehen sind, geben an, dass die Rückgabewerte dieser Methoden von Spring verwaltet werden sollen. Diese Konfigurationsklassen ersetzen traditionelle XML-basierte Konfigurationsdateien und bieten eine moderne, programmatische Methode zur *Bean*-Definition. Ein einfaches Beispiel einer `@Configuration` Klasse könnte wie folgt aussehen:

```
@Configuration
public class AppConfig {
    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

Abbildung 3 Beispiel einer Konfigurationsklasse, selbst erstellt

In diesem Beispiel definiert die Klasse *AppConfig*() eine Methode myBean*()*, die ein neues MyBean() Objekt instanziiert und zurückgibt. Spring wird dann dieses Objekt als *Bean* im Anwendungskontext verwalten.

5.2 Verwendung von @Bean Annotationen

Die @Bean Annotation wird in @Configuration Klassen verwendet, um Methoden zu kennzeichnen, die *Beans* erzeugen. Diese *Beans* sind Teil des Containers und werden entsprechend den angegebenen Konfigurationen verwaltet. *Beans* können Abhängigkeiten zu anderen *Beans* haben, welche durch Spring automatisch injiziert werden. Hier ist ein Beispiel, das zeigt, wie Abhängigkeiten zwischen *Beans* verwaltet werden können:

```
@Configuration
public class ServiceConfig {
    @Bean
    public ServiceA serviceA(DependencyB dependencyB) {
        return new ServiceA(dependencyB);
    }

    @Bean
    public DependencyB dependencyB() {
        return new DependencyB();
    }
}
```

Abbildung 4 Verwendung der @Bean Annotation

In diesem Beispiel erzeugt die Methode dependencyB*()* ein Bean vom Typ DependencyB, das als Abhängigkeit in ServiceA() verwendet wird. Spring sorgt für die korrekte Injektion dieser Abhängigkeit, wenn das *Bean ServiceA* erstellt wird.

5.3 Vorteile der Konfiguration über *@Configuration* und *@Bean*

Die Nutzung von `@Configuration` und `@Bean` bietet mehrere Vorteile gegenüber traditionellen Konfigurationsmethoden:

- **Transparenz und Flexibilität**: Die Konfiguration ist direkt im Code und nicht in externen XML-Dateien. Dies erleichtert das Verständnis und die Verwaltung der Anwendungskonfiguration.
- **Bedingte Konfiguration**: Mit Spring's `@Profile` und `@Conditional` Annotationen lassen sich leicht verschiedene Konfigurationsszenarien für unterschiedliche Umgebungen erstellen.
- **Einfachere Refaktorisierung und Fehlerbehebung**: Da die Konfigurationen Teil des Codes sind, profitieren sie von der Typsicherheit und den Werkzeugen einer integrierten Entwicklungsumgebung (IDE).

6. Spring Managed Beans

In der Welt des Spring Frameworks spielen *Managed Beans* oder einfach *Beans* eine Schlüsselrolle. Sie sind die Objekte, deren Lebenszyklus vom Container verwaltet wird. Dies umfasst die Erstellung, Konfiguration und möglicherweise die Zerstörung dieser Objekte. Die Verwaltung durch Spring bietet eine robuste Grundlage für die Anwendungsentwicklung, indem sie konsistente Mechanismen zur Konfiguration und Verwaltung von Objektabhängigkeiten bietet.

6.1 Annotation und Bean-Definition

Die Definition von *Beans* in Spring erfolgt traditionell über XML-Konfiguration oder Annotationen. Annotationen sind jedoch aufgrund ihrer Einfachheit und Lesbarkeit in der modernen Entwicklung die bevorzugte Methode geworden. Einige der häufigsten Annotationen umfassen:

- **`@Component`***:* Allgemeine Annotation, die jede Klasse als *Bean* kennzeichnet.
- **`@Service`**: Markiert eine Klasse für die Geschäftslogikschicht.
- **`@Repository`**: Kennzeichnet eine Klasse als Data Access Object (DAO).
- **`@Controller`**: Definiert eine Klasse als Controller in einem MVC-Webanwendungsframework.

Diese Annotationen erleichtern das automatische Scannen von Komponenten durch Spring, wodurch Entwickler nicht jede *Bean* explizit in einer Konfigurationsdatei deklarieren müssen. Stattdessen identifiziert und registriert *Beans* automatisch, basierend auf den festgelegten Pfaden und Kriterien.

6.2 Scannen und Erkennen von Beans

Das automatische Erkennen von *Beans* wird durch die `@SpringBootApplication` Annotation vereinfacht, die eine Kombination aus `@Configuration`, `@EnableAutoConfiguration` und `@ComponentScan` darstellt. Diese Annotation gibt Spring an, wo es nach *Beans* suchen soll, und aktiviert automatisch die Konfiguration basierend auf den im Klassenpfad verfügbaren Abhängigkeiten und den beanspruchten *Beans*.

6.3 Vorteile der Verwendung von Spring Managed Beans

Die Nutzung von *Spring Managed Beans* bietet mehrere Vorteile:

- **Einfachheit**: Die Konfiguration durch Annotationen reduziert den Bedarf an umfangreichem *Boilerplate-Code*, was den Code übersichtlicher und einfacher zu verstehen macht.
- **Flexibilität**: Durch das Scannen von Komponenten kann Spring automatisch *Beans* erkennen und konfigurieren, was die Entwicklungszeit verkürzt, und die Flexibilität erhöht.
- **Erweiterbarkeit**: Die Verwendung von *Include/ Exclude* Filters bietet eine starke Kontrolle über die Anwendungskonfiguration und erleichtert die Anpassung an unterschiedliche Betriebsumgebungen.

6.4 Lebenszyklus von Beans

Der Lebenszyklus von *Beans* im Spring Framework beschreibt die verschiedenen Phasen, die ein *Bean* von seiner Erstellung bis zur Zerstörung durchläuft. Dieses Wissen ist essenziell, um komplexe Anwendungen effektiv zu managen und zu optimieren. Der Lebenszyklus eines *Beans* lässt sich in mehrere Phasen unterteilen:

- **Instanziierung**: Die *Bean* wird vom Container erstellt. Dies geschieht basierend auf der *Bean-Definition*, die in einer `@Configuration` Klasse oder durch andere Konfigurationsmethoden wie XML oder Annotationen in der Klasse selbst festgelegt wurde.
- **Abhängigkeitsinjektion**: Nach der Instanziierung injiziert Spring alle nötigen Abhängigkeiten in den *Bean*. Dies kann über *Constructor Injection*, *Setter Injection* oder *Field Injection* erfolgen.
- **Initialisierung**: Nachdem alle Abhängigkeiten injiziert wurden, kann die *Bean* initialisiert werden. Hier können Entwickler benutzerdefinierte Initialisierungsmethoden definieren, um zusätzliche Aufgaben vor der Nutzung der *Bean* durchzuführen.
- **Verwendung**: Die *Bean* steht nun zur Nutzung bereit und kann im Rahmen der Anwendung seinen Dienst verrichten.
- **Zerstörung**: Wenn die Anwendung oder der Anwendungskontext heruntergefahren wird, werden *Beans* entsprechend zerstört. Auch hier können benutzerdefinierte Methoden definiert werden, um Ressourcen freizugeben oder andere Aufräumarbeiten durchzuführen.

6.5 Spring Managed Beans am Beispiel einer Kaffeemaschine

Wie schon zuvor erwähnt, ist ein Schlüsselkonzept von Spring die sogenannten *Managed Beans*, die vom Container verwaltet werden. Um diesen Prozess besser zu verstehen, stellen

wir uns eine Kaffeemaschine vor, die verschiedene Komponenten wie einen Wasserkocher, eine Kaffeemühle und einen Milchaufschäumer verwendet.

6.5.1 Deklarative Komponentenregistrierung mit `@Component`

Um eine Klasse als *Managed Bean* zu registrieren, nutzen wir die `@Component`-Annotation. Nehmen wir an, wir haben eine Klasse namens Milchaufschäumer, die in unserer Anwendung als Komponente fungieren soll:

```
@Component
public class MilkFrother {
    public void froth() {
        System.out.println("Milch wird aufgeschäumt...");
    }
}
```

Abbildung 5 Klasse, die mit @Component annotiert wurde, selbst erstellt

6.5.2 Automatisches Scannen mit `@SpringBootApplication`

Spring Boot verwendet die `@SpringBootApplication` Annotation, die unter anderem das automatische Scannen von Komponenten durch `@ComponentScan` einleitet. Dadurch werden alle mit `@Component` annotierten Klassen im aktuellen Paket und dessen Unterpaketen automatisch erkannt:

```
@SpringBootApplication
public class CoffeeMachineApplication {
    public static void main(String[] args) {
        SpringApplication.run(CoffeeMachineApplication.class, args);
    }
}
```

Abbildung 6 Mit @SpringBootApplication anotierte Klasse, selbst erstellt

6.5.3 Der Ablauf des Classpath Scannings

Der Erkennungsprozess für Komponenten innerhalb des Containers läuft folgendermaßen ab:

- **Start des Containers**: Beim Hochfahren der Anwendung initialisiert Spring den Container.
- **Erkennung von Komponenten**: Spring durchsucht den Classpath nach allen mit `@Component` annotierten Klassen.
- **Instanziierung der Komponenten**: Für jede erkannte Komponente erstellt der Container eine Instanz.
- **Aufnahme in den Kontext**: Diese Instanzen werden im Spring Anwendungskontext verwaltet und sind somit applikationsweit verfügbar.

6.5.4 Spezifische Rollen durch erweiterte Annotationen

Für eine klarere Rollendefinition innerhalb der Anwendung können spezifischere Annotationen wie *`@Service`* oder *`@Repository`* genutzt werden. In unserem Beispiel könnte ein Service, der die Zubereitung des Kaffees steuert, folgendermaßen aussehen:

```
@Service
public class CoffeeService {
    private final MilkFrother milkFrother;

    @Autowired
    public CoffeeService(MilkFrother milkFrother) {
        this.milkFrother = milkFrother;
    }

    public void prepareCoffee() {
        milkFrother.froth();
        System.out.println("Kaffee wird zubereitet...");
    }
}
```

Abbildung 7 Mit @Service annotierte Klasse, selbst erstellt

6.6 Feinabstimmung mit @ComponentScan und @Configuration

Mit `@ComponentScan` in Kombination mit `@Configuration` kann genau festgelegt werden, welche Komponenten der Container erkennen und verwalten soll. Angenommen, wir wollen nur bestimmte Teile unserer Kaffeemaschine durch Spring verwalten lassen:

```
@Configuration
@ComponentScan(basePackageClasses = {CoffeeService.class, MilkFrother.class})
public class CoffeeMachineConfig {
    // Konfigurationsdetails...
}
```

Abbildung 8 Mit @Configuration und @ComponentScan annotiere Konfigurationsklasse

6.6.1 Feintuning mit IncludeFilters und ExcludeFilters

Spring erlaubt es uns, mit `IncludeFilters` und `ExcludeFilters` noch genauer zu bestimmen, welche *Beans* erkannt und verwaltet werden sollen. Angenommen, wir haben eine Test-Komponente in unserer Kaffeemaschinen-Anwendung, die wir nur unter bestimmten Bedingungen einbinden möchten:

```
@Configuration
@ComponentScan(basePackageClasses = CoffeeMachineApplication.class,
               includeFilters = @Filter(type = FilterType.ASSIGNABLE_TYPE, classes
               = DevelopmentBean.class),
               excludeFilters = @Filter(type = FilterType.ANNOTATION, classes =
               ExcludeFromScan.class))
public class CoffeeMachineConfig {
    // Konfiguration, die bestimmte Beans ein- oder ausschließt
}
```

Abbildung 9 @Componentenscan mit je einem Include und Exclude Filter, selbst erstellt

In diesem Beispiel wird `DevelopmentBean` explizit eingeschlossen, während alle *Beans*, die mit `@ExcludeFromScan` annotiert sind, ausgeschlossen werden. Dies bietet eine hohe Flexibilität bei der Konfiguration der Anwendungskomponenten.

7. Dependency Injection

Dependency Injection, kurz DI, ist dabei eine spezielle Ausprägung der IoC und bezeichnet den Prozess, bei dem die Abhängigkeiten einer Komponente durch den IoC-Container bereitgestellt werden. DI sorgt dafür, dass Komponenten ihre Abhängigkeiten in Form von Konstruktorargumenten, Setter-Methoden oder Feldern erhalten, anstatt sie selbst zu instanziieren.

7.1 Das Problem traditioneller Ansätze

Stellen Sie sich eine Kaffeemaschine vor, die verschiedenen Kaffee zubereiten kann – sei es Espresso, Latte Macchiato oder Cappuccino. Jede Kaffeeart benötigt eine spezielle Zubereitungsmethode und somit verschiedene Zutaten oder Komponenten, wie etwa einen Milchaufschäumer für Latte Macchiato oder einen starken Espresso-Shot für den Cappuccino. In einer traditionell programmierten Kaffeemaschine könnten diese Komponenten fest im Code der Kaffeemaschine integriert sein:

```
@Component
public class CoffeeMachine {
	private final EspressoMaker espressoMaker = new EspressoMaker();
	private final MilkFrother milkFrother = new MilkFrother();

	public Coffee makeLatte() {
		Coffee coffee = espressoMaker.makeEspresso();
		coffee.addMilk(milkFrother.frothMilk());
		return coffee;
	}
}
```

Abbildung 10 Erstellung einer Kaffeemaschine ohne DI, selbst erstellt

Diese Herangehensweise macht es schwierig, die Kaffeemaschine für neue Kaffeearten zu erweitern oder Komponenten wie den `EspressoMaker()` oder `MilkFrother()` durch

verbesserte Versionen zu ersetzen. Die Lösung - *Dependency Injection*. Mit DI überlässt man die Bereitstellung der Komponenten, wie `EspressoMaker()` und `MilkFrother()` einem externen Container. DI kann auf verschiedene Weisen umgesetzt werden, die häufigsten Methoden in Spring sind:

- **Constructor-Injection**: Dabei werden die Abhängigkeiten durch den Konstruktor des *Beans* bereitgestellt. Dies ist oft die bevorzugte Methode, da sie die Unveränderlichkeit der Objekte sicherstellt und Abhängigkeiten klar definiert.
- **Setter-Injection**: Hier werden Setter-Methoden verwendet, um Abhängigkeiten nach der Instanziierung des *Beans* zu injizieren. Dies ist nützlich, wenn die Abhängigkeiten optional sind oder wenn ihre Werte geändert werden müssen.
- **Field-Injection**: Bei dieser Methode werden die Abhängigkeiten direkt in die Felder des *Beans* injiziert. Obwohl dies für kleinere oder weniger komplexe Projekte bequem sein kann, wird es wegen der schwierigeren Wartung und geringeren Transparenz oft kritisiert.

Das heißt, wenn der Container hochfährt, initialisiert er alle Objekte und in einer weiteren Phase werden diese ganzen benötigten Abhängigkeiten aufgelöst und dann wird diese Verdrahtung stattfinden, sodass am Ende ein komplettes Objekt Geflecht liegt. Nachfolgend werden die genannten Methoden beschrieben.

7.1.1 Constructor Injection

Eine Möglichkeit, DI zu nutzen, ist die *Constructor Injection*, bei der die notwendigen Komponenten über den Konstruktor bereitgestellt werden:

```
@Component
public class CoffeeMachine {

	private final EspressoMaker espressoMaker;
	private final MilkFrother milkFrother;

	@Autowired
	public CoffeeMachine(EspressoMaker espressoMaker, MilkFrother milkFrother) {
		this.espressoMaker = espressoMaker;
		this.milkFrother = milkFrother;
	}

	public Coffee makeLatte() {
		Coffee coffee = espressoMaker.makeEspresso();
		coffee.addMilk(milkFrother.frothMilk());
		return coffee;
	}
}
```

Abbildung 11 Constructor Injection, selbst erstellt

7.1.2 Setter Injection

Eine weitere Möglichkeit ist die *Setter Injection*, die es erlaubt, Abhängigkeiten nach der Objekterstellung zu ändern. Zu beachten ist, dass hier kein `final` gesetzt werden.

```
@Component
public class CoffeeMachine {

	private EspressoMaker espressoMaker;
	private MilkFrother milkFrother;

	@Autowired
	public void setEspressoMaker(EspressoMaker espressoMaker) {
		this.espressoMaker = espressoMaker;
	}

	@Autowired
	public void setMilkFrother(MilkFrother milkFrother) {
		this.milkFrother = milkFrother;
	}

	// ...
}
```

Abbildung 12 Setter Injection, selbst erstellt

7.1.3 Field Injection

Field Injection injiziert die Abhängigkeiten direkt in die Felder der Klasse, ohne *Setter* oder *Konstruktoren* zu benötigen:

```
@Component
public class CoffeeMachine {

	@Autowired
	private EspressoMaker espressoMaker;

	@Autowired
	private MilkFrother milkFrother;

	// ...
}
```

Abbildung 13 Field Injection, selbst erstellt

7.2 Vorteile von Dependency Injection

Die Anwendung von DI im Spring Framework bietet zahlreiche Vorteile:

- **Entkopplung**: DI fördert eine geringere Kopplung zwischen den Komponenten einer Anwendung, was zu einer flexibleren und wartungsfreundlicheren Codebasis führt.
- **Erleichterte Unit-Tests**: Da Abhängigkeiten extern bereitgestellt werden, können sie leicht durch Mock-Objekte in Unit-Tests ersetzt werden, was die Testbarkeit erheblich verbessert.
- **Vereinfachte Konfiguration**: Abhängigkeiten und ihre Konfigurationen können zentral verwaltet werden, was Änderungen und Erweiterungen erleichtert.

- **Austauschbarkeit**: Verschiedene Implementierungen von `EspressoMaker()` und `MilkFrother()` leicht austauschen, was die Flexibilität und Erweiterbarkeit erhöht.
- **Skalierung**: Neue Kaffeekreationen hinzufügen, ohne die bestehende Klassenstruktur zu ändern, indem wir einfach neue Abhängigkeiten hinzufügen.

8. Optionale Abhängigkeit

Spring Managed Beans können fehlen hier z.B.: Milch, Kunde 1 kauft es, Kunde 2 nicht. Dann soll der Container keine Injizierung durchführen. Das muss allerdings explizit ausgezeichnet werden. Die nachfolgende Abbildung zeigt ein Beispiel hierfür.

```
Option 1: Annotationsattribut
@Autowired( required = false )
MilkFrother milkFrother;


Option 2: @Nullable
@Autowired @Nullable MilkFrother milkFrother;


Option 3: Einsatz von Optional
@Autowired Optional<MilkFrother> milkFrother;


Option 4: ObjectProvider
@Autowired ObjectProvider<MilkFrother> milkFrother;
```

Abbildung 14 Optionale Abhängigkeit, selbst erstellt

9. Zyklische Abhängigkeiten

Zyklische Abhängigkeiten entstehen, wenn zwei oder mehr Klassen in einer Weise voneinander abhängen, dass sie direkt oder indirekt ihre eigene Instanziierung verursachen. Dies kann zu Problemen bei der Initialisierung der beteiligten Klassen führen. Um ein Beispiel für zyklische Abhängigkeiten in unserem Kaffeemaschinen-Beispiel zu integrieren, könnten wir uns vorstellen, dass die *`CoffeeMachine`*`()` eine Benachrichtigung senden möchte, wenn der Kaffee fertig ist, und dass der `NotificationService`, der für das Senden dieser Benachrichtigungen zuständig ist, auch Informationen von der Kaffeemaschine benötigt, um zu wissen, welche Art von Kaffee zubereitet wurde.

```
@Component
public class CoffeeMachine {

	private final EspressoMaker espressoMaker;
	private final MilkFrother milkFrother;
	private NotificationService notificationService; // Zyklische Abhängigkeit

	@Autowired
	public CoffeeMachine(EspressoMaker espressoMaker, MilkFrother milkFrother) {
		this.espressoMaker = espressoMaker;
		this.milkFrother = milkFrother;
	}

	public void setNotificationService(NotificationService notificationService)
	{
		this.notificationService = notificationService;
	}

	public Coffee makeCoffee() {
		// Kaffee wird zubereitet...
		notificationService.sendNotification("Ihr Kaffee ist fertig!");
		return new Coffee();
	}
}
```

Abbildung 15 Beispiel für Zyklische Abhängigkeit, selbst erstellt

```
@Component
public class NotificationService {

	private CoffeeMachine coffeeMachine; // Zyklische Abhängigkeit

	@Autowired
	public void setCoffeeMachine(CoffeeMachine coffeeMachine) {
		this.coffeeMachine = coffeeMachine;
	}

	public void sendNotification(String message) {
		// Sendet eine Benachrichtigung, dass der Kaffee fertig ist
		System.out.println(
			message +
			coffeeMachine.getClass()
			.getSimpleName()
		);
	}
}
```

Abbildung 16 NotificationService, selbst erstellt

In diesem Szenario hat *`CoffeeMachine`*`()` eine Abhängigkeit zu *`NotificationService`*, um eine Benachrichtigung zu senden, wenn der Kaffee fertig ist. Gleichzeitig benötigt `NotificationService` eine Referenz auf *`CoffeeMachine`*`()`, um Details über den zubereiteten Kaffee in der Benachrichtigung zu inkludieren. Dies führt zu einer zyklischen Abhängigkeit, die Spring standardmäßig nicht auflösen kann.

9.1 Probleme durch zyklische Abhängigkeiten

Zyklische Abhängigkeiten können verschiedene Probleme verursachen, darunter:

- **Initialisierungsfehler**: Der Spring Container kann die beteiligten *Beans* nicht korrekt erstellen, da jede *Bean* auf die Instanziierung der anderen wartet.
- **Schwierigkeiten bei der Wartung**: Zyklische Abhängigkeiten können den Code schwer verständlich und wartbar machen, da die Entkopplung der Komponenten nicht ausreichend ist.
- **Eingeschränkte Testbarkeit**: Unit-Tests werden komplizierter, da die *Beans* nicht leicht isoliert voneinander getestet werden können.

9.2 Lösungen für zyklische Abhängigkeiten

Spring bietet mehrere Ansätze, um zyklische Abhängigkeiten zu handhaben oder zu vermeiden:

- **Setter- oder Field Injection verwenden**: Anstelle der *Constructor Injection* kann die *Setter-* oder *Field Injection* verwendet werden, um die Abhängigkeiten erst nach der Erstellung einer *Bean* zu setzen. Dies kann das Problem umgehen, birgt jedoch das Risiko, dass die *Beans* in einem nicht vollständig initialisierten Zustand verwendet werden.
- **Designänderungen**: Oftmals ist die beste Lösung für zyklische Abhängigkeiten eine Überarbeitung des Designs. Das kann bedeuten, dass man gemeinsame Funktionalitäten in eine separate Bean auslagert oder die Anwendung so strukturiert, dass eine klare Hierarchie ohne Abhängigkeitsschleifen entsteht.
- **Verwendung von *@Lazy Annotation***: Die `@Lazy Annotation` in Spring kann dazu beitragen, die Erstellung einer *Bean* zu verzögern, bis sie tatsächlich benötigt wird. Dies kann in einigen Fällen die Probleme durch zyklische Abhängigkeiten lösen. Im Kontext unseres Beispiels könnten wir `@Lazy` im *`NotificationService`* verwenden:

```
@Component
public class CoffeeMachine {

	@Autowired
	public void setNotificationService(@Lazy NotificationService notificationService) {
		this.notificationService = notificationService;
	}
	// ...
}
```

Abbildung 17 Verwendung von @Lazy, selbst erstellt

10. Benutzerdefinierte `init()` und `destroy()`

Spring bietet mehrere Möglichkeiten, benutzerdefinierte Methoden für die Initialisierung und Zerstörung von *Beans* zu definieren:

- **@PostConstruct und @PreDestroy Annotationen**: Diese Annotationen markieren Methoden, die direkt nach der Instanziierung bzw. direkt vor der Zerstörung des *Beans* ausgeführt werden sollen. Diese Methoden eignen sich für alle notwendigen Initialisierungs- und Aufräumarbeiten und werden wie nachfolgend gezeigt genutzt:

```
@Component
public class ExampleBean {

    @PostConstruct
    public void init() {
        // Initialisierungslogik
    }

    @PreDestroy
    public void destroy() {
        // Aufräumlogik
    }
}
```

Abbildung 18 Beispiel init() und destroy() mit den Annotationen @PostConstruct und @PreConstruct, selbst erstellt

- **Implementierung von InitializingBean und DisposableBean**: Durch Implementierung dieser Interfaces kann ein Bean die *afterPropertiesSet()* Methode für die Initialisierung und die `destroy()` Methode für die Zerstörung definieren.

```
@Component
public class AnotherExampleBean implements InitializingBean, DisposableBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        // Initialisierungslogik
    }

    @Override
    public void destroy() throws Exception {
        // Aufräumlogik
    }
}
```

Abbildung 19 Beispiel afterPropertiesSet() und destroy() der Interfaces InitialingBean und DisposableBean, selbst erstellt

Die Kontrolle über den Lebenszyklus von *Beans* ermöglicht es Entwicklern, Ressourcen effizient zu verwalten, die Leistung zu optimieren und Probleme zu vermeiden, die durch nicht freigegebene Ressourcen entstehen können. Es stellt sicher, dass *Beans* korrekt initialisiert sind, bevor sie verwendet werden, und dass sie ordnungsgemäß aufgeräumt werden, wenn die Anwendung beendet wird.

11. Zusammenfassung

In diesem Artikel haben wir eine umfassende Betrachtung des Spring Frameworks vorgenommen, die seine Kernkomponenten und Prinzipien beleuchtet. Beginnend mit dem *Spring Container* und der *Inversion of Control*, haben wir die Bedeutung dieser Konzepte für die Schaffung einer flexiblen und entkoppelten Architektur herausgearbeitet. Die Rolle von *Spring Managed Beans* und die Mechanismen der *Dependency Injection* wurden ebenso betrachtet, um die Vorteile dieser Techniken in der Verbesserung der Codequalität und der Förderung von Testbarkeit zu verdeutlichen.

Wir haben auch die Herausforderungen durch zyklische Abhängigkeiten adressiert und Lösungsstrategien aufgezeigt, die helfen, solche Probleme in Anwendungen zu vermeiden oder zu beheben. Zusätzlich haben wir das Konfigurationsmanagement in Spring, einschließlich der Verwendung von `@Configuration` Klassen und `@Bean` Annotationen, erörtert und den Lebenszyklus von *Beans* detailliert, wobei besonderes Augenmerk auf benutzerdefinierte `init()`- und `destroy`() Methoden gelegt wurde.

Durch das Verständnis dieser Konzepte und Mechanismen sind Entwickler in der Lage, die Vorteile des Spring Frameworks voll auszuschöpfen und Anwendungen zu entwickeln, die nicht nur leistungsstark und skalierbar sind, sondern auch leicht zu warten.

12. Über die Autoren

Brandon Wohlwend

Während seines Mathestudiums an der WWU Münster, brachte er sich autodidaktisch diverse Programmiersprachen und Frameworks bei und vertiefte sich dabei in der Webentwicklung. Durch die Arbeit an der MBTSuite entwickelte er sich hin zum Fullstack Entwickler.

Sebastian Noppenberger

Nach seinem Bachelorabschluss in Medieninformatik, an der TH Nürnberg, sammelte er zunächst Erfahrungen als Frontendentwickler. Durch die Arbeit an der MBTSuite entwickelte er sich zum Fullstack Entwickler mit dem Schwerpunkt in Angular und Java.

13. Über sepp.med

Qualität sichert Erfolg

sepp.med – das heißt jahrzehntelange Erfahrung in der agilen Softwareentwicklung und der agilen Qualitätssicherung für sicherheitskritische Systeme und Prozesse. Ihr verlässlicher Partner bei der agilen Entwicklung, Integration und Qualitätssicherung.

sepp.med ist ein mittelständisches, inhabergeführtes Unternehmen, das sich auf IT-Lösungen mit integrierter Qualitätssicherung in komplexen, sicherheitskritischen Bereichen spezialisiert hat. Wir setzen auf strukturiertes Vorgehen, konsequent nach den Maßgaben der Kunden und den Anforderungen des jeweiligen Projekts mit zielgerichtetem Einsatz von Prozessen, Methoden und Werkzeugen.

Weit über das fränkische Medical Valley hinaus ist das in Röttenbach bei Erlangen ansässige Unternehmen bekannt für seine besondere Expertise in sicherheitskritischen Industrien:

- Medizintechnik
- Pharmazie
- Automobiltechnik
- Avionik
- Öffentlicher Sektor
- Finance & Assurance

14. Abbildungsverzeichnis