Spring Core – Førebuande workshop

Mål: Repetera Spring-mekanismane som test-workshopen føreset at me kan.

Tidsramme: 4 timar (4 × 50 min + pausar)

Gjennomgåande mønster i kvar blokk:

1. Korleis det fungerar i Spring Framework (utan Boot)
2. Korleis Spring Boot forenklar det same
3. Samanlikning og refleksjon – kva Boot gøymer, og kvifor det er viktig å forstå kva som ligg under

Blokk 1: Dependency Injection og IoC-containeren (50 min)

1 – Spring Framework

- Problemet med tett kobling og kvifor Inversion of Control finst
- Opprette ein `ApplicationContext` manuelt med `AnnotationConfigApplicationContext`
- Definere bønner med `@Configuration` og `@Bean` – eksplisitt og synleg
- Konstruktørinjeksjon: kople bønnene saman for hand i Java-konfig
- Bean-scope og livssyklus (`@PostConstruct`, `@PreDestroy`)

Hollywood-prinsippet

> "Don't call us – we'll call you."

Når ein skodespelar møter til audition, ringer han ikkje regissøren kvar dag for å spørje om han fekk rolla. Han gjev frå seg kontaktinformasjonen sin og ventar. Regissøren tek kontakt når tida er inne.

Dette er eit designprinsipp frå ca 1988. Det vart nemnt i fleire artiklar om rammeverkdesign, og seinare trekt fram i "Design Patterns: Elements of Reusable Object-Oriented Software" (også kjent som GoF-boka).
I programvareverda kallar vi det ...?

Inversion of Control

Me kjenner prinsippet, men har kanskje ikkje tenkt over at det er akkurat det som føregår i Spring. Eit designprinsipp som nærmar seg 40 års alder, med ein container rundt.

Me kjenner prinsippet, men kva er mekanisma som let oss gjera det…?

IoC er prinsippet – me gir frå oss kontrollen.

Dependency Injection er mekanisma – korleis avhengigheitene faktisk kjem inn.

IoC er "don't call us", DI er "here's your script and costume when you arrive on set".

Ein annan fasilitet må nemnast i samme andedrag, kva er det som mogleggjer DI i t.d. Spring?

ApplicationContext

Pakke med kjende patterns: Factory, Registry, Lifecycle Manager.

La oss sjå på `GenericApplicationContext` og `registerBean()` om ei lita stund.

Me vil sjå tre nivå av det same prinsippet:

1. `registerBean()` – me skriv oppskrifta imperativt
2. `@Configuration` / `@Bean` – me deklarerar kva som skal finnast
3. `@Component` + scanning – me let rammeverket oppdage det sjølv

Konstruktørinjeksjon

Kvifor ikkje setter eller felt?

Ein klasse som tek imot avhengigheiter i konstruktøren, kan ikkje eksistera i ein ugyldig tilstand. (Det av same grunn at me lagar felt `final`.

Kvart steg gøymer meir enn det føregåande. Spørsmålet er alltid: forstår me kva som vert gøymt?

Øving 1.1

Diskusjon etter øving 1.1

1. Kva teknikk er brukt av Spring til å oppretta bønner og kobla avhengigheitar?

- i. Set breakpoint på linja med `context.refresh()`. (Me er i `spring-plain`-modulen.)
- ii. Naviger til `BeanUtils.instantiateClass()` i biblioteket `org.springframework:spring-beans`.
- iii. Start debugging av applikasjonen. Sjå at den stoppar på kallet til `context.refresh()`, og la den gå vidare til ei linje i `BeanUtils.instantiateClass()`. Stopp der.
- iv. Gå til 'Frames'-fana i 'Debugger'-panelet, høgreklikk dersom det er ei utgråa linje med 'hidden frames' e.l. Ekspander. Då ser me heile kallet frå vår kode til `BeanUtils.instantiateClass()`.
- v. Dersom me grev djupare i `BeanUtils.instantiateClass()`, finn me at den resulterande pipelina etter kvart opprettar ein instans av klassa som den har fått ein konstruktør for.
- vi. La oss gjera ei forenkla simulering: Eige notat.

Øving 1.2

1. Repeter 1.1.10. i `spring-boot`.
2. Legg til ein `@Configuration`-klasse (konvensjonelt plassert i `config/`-mappa), og opprett nokre `Bean`-ar i den.
3. Skriv om MocktailApplication til å nytta bønnene frå førre punkt.

Øving 1.3

5. Oppgåve 1.3: Kan du fjerna heile `config/`-mappa, og oppnå det samme ved å legga til nokre sentrale Spring Boot-annotasjonar?

- Her er det fleire tilnærmingar. Gå gjennom den openbare, samt den kanskje meir konvensjonelle.

Øving 1.4

1. Samanlikn tal på bønner i `spring-plain` og `spring-boot`.
   notat

2. Me hadde nokre fleire bønner i `spring-boot`.

- Kor kjem dei frå?
- Klarar me oss utan dei?
- Dei tynne JAR-filer i modulane representerar prosjektkoden. Den feite JAR-en inneheld alt som Spring Boot dreg inn, i tillegg til prosjektkoden.
- Kva skjer no dersom me tek inn `BeanPostProcessor`-en frå 1.1.12.7? I MocktailApplication.java:
  notat

3. Korleis kan me pakka modulane som JAR-filer?

- `mvn clean package`
- `ls -lh ./spring-plain/target/spring-plain-*.jar ./spring-boot/target/spring-boot-*.jar ./spring-boot/target/spring-boot-*.jar.original`
- Tal på bønner samanlikna med tal på MB = Prisen for konvensjonane.

4. Bean-scopes

```
var service1 = context.getBean(MocktailService.class);
var service2 = context.getBean(MocktailService.class);
System.out.println("Same instans? " + (service1 == service2));
```

5. Livssyklus-hooks

```
@PostConstruct
public void init() {
    System.out.println(">>> MocktailRepository oppretta");
}

@PreDestroy
public void cleanup() {
    System.out.println(">>> MocktailRepository vert stengd");
}
```

spring-plain

Påkrevd ny avhengigheit i `main`:

```
<dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId>
    <version>3.0.0</version>
</dependency>
```

I tillegg må annotasjonsprosesseringa registrerast, t.d. i MocktailApplication.java:

```
context.registerBean(CommonAnnotationBeanPostProcessor.class);

---

<!-- Etter context.refresh(); -->
context.registerShutdownHook(); <!-- Dette skjer automatisk i Spring Boot, og er noko av det som gjer at livssyklusen "berre funkar". -->
```

6. Kva skjer dersom me fjernar jakarta.annotation-api frå `spring-plain`?

Koden kompilerer og køyrer heilt fint, men livssyklus-hooks-ane vert aldri køyrde.

- Poeng: Spring er ikkje magisk – det kan berre reagere på det som faktisk er tilgjengeleg på classpath.

7. Eksperimenter med å øydelegga applikasjonane.

- Fjern registreringa av eit repository i `spring-plain`.
- Fjern annotasjon som registrerar eit repository frå `spring-boot`.
- Samanlikn feilmeldingane.

Oppsummering av øving 1

1. Spring utan Boot

Kva problem løyser DI?

- Tett kobling: når ein klasse sjølv opprettar avhengigheitene sine med `new`, vert det vanskeleg å byte ut, teste og vedlikehalde koden
- Inversion of Control: i staden for at klassen styrer sine eigne avhengigheiter, let vi noko utanfrå (containeren) ta ansvaret

ApplicationContext

- Spring sin IoC-container – ei samling av objekt (bønner) som containeren opprettar, koplar saman og styrer livssyklusen til
- GenericApplicationContext og registerBean() – den mest eksplisitte måten å registrera bønner på

Konstruktørinjeksjon

- Avhengigheitar kjem inn via konstruktøren – klassen sjølv veit ikkje kor dei kjem frå
- Dette gir laus kobling: Klassen er avhengig av eit grensesnitt eller ein type, ikkje ein konkret implementasjon.

Del 2 – Spring Boot

`@Configuration` og `@Bean`

Deklarativ registrering – same mekanisme som `registerBean()`, men meir lesbar

Kvar `@Bean`-metode returnerar eit objekt som containeren tek eigarskap over

Stereotyp-annotasjonar og component scanning

- `@Component`, `@Service`, `@Repository` – annotasjonar som let Spring oppdaga bønner automatisk
- `@SpringBootApplication` aktiverar component-scanning og plukkar opp alt i same pakke og underpakkar

Implisitt konstruktørinjeksjon

Når ein klasse berre har éin konstruktør, treng ein ikkje `@Autowired`. Spring forstår kva den skal gjera

`CommandLineRunner`: Idiomatisk måte å køyra logikk ved oppstart – sjølv denne vert injisert av containeren

Del 3 – Samanlikning

Frå eksplisitt til implisitt

`registerBean()` → `@Configuration/@Bean` → `@Component` + component-scanning: tre nivå av abstraksjon, same underliggjande mekanisme

- Kva vinn vi? Mindre kode.
- Kva mistar vi? Synlegheit, det er vanskelegare å sjå kva som finst i konteksten

Bean-scope og livssyklus

- Singleton (standard): éin instans per container – same objekt vert injisert overalt
- Kva andre scopes finst? Bean Scopes.
- `@PostConstruct` og `@PreDestroy` – livssyklus-hooks som fungerar likt i begge variantane

Kvifor dette er viktig for testing

Kort frampeik (utan å gå inn i testverktøy): dersom me forstår kva konteksten inneheld og korleis bønner er kopla saman, forstår me òg kva me kan byte ut og isolere

Blokk 2: Konfigurasjon og profilar (50 min)

1 – Spring Framework

- `PropertySource` og `Environment`-abstraksjonane
- Lese eigenskapar med `@Value` frå ei `.properties`-fil registrert manuelt
- `@Profile` på `@Configuration`-klassar – aktivere med `ctx.getEnvironment().setActiveProfiles()`

Blokk 3: Auto-configuration og AOP (50 min)

1 – Spring Framework

Auto-configuration-grunnlaget

- Kva du må gjere sjølv: `DataSource`-bean, `EntityManagerFactory`, `TransactionManager` – alt er eksplisitt

Øving 3: Prioritet og overstyring

Overstyring av `mocktail.bar-name` på ulike måtar i `spring-boot`: Legg til ulike verdiar på følgande stadar:

- i. Verdien i application.yml
- ii. Ein profilspesifikk application-summer.yml
- iii. Miljøvariabel: MOCKTAIL_BAR_NAME=Strandbar
- iv. Kommandolinje: --mocktail.bar-name=Fjellbar

1. Før me køyrer applikasjonen: Kven av 1 - 4 ovanfor vinn? *Prioritetsrekkjefølgja: fil → miljøvariabel → kommandolinje*.
2. Køyr applikasjonen og stadfest/falsifiser.

Blokk 4: Sjølvstudium: Webblaget og persistenslaget (50 min)

1 – Spring Framework

- Spring MVC utan Boot: `DispatcherServlet`-registrering, `WebApplicationContext`, `@EnableWebMvc`
- Handskrive `DataSource`, `LocalContainerEntityManagerFactoryBean`, `PlatformTransactionManager`
- `@Transactional` i praksis – no forstår deltakarane at dette er AOP-proxyen frå blokk 3

2 – Spring Boot

- Embedded server (Tomcat) startar automatisk – ingen servlet-konfig naudsynt
- `@RestController` + Jackson auto-konfigurert ut av boksen
- Spring Data JPA: skriv eit grensesnitt, Boot genererer implementasjon og koplar til ein auto-konfigurert DataSource
- `@ExceptionHandler` / `@ControllerAdvice` for feilhandtering

3 – Samanlikning

- Side-om-side: manuell servlet- og JPA-oppsett vs. Boot-versjonen