Spring Framework – Handout

Martin Ledvinka
martin.ledvinka@fel.cvut.cz

25. listopadu 2015

1 Spring Framework

Spring framework vznikl v roce 2002 jako alternativa tehdejší težkopádné enterprise verze Javy – J2EE (dnešní Java EE je na tom o mnoho lépe, částečně právě díky konkurenci Springu). Jeho hlavní devizou byla nezávislost na aplikačním serveru – Spring ho nepotřebuje, bez problému ho lze použít v Java SE aplikaci – a relativně snadná konfigurace (oproti tehdejší J2EE). Dnes jde o rozsáhlou knihovnu poskytující kromě základních modulů s DI, JDBC a ORM integrací, MVC webovým frameworkem a podporou pro zabezpečení aplikace též modul pro dávkové operace, sociální sítě či mobilní vývoj.

Základem Springu jsou dva propojené koncepty – *Dependency Injection (DI)* a *Inversion of Control (IoC)*.

Dependency Injection umožňuje vyhnout se hardcoded vazbám mezi objekty. V kostce řečeno, komponenty (třídy) závisí na jiných pomocí veřejného API (např. interface), konkrétní implementaci (vytvořenou samozřejmě programátorem) pak do komponenty dodá (*inject*) Spring. Tato organizace umožňuje vyvíjet komponenty, jejichž vývoj, úpravy a případná výměna jsou na sobě minimálně závislé.

Inversion of Control přenáší odpovědnost za řízení toku programu na knihovnu třetí strany (v našem případě Spring), která pak volá námi vytvořený kód. Tento princip je prakticky nutností pro všechny aplikace, které závisí na vnějším vstupu. Např. v aplikaci s uživatelským rozhraní je program řízen frameworkem, který vykresluje a spravuje toto rozhraní a náš kód je volán až v případě uživatelovy akce. Pokud bychom se pokusili o opak, náš program by uživatele vyzval k akci a na jejím základě by program provedl nějakou logiku. Pak by program opět stál a čekal na vstup. V případě frameworku s IoC se jedná o generický nástroj starající se o svou zodpovědnost (uživatelský vstup) a business logiku obstarává naše aplikace. To vše přispívá k lepší stuktuře kódu a jeho zaměření. Někdy se IoC charakterizuje jako tzv. hollywoodský přístup – "Don't call us, we'll call you."

Vývoj aplikace se Springem se tedy víceméně skládá z vývoje komponent (*bean*), které Spring za běhu propojí pomocí DI. Řízení běhu aplikace pak má na starosti rozhraní spravované za pomoci Springu, které volá náš kód.

Náš výklad se bude točit okolo dvou klíčových slov – *bean* a *inject*:

Bean je instance spravovaná Springem.

Inject je mechanismus, kterým Spring vkládá závislosti (dependency) do bean. Těmito závislostmi mohou být buď jiné beany, nebo literální hodnoty (String, primitivní typy, kolekce hodnot).

1.1 Spring Bean

Jak se objekty nějaké třídy stanou beanou? Celkem snadno, stačí buď třídu označit jako Spring komponentu nebo instanci vytvořit předem definovaným způsobem, který Spring rozeznává a pozná tak, že má vytvořenou instanci registrovat jako beanu.

Beana je pak spravována Springem (Spring context, nebo taky container), může být injektována do jiných objektů a je dostupná ze všech částí aplikace.

1.1.1 Jak vytvářet beany?

V zásadě jsou tři způsoby.

1. Anotace třídy Pokud třídu oanotujeme některou ze Springem rozeznávaných anotací (`@Component`, `@Repository`, `@Service`, `@Controller` apod.), Spring sám při inicializaci kontextu vytvoří instanci dané třídy a bude ji spravovat jako bean. Tato beana může být injektována do jiných bean nebo se na ni můžeme dotázat přímo kontejneru.

2. Deklarace v konfiguračním souboru Spring byl vždy konfigurován primárně pomocí XML souborů. V takové souboru můžeme deklarovat beanu elementem `bean` pomocí následující syntaxe:

```
<bean id="identifikator"class="fully.qualified.class.Name"/>
```

V této deklaraci lze též nastavit parametry beany. Podívejte se do souboru *applicationContext.xml* v ukázkovém projektu. Tam je takových bean několik.

3. Ruční vytvoření Beanu můžeme též vytvořit my sami. Jednou variantou je metoda s anotací `@Bean`, která vrací instanci beany. Tato metoda musí být buď ve třídě s anotací `@Configuration`, nebo v jiné bean třídě[1] Další variantou je přímo factory třída implementující `BeanFactory` interface (nepříliš obvyklé).

[1] Více v dokumentaci na https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html

1.1.2 Bean Scope

Každá bean spravované Springem má svou životnost – *scope*. Ta určuje, kolik instancí dané třídy vznikne a jak dlouho budou v aplikaci existovat. Níže uvádíme výčet hlavních scope hodnot:

singleton *Default* scope hodnota. Znamená, že bude existovat jediná beana dané třídy. Takové beany by měly být stateless, nebo velmi obezřetně hlídat svůj vnitřní stav, protože může snadno dojít k současnému volání metody/metod z více vláken. Singleton beany jsou vytvořeny hned při startu Spring kontextu.

prototype Vždy je vytvořena nová instance beany.

request Nová bean je vytvořena pro každý HTTP request. Platné pouze pro webové aplikace.

session Nová bean pro každou HTTP session. Platné pouze pro webové aplikace.

Pro nás jsou zajímavé hlavně typy singleton, request a session. Scope se definuje pomocí anotace `@Scope`, případně atributu `scope` v XML elementu `bean`.

1.1.3 Použití bean

Beany lze používat v instancích tříd spravovaných Springem, čili v jiných beanách. Dostaneme je tam pomocí již zmíněného mechanismu injekce. V XML konfiguraci toto provedeme pomocí atributu `ref`, kterém se odkážeme na id jiné beany (musí existovat). V konfiguraci pomocí anotací typicky použijeme anotace `@Autowired` nebo `@Inject`[2]

1.2 Spring konfigurace

Původně jediným způsobem konfigurace Springu byly XML soubory. To vedlo často k extrémně dlouhým a těžko spravovatelným dokumentům (nezapomeňte, že každou beanu v aplikaci bylo třeba v XML deklarovat). Naštěstí s příchodem anotací v Java 5 přibyla možnost konfigurovat Spring programově. Kromě anotací to znamená též možnost vytvářet beany konfigurující integraci s jinými frameworky (např. JPA, vizte konfiguraci testů v demo projektu).

Způsobů injekce je několik:

Constructor – instance je injektována jako argument konstruktoru. Dnes se používá jen výjimečně (málokomu se chce psát konstruktor, když nemusí). Použití je snadné, buď se vnoří element `constructor-arg` do deklarace beany v XML, nebo se příslušný konstruktor oanotuje anotací `@Autowired`.

[2] Proč dvě varianty? Anotace `@Inject` je součástí Java CDI standardu, který vznikl dlouho poté, co s DI přišel Spring. Spring se ovšem rozhodl podporovat tuto anotaci vedle "své" anotace `@Autowired`. Všude, kde píšeme `@Autowired`, může být i `@Inject`.

Setter – injekce pomocí setter metody. Používáno pro integraci s jinými frameworky (vizte opět nastavení naší demo aplikace). V XML použijeme element `property`, v Javě anotaci `@Autowired` na setteru.

Field – injekce přímo do fieldu ve třídě, žádný setter není potřeba. Dnes nejčastěji používaná varianta. Stačí fieldu dát anotaci `@Autowired`.

1.2.1 XML konfigurace

Čistě XML konfigurace je dnes spíše na ústupu. Většinou se používá hybridní spojení Java + XML konfigurace. XML konfiguraci je třeba, stejně jako Javovskou, načíst, aby aplikace poznala, že má Spring kontejner nastartovat. To se ve webové aplikaci provede jejím referencováním z *web.xml*, ve springovských testech pak její referencí v anotaci `@ContextConfiguration`.

Jak se můžete přesvědčit v *applicationContext.xml* v demo aplikaci, XML konfigurace se skládá z deklarace bean, které mají unikátní identifikátory a třídu, jejíž instancí jsou. Jiné beany mohou být referencovány v atributu `ref` pomocí id. Jelikož XML konfigurace používá setter injection, musí mít cílová třída příslušnou setter metodu.

Proč ještě používat XML konfigurace? Jak již bylo řečeno na cvičení, XML konfigurace se může hodit v situacích, kdy potřebujete měnit nastavení aplikace dle prostředí. Je snadné přibalit do war/jar archivu jiný xml soubor. V případě Java konfigurace musíte znovu zkompilovat celý projekt.

1.2.2 Java konfigurace

Pro pravověrné Javisty jasně lepší varianta – celou Spring konfiguraci lze provést přímo v Javě pomocí anotací a factory metod. Opět je potřeba zajistit start Spring kontejneru, tentokrát např. poděděním abstraktní třídy `Abstract AnnotationConfigDispatcherServletInitializer`, která funguje jako listener při startu webového kontejneru (Tomcat) a která spustí Spring. V testech pak stačí v anotaci `@ContextConfiguration` odkázat na třídu/y s konfigurací (uvidíme dále).

Beany zajišťující integraci s jinými frameworky (ORM) vytvoříme pomocí factory metod, ostatní beany vytvoří Spring dle našich anotací. Je ale třeba Springu říct, které soubory má použít pro konfiguraci a kde hledat beany. První zvládneme pomocí anotace `@Configuration`, druhé pak anotací `@ComponentScan`, která určuje, ve kterých balíčcích (package) má Spring pátrat po beanách. Tyto anotace se obvykle používá společně.

1.2.3 Hybridní konfigurace

Oba způsoby konfigurace lze samozřejmě kombinovat. V XML tak například nastavíme přístup k databázi, ale všechny beany naší aplikace budeme vytvářet pomocí anotací. V XML stačí Springu říct, ať použije pro konfiguraci také anotace

(element `annotation-config`) a kde má hledat beany či konfiguraci (element `component-scan`). Opět se stačí podívat do demo projektu.

2 Demo aplikace

Naše ukázková aplikace používá dva způsoby konfigurace – hlavní aplikace je konfigurována hybridní XML a Java konfigurací, testy jsou konfigurovány čistě programově. Většina důležitých informací je přímo v komentářích v kódu, doporučujeme proto kód si projít a porozumět mu. Zmiňme nyní některé zajímavosti.

- Aplikace používá hned několik springovských modulů. Podívejte se do *pom.xml*, kde jsou deklarovány závislosti na nich.
- Přístupové údaje k databázi jsou uloženy v separátních souborech. Takto by ani nebylo nutné měnit celou Spring konfiguraci, stačilo by vyměnit v případě nutnosti jen ony properties soubory.
- Zároveň si můžete všimnout, že zatímco aplikace je nastavena pro přístup k PostgreSQL databázi, testy používají in-memory H2 databázi. To je častý postup. In-memory databáze je mnohem rychlejší na vytvoření, takže jsou s ní testy mnohem rychlejší, než by byly nad full-blown db strojem typu PostgreSQL. Po každém testu pak můžeme databázi zahodit a vytvořit novou. Problém závislosti na db stroji zde raději neřešme, i v praxi byste se měli snažit vyhnout db-specific nastavení. Snad jen konstatujme, že např. H2 má módy kompatibility pro různé jiné db stroje.
- Beana `dataSource` deklaruje data source, skrze který se přistupuje k db. Je to v podstatě mezivrstva mezi JPA a JDBC driverem. Jejím primárním účelem je poskytovat connection pool pro lepší využití spojení s databází.

2.1 XML konfigurace

Hlavní aplikace je konfigurována hybridně. XML nastavení je v *WEB-INF/context/applicationContext.xml*. Všimněte se, že dokument importuje hned několik XML jmenných prostorů (namespace). To proto, že různé moduly mají svůj namespace. Beana `propertyConfigurer` slouží ke čtení property souborů, jejichž hodnoty jsou pak použity k nastavení ostatních bean (hlavně `dataSource`).

Opět zde zmiňme element `spring-configured`, který umožňuje do instancí tříd anotovaných `@Configurable` injektovat závislosti, ačkoliv nejsou Spring beanami. My toho využíváme ve třídě `User`, jejíž instance spravuje persistence provider, ale do nich přesto chceme injektovat `HashProvider`.

2.2 Java konfigurace

Čistě javovská konfigurace je použita v testech. Pro běh testů nepoužíváme standardní JUnit runner, ale jeho upravenou verzi – `SpringJUnit4ClassRunner`. Ten nám umožňuje nastartovat Spring kontext i v rámci testů. Konfigurace

je ve třídách `PersistenceConfig` a `ServiceConfig` a je načtena pomocí anotace `@ContextConfiguration`. Načtení do base třídy všech ostatních testů nám umožňuje její sdílení. Jinak bychom museli všechny anotace deklarovat na všech testovacích třídách.

Z hlediska konfigurace je nejzajímavější třída `PersistenceConfig`, která nastavuje persistenci aplikace. Při bližším pohledu snadno identifikujete odpovídající pasáže v `PersistenceConfig` a v *applicationContext.xml*.

Na `ServiceConfig` je zajímavé snad jen to, že kromě tří anotací je to zcela prázdná třída.

3 Vyzkoušejte si

Zkuste do demo aplikace přidat správu literárních žánrů. Tedy přidejte entitu `Genre`, která bude mít many-to-many vztah s knihou (vida, podobný požadavek je v CP1:)), vytvořte pro ni DAO a service. Zkuste si naimplementovat např. získávání knih podle žánrů, přidávání žánrů knihám apod.

Pokud vám nevyhovuje systém DAO, který jste viděli na 7. cvičení, klidně použijte ten z cvičení minulého.