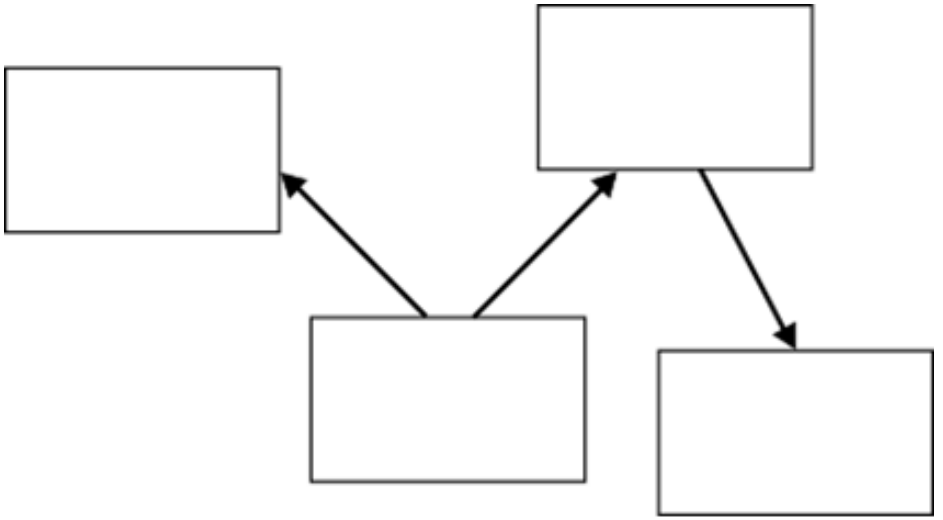
IoC kontejner

Vazby mezi třídami v aplikaci

- Každá Java aplikace se skládá z řady tříd, které mají mezi sebou vazby.
- Například když chcete přečíst data o zákaznících z databáze, tak musíte mít připojení do příslušné databáze.
- Vazby jsou obvykle tvořeny settery, nebo parametry v konstruktorech. Do nich vložíte objekty, které vytvoříte pomocí operátoru new.
- Definování těchto vazeb zjednodušuje IoC kontejner Springu.

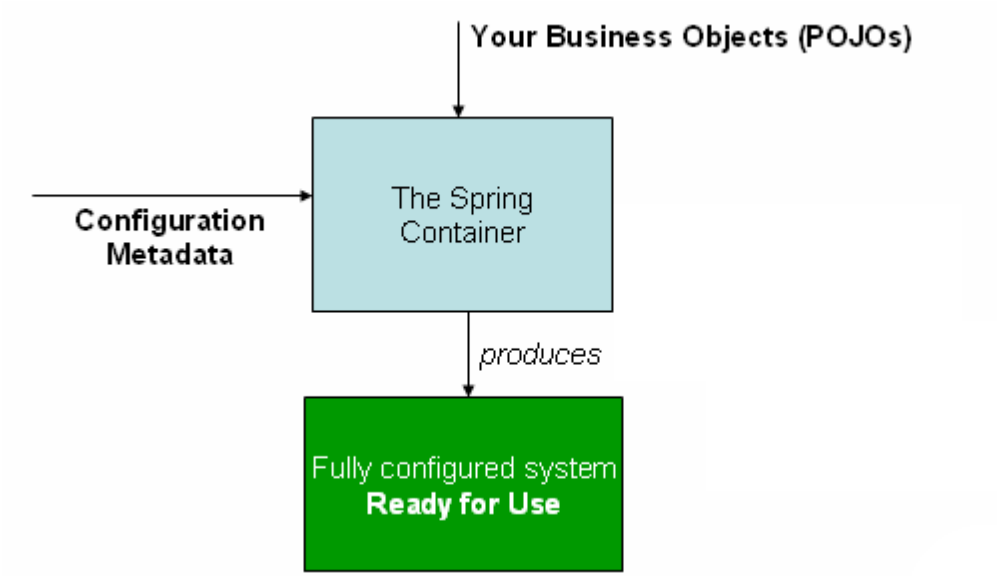


IoC (Inversion of Control) container

- **IoC kontejner** je klíčovou součástí Springu. Jedná se o implementaci návrhového vzoru Inversion of Control (IoC), výstižněji pojmenovaného také jako Dependency Injection (DI).

- Návrhový vzor DI odstraňuje málo flexibilní těsné vazby mezi objekty tím, že převádí odpovědnost za vytváření, inicializaci objektů, nastavování vazeb mezi objekty a rušení objektů na tzv. IoC kontejner.

Spring container

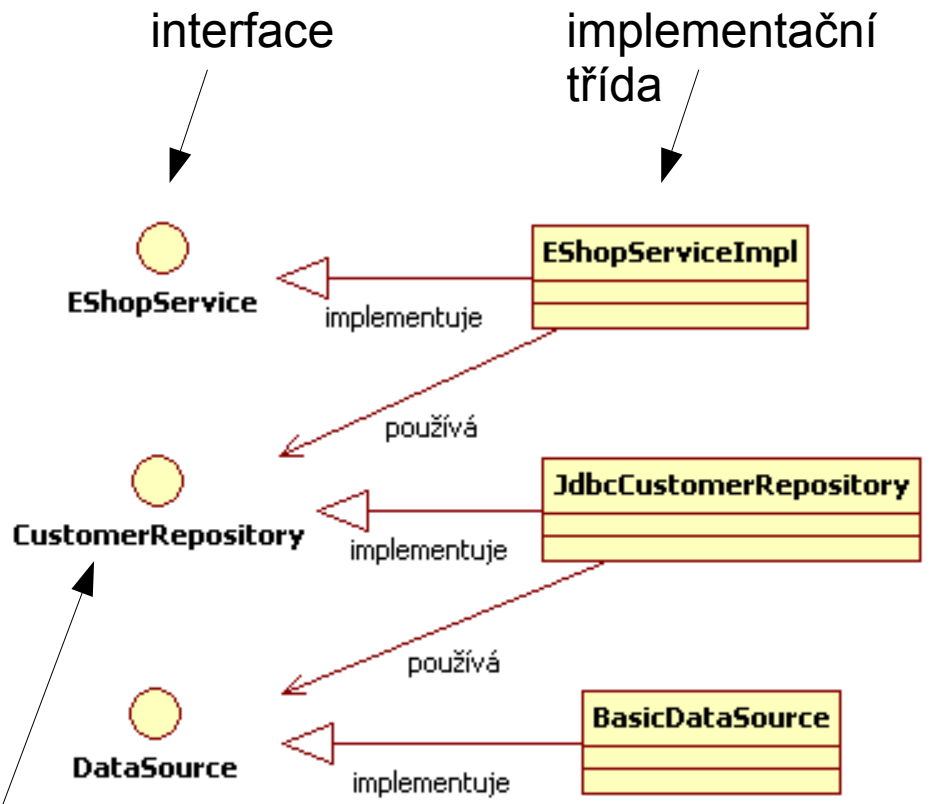


- Základem Spring kontejneru je IoC kontejner. Do tohoto kontejneru vstupují:
  1. **Java třídy**
  2. **Konfigurace**

- Spring kontejner z těchto dvou vstupů vytvoří plně nakonfigurovaný systém, připravený pro použití.

Vstupy do Spring kontejneru

- Na obrázku vpravo je zjednodušený UML diagram tříd, pomocí kterých můžeme získat data o zákaznících z databáze.
- Jedná se o klasický příklad použití třívrstvé architektury.
- Na školení budeme používat modernější zjednodušenou variantu, ve které místo JDBC připojení do databáze použijeme JPA.



Zjednodušeně:
Repository = DAO
(Data Access Object)

POJO třídy

```
public class JdbcCustomerRepository implements CustomerRepository {
    private DataSource dataSource;
    public JdbcCustomerRepository(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

Potřeba pro low-level práci s databází

```
public class EShopServiceImpl implements EShopService {
    private CustomerRepository customerRepository;
    public EShopServiceImpl(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }
}
```

Potřeba pro volání metod pro práci s databází

Typy konfigurace metadat

- Existuje několik způsobů, jak definovat Spring metadata:
  - **XML** konfigurační soubor Springu
  - **Anotace** (Spring anotace, nebo standardní JSR-330 anotace, které jsou od Java EE 6)
  - **Java Config** – konfigurace pomocí Java tříd

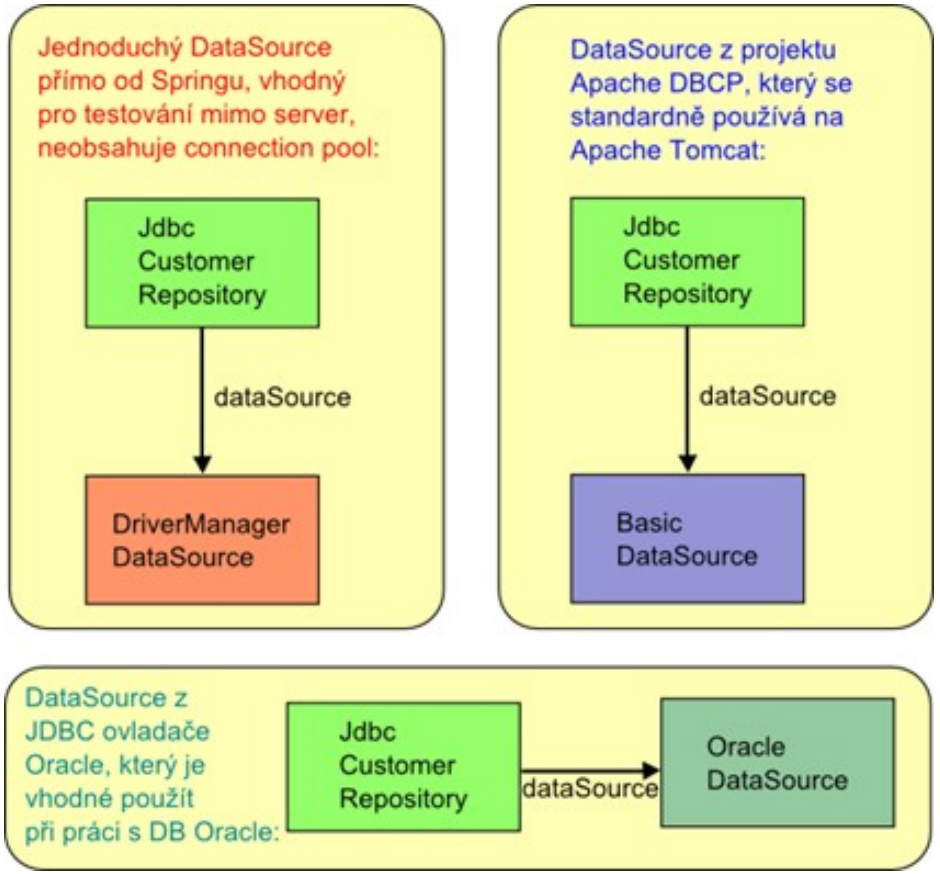
- Na následujících snímcích si postupně probereme tyto způsoby konfigurace metadat a nakonec uvedeme jejich výhody a nevýhody. Tyto způsoby konfigurace se v každém projektu obvykle kombinují.

Metadata

```
<beans>
    <bean id="dataSource" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
        <!-- konfigurace připojení do databáze -->
    </bean>

    <bean id="customerRepository" class="priklad.JdbcCustomerRepository">
        <constructor-arg name="dataSource" ref="dataSource" />
    </bean>

    <bean id="eshopService" class="priklad.EShopServiceImpl">
        <constructor-arg name="customerRepository" ref="customerRepository" />
    </bean>
</beans>
```

Dependency Injection

- V předcházejícím snímku byl pro přístup do databáze použit DataSource z DBCP projektu (`BasicDataSource`), který obsahuje connection pool.
- V případě testování aplikace nepotřebujeme přímo connection pool, ale stačí nám jednoduše jedno připojení do databáze. K tomu můžeme použít `DriverManagerDataSource`.
- V případě, že aplikace používá Oracle databázi, můžeme použít `OracleDataSource`.
- Změna jednoho typu DataSource za jiný je ve Springu pouhou drobnou změnou v konfiguraci.



Vytvoření a spuštění Spring kontejneru I.

- Spring kontejner je možné vytvořit a spustit několika způsoby. Každý ze způsobů odpovídá možnému použití Spring frameworku:
  - Ručně přímo v kódu aplikace – u Java SE aplikací (příklad níže)
  - Automaticky pomocí servletu u webové aplikace – u Java EE aplikací (bude u Spring Web MVC)
  - Automaticky v případě JUnit testovací aplikace – při tvorbě integračních testů (bude u JPA – Hibernate)

XML konfigurační soubor se hledá v classpath

```
//rucni vytvoreni kontextu
ClassPathXmlApplicationContext applicationContext =
   new ClassPathXmlApplicationContext("applicationContext.xml");
//nacteni beany eshopService
EShopService service = applicationContext.getBean(EShopService.class);
applicationContext.close();
```

Vytvoření a spuštění Spring kontejneru II.

- Pro načtení XML konfiguračního souboru mimo classpath je možné použít třídu `FileSystemXmlApplicationContext`.
- Pokud více preferujete anotace, pak můžete také použít třídu `AnnotationConfigApplicationContext`, kterou použijete následovně:

Kde se nachází volitelná XML konfigurace.
Může také obsahovat např. `file:c:/applicationContext.xml`

```
@Configuration
@ImportResource("classpath:applicationContext.xml")
@ComponentScan("cz.jiripinkas.example.eshop")
public class Application {
   public static void main(String[] args) {
   AnnotationConfigApplicationContext applicationContext =
     new AnnotationConfigApplicationContext(Application.class);
   // zbytek je stejný
   }
}
```

Kde se nacházejí třídy s anotacemi
(bude vysvětleno později)

Upozornění: Anotace jsou skvělé a většinou je vhodné je používat v maximální míře, ale v některých případech je efektivnější použít XML.

Spring XML konfigurační soubor

- Nejjednodušší Spring XML konfigurační soubor:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
  <!-- konfigurace bean -->
</beans>
```

- Můžete mít více Spring XML konfiguračních souborů. Aby je Spring zaregistroval, tak je buď musíte předat do konstruktoru třídy, která vytvoří instanci třídy `ApplicationContext`, nebo je můžete vzájemně importovat:

```
<import resource="dalsi_spring_xml_soubor.xml" />
```

Spring & Maven I. - jednoduchá aplikace

```
<properties>
   <spring.version>3.2.4.RELEASE</spring.version>
</properties>
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
   </dependency>
</dependencies>
```

Sem dejte aktuální verzi Springu

Základní knihovna Springu, většinou maximálně vhodná jenom pro „hello world“. Další často používané závislosti jsou `spring-web`, `spring-webmvc` atd.

Základní dependencies: http://projects.spring.io/spring-framework/

Všechny dependencies: http://spring.io/blog/2009/12/02/obtaining-spring-3-artifacts-with-maven/

Spring & Maven II. - webová aplikace

```
<properties>
   <spring.version>3.2.4.RELEASE</spring.version>
</properties>

<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>${spring.version}</version>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${spring.version}</version>
   </dependency>
</dependencies>
```

Závislosti potřebné ze Springu když chcete vyvíjet Java EE aplikace postavené na JPA (Hibernate), Spring a Spring Web MVC. Ostatní závislosti nejsou zapotřebí, protože se stáhnou transitivně.

Spring & Maven III. - logování I.

- Spring závisí na jediné knihovně a tou je Apache Commons Logging. Pokud ji nechcete používat a místo toho chcete používat SLF4J a Log4J nebo jiný logovací framework (např. Logback), pak udělejte následující:
- Nastavte exclusion u Spring dependencies na knihovnu `commons-logging`
- Pro logování pomocí Log4j přidejte následující dependency: `slf4j-api, jcl-over-slf4j, slf4j-log4j12, log4j`
- Přesný popis je na následujících stránkách.

Pokračování

Spring & Maven III. - logování II.

```
<!-- Spring dependencies -->
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-orm</artifactId>
   <version>${spring.version}</version>
   <exclusions>
      <exclusion>
         <artifactId>commons-logging</artifactId>
         <groupId>commons-logging</groupId>
      </exclusion>
   </exclusions>
</dependency>
```

Pokračování

Spring & Maven III. - logování III.

```
<!-- logging -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>${slf4j.version}</version>
</dependency>
```

Pokračování

Spring & Maven III. - logování IV.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>${log4j.version}</version>
</dependency>
```

Spring BOM

- Pokud byste používali více knihoven ze Springu (jako Spring Security nebo Spring Data JPA), pak byste narazili na jeden problém: Každá knihovna závisí na jiné verzi. Jak tento problém vyřešit? Přidejte do `pom.xml` toto a u Framework Spring dependencies neuvádějte tag `version`:

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-framework-bom</artifactId>
         <version>${org.springframework.version}</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

Nebo Spring IO Platform BOM:
http://platform.spring.io/platform/

Příklady XML konfigurace bean

- Nejjednodušší definice beany bez závislostí:

```
<bean id="testService" class="priklad.TestServiceImpl" />
```

To samé jako:

```
TestServiceImpl testService = new TestServiceImpl();
```

- Hodnota atributu „id“ musí být unikátní. Tento atribut je nepovinný, ale když ho neuvedete, tak nebudete moci získat odkaz na tuto beanu.
- V případě, že neuvedete hodnotu atributu class, pak se tato beana stane abstraktní. XML konfigurace je flexibilní v tom ohledu, že je u bean možné použít něco jako dědičnost.
- Místo atributu „id“ můžete alternativně používat atribut „name“. Ten má tu výhodu, že pomocí něj můžete definovat více názvů jedné beany a také v názvu beany můžete používat znaky jako je například lomítko.

Příklady XML konfigurace bean

- Constructor Dependency Injection:

```
<bean id="testService" class="priklad.TestServiceImpl">
  <constructor-arg name="repository" ref="repository" />
</bean>
```

- To samé jako:

```
TestServiceImpl testService = new TestServiceImpl(repository);
```

- Pomocí konstruktoru je možné nastavovat povinné atributy.
- Celkově bych ale Constructor Dependency Injection většinou nedoporučil a snažil se ji používat v minimální míře.
- Zkrácený způsob zápisu pomocí c namespace (od Spring 3.1):

```
<bean id="testService" class="priklad.TestServiceImpl"
      c:repository-ref="repository" />
```

Příklady XML konfigurace bean

- Setter Injection:

```
<bean id="testService" class="priklad.TestServiceImpl">
  <property name="repository" ref="repository" />
</bean>
```

- To samé jako:

```
TestServiceImpl testService = new TestServiceImpl();
testService.setRepository(repository);
```

- Pomocí setteru je vhodné nastavovat všechny atributy. I při použití setteru je ale možné nastavit povinnost nastavení takového parametru uvedením anotace `@Required` před setterem. V rámci definice jedné beany je možné kombinovat constructor a setter injection.
- Zkrácený způsob zápisu pomocí p namespace:

```
<bean id="testService" class="priklad.TestServiceImpl"
                       p:repository-ref="repository" />
```

Další XML konfigurace bean

- Beanu je možné vytvořit také pomocí statické factory metody (vhodné pro legacy aplikace, protože dříve tento způsob tvorby objektů byl velice populární).
- Beanu můžete také vytvořit pomocí instanční (nestatické) metody jiné beany.
- Spring je v tomto ohledu velice flexibilní.
- Všechny beany jsou typicky vytvořeny na začátku při tvorbě Spring contextu (**vytváření bean je tzv. eager**). Je možné toto změnit tak, aby se beany vytvářely podle potřeby při tvorbě aplikace (**nastavit vytváření bean na lazy**). Tím se výrazně urychlí start aplikace, ale na chyby v konfiguraci se poté může přijít až v průběhu běhu aplikace. **NEDOPORUČUJI!**

Konfigurace bean pomocí anotací

- Přestože je možné vytvořit kompletní konfiguraci aplikace pomocí XML, v dnešní době se velice často setkáte s jiným typem konfigurace – pomocí anotací. Anotace jsou rozšíření jazyka Java od verze Java 5 a mají stejný účel jako konfigurace v XML. Oproti XML mají ale tu výhodu, že se (obdobně jako javadoc) definují přímo u tříd a jejich zápis je kompaktnější než zápis v XML.
- Vzhledem k tomu, že jsou anotace alternativním způsobem konfigurace, je nutné jejich použití povolit. Existují různé typy anotací, které se aktivují různým způsobem. Univerzální způsob aktivace všech typů anotací je následujícím způsobem:

```
<context:component-scan base-package="cz.firma.nazev-projektu" />
```

- Postup:

1) Nejprve je nutné do XML konfiguračního souboru přidat schéma `context`
2) Do atributu `base-package` se zadá název balíčku, ve kterém se budou hledat anotace. Poznámka: anotace se budou hledat i ve všech pod-balíčcích definovaného balíčku.

Konfigurace bean pomocí anotací

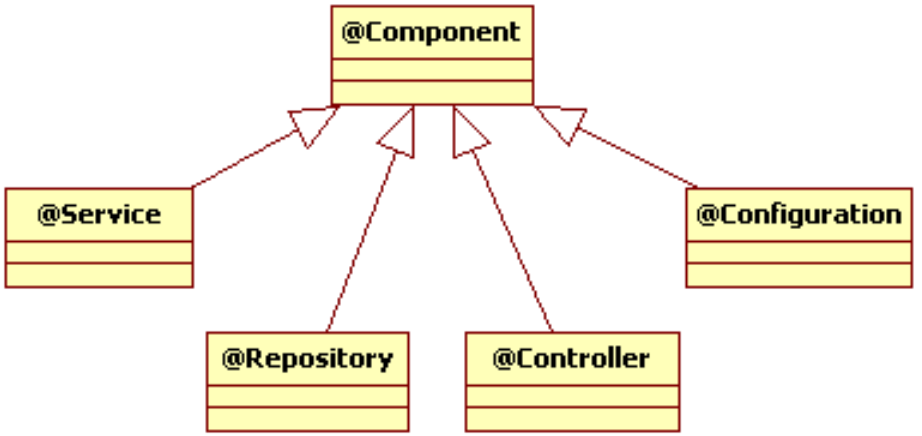
- Pokud používáte Java Config konfiguraci, pak anotace zapnete tímto způsobem:

```
@Configuration

@ComponentScan("cz.firma.nazev-projektu")

public class Main { }
```

- **Poznámka:** Nemusíte specifikovat pouze jeden balíček, ale i více balíčků, například:
  - `"cz.firma.nazev-projektu-1, cz.firma.nazev-projektu-2"`
- **Poznámka 2:** Toto lze taky, ale není to doporučené („oscanuje“ se opravdu hodně tříd):
  - `"cz, com, org"`
- **Poznámka 3:** Je také možné určité třídy ze „scanování“ vyloučit:
  - `@ComponentScan(value="cz.firma.nazev-projektu", excludeFilters=@ComponentScan.Filter(type=FilterType.ANNOTATION, value=Controller.class))`

Konfigurace bean pomocí anotací

- Pro definování beany můžete použít jednu z anotací uvedených na obrázku vpravo.
- V celé aplikaci můžete například používat pouze anotace `@Component`, ale vzhledem k tomu, že se Spring aplikace obvykle skládá z vrstev, je vhodné pro každou vrstvu používat jiný typ anotace.
- Pro servisní třídy se používají anotace `@Service`, pro repository anotace `@Repository` atpd.



+ @RestController (od Spring 4)

Konfigurace bean pomocí anotací

- Konfigurace pomocí anotací:
- Konfigurace pomocí XML:

Nepovinné. Pokud neuvedete, pak bude mít bean id rovno názvu třídy s malým prvním počátečním písmenem (např. u této beany by to bylo `testServiceImpl`)

```
@Service("testService")

public class TestServiceImpl
  implements TestService {

@Autowired
private Repository repository;
}
```

```
<bean id="testService"
      class="priklad.TestServiceImpl">
   <property name="repository"
      ref="repository" />
</bean>
```

Standardně se provádí autowiring pomocí typů. Pokud používáte interface, pak můžete mít více implementací takového interface. Spring v tom případě vyhodí chybu. Také pokud nenalezne vhodnou beanu, pak vyhodí chybu. Tato chyba se vyhazuje při startu Spring kontejneru (při startu aplikace). Pokud ve výjimečných situacích nechcete provádět autowiring pomocí typů, můžete použít `@Autowired @Qualifier("bean-id")`

@Autowired & Spring 4.3

- Autowired anotace může být před:
  - Atributem
  - Setterem
  - Konstruktorem
  - U metod s @Bean anotací také před argumentem metody
- Od Spring 4.3 je Autowired anotace nad konstruktorem nepovinná (pokud neexistuje jiný konstruktor). Nicméně je doporučené ji tam stále přidávat.

Konfigurace bean pomocí anotací

- V Java EE 6 byly uvedeny standardizované anotace (pod názvem JSR-330, neboli také @Inject). Od verze Spring 3.0 je možné tyto anotace používat. Umí skoro to samé co Springové anotace a jejich použití je úplně stejné:

Spring	javax.inject.*
`@Autowired`	`@Inject`
`@Component`	`@Named`
`@Scope`	`@Scope`
`@Scope("singleton")`	`@Singleton`
`@Qualifier`	`@Named`

- Po jejich uvedení se myslelo, že v nových aplikacích nahradí Spring anotace, ale nějak to nedopadlo … také kvůli tomu, že umí o malinko méně (nemají například @Service, @Repository apod.) a jejich použití je o malinko složitější.

@Inject anotace

- Pokud nepoužíváte aplikační server, pak pro podporu @Inject anotací přidejte do `pom.xml`:

```
<dependency>
   <groupId>javax.inject</groupId>
   <artifactId>javax.inject</artifactId>
   <version>1</version>
</dependency>
```

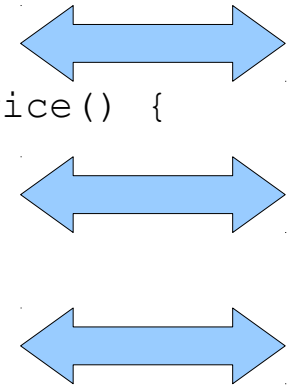
Další anotace

- `@Lazy`
  - Lazy inicializace beany (u @Component nebo @Bean) nebo atributu (u @Autowired)
- `@Order`
  - Pořadí inicializace bean
- `@DependsOn`
  - Pořadí inicializace bean kdy jedna vyloženě závisí na druhé

Konfigurace bean s Java config

- Java config konfigurace:

```
@Configuration
public class ServiceConfig {
 @Bean
 public TestService testService() {
   TestServiceImpl service =
     new TestServiceImpl();
   service.setRepository
    (repository());
   return service;
 }
 @Bean
 public Repository repository() { }
}
```



- Konfigurace pomocí XML:

```
<bean id="testService"
   class="priklad.TestServiceImpl">
   <property name="repository"
      ref="repository" />
</bean>
```

Kdy použít jakou konfiguraci?

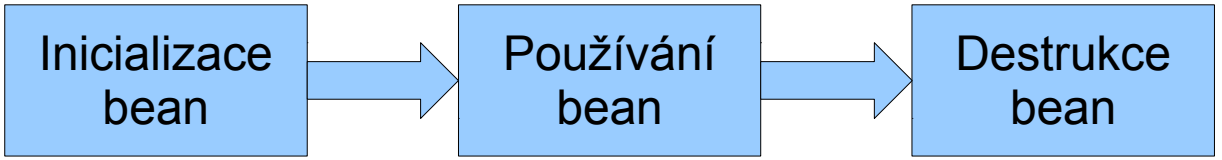
- Každý typ konfigurace má použití v jiných situacích:

- **XML konfigurace** nemá žádná omezení, ale je pracnější na vytvoření. Pomocí XML je možné přidat do Spring kontextu i beanu, ke které není zdrojový kód. XML konfigurace se používá pro integraci (např. připojení do databáze).

- **Anotace** jsou vhodné pro Vaše doménové objekty. Používání anotací výrazně urychluje vývoj. Anotace nemůžete použít u tříd, ke kterým nemáte zdrojový kód.

- **Java konfigurace** je vhodná pro nastavení Spring kontejneru bez použití XML a pro integrování legacy kódu, protože programátor má plnou kontrolu nad tvorbou Spring beany a její konfigurací.

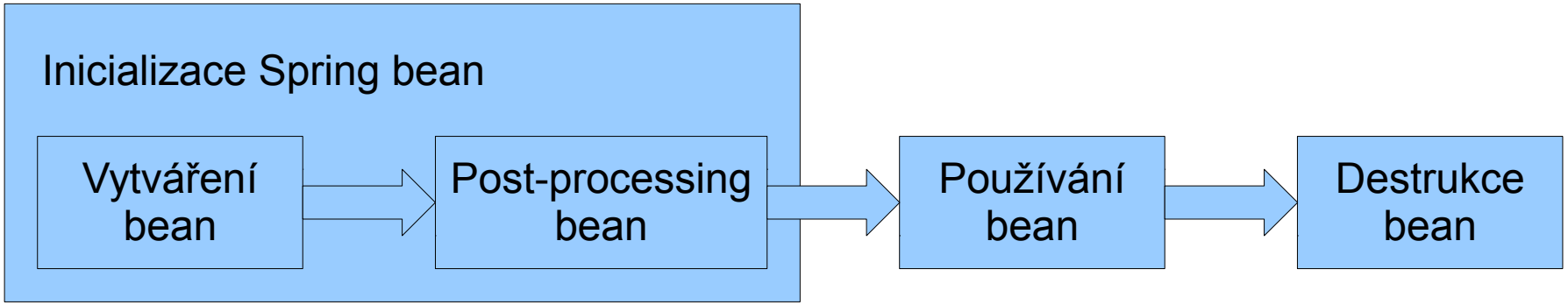
Životní cyklus Spring beany



- **Při vytvoření** Spring contextu se provede inicializace Spring bean (zavolají se konstruktory, nastaví se properties atpd.). Přitom je možné zavolat inicializační metodu beany.
- **Při rušení** Spring contextu se Spring beany destruují – odebírají se z paměti. Přitom je možné zavolat destrukční metodu beany.

Fáze životního cyklu	Anotace	XML
inicializace	@PostConstruct	init-method
destrukce	@PreDestroy	destroy-method

Životní cyklus Spring contextu



- Beany jsou instanciovány vždy ve správném pořadí. Jenom pozor na cyklické závislosti pomocí konstuktorů, tam se Spring dostává do klasického chicken-egg problému. Pokud Spring narazí na problém při vytváření bean, tak vyhodí výjimku a ukončí aplikaci.
- Při post-processingu bean se volají speciální beany, které mohou hromadně modifikovat definice bean ještě předtím, než jsou beany instanciovány. Právě to dělá tag `<context:property-placeholder>`, který načte konfiguraci z properties souborů a reálnými hodnotami nahradí šablonu definovanou v konfiguraci Springu.
- Více na další stránce.

<context:property-placeholder>

Konfigurační soubor Springu:

```
<context:property-placeholder location="classpath:jdbc.properties" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
   <property name="driverClassName" value="${jdbc.driverClassName}"/>
   <property name="url" value="${jdbc.url}"/>
   <property name="username" value="${jdbc.username}"/>
   <property name="password" value="${jdbc.password}"/>
</bean>
```

jdbc.properties:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:mem:test
jdbc.username=sa
jdbc.password=
```

Upozornění: tag `<context:property-placeholder>` je možné definovat maximálně jednou!

Hodnoty z properties souborů je také možné získávat přímo pomocí anotace `@Value`:

```
@Service
public class MailerService {

     @Value("${admin.email}")
     private String adminEmail;

}
```

Context property placeholder pomocí Java Config 1. způsob (>= Spring 3.1)

- Při použití tohoto způsobu není možné použít `@Value`!
- Konfigurace:

```
@Configuration
@PropertySource("classpath:config.properties")
public class ApplicationConfig {

@Autowired private Environment environment;

@Bean public AdminService adminService() {
   AdminService adminService = new AdminService();
   adminService.setAdminEmail(environment.getProperty("admin.email"));
   return adminService;
}
```

Context property placeholder pomocí Java Config 2. způsob (>= Spring 3.2)

- 1. konfigurace:

```
@Configuration
public class ApplicationConfig {
   @Bean public static PropertySourcesPlaceholderConfigurer configurer() {
      PropertySourcesPlaceholderConfigurer configurer
             = new PropertySourcesPlaceholderConfigurer();
      configurer.setLocation(new ClassPathResource("config.properties"));
      return configurer;
   }
}
```

Všimněte si klíčového slova static!

- 2. použití:

```
@Value("${admin.email}")
private String adminEmail;
```

Spring Profiles I.

- Od verze Spring 3.1 je možné používat profily. K čemu jsou dobré? Zejména k rozlišení testovacího a produkčního prostředí.
- Uvnitř Spring XML konfiguračního souboru můžete definovat profily následovně:

```
<beans profile="prod">
   <context:property-placeholder location="file:/opt/conf/db/jdbc.properties" />
</beans>

<beans profile="dev">
   <context:property-placeholder location="file:c:/konfigurace/db/jdbc.properties" />
</beans>
```

Tady je obecně definice Spring bean – například mimo jiné připojení do databáze apod.

- Nastavení profilu u konkrétní beany:

```
@Service @Profile("dev") public class MyTestService { }
```

Spring Profiles II.

- Můžete také vytvořit svoji vlastní anotaci a tím si zjednodušit definici profilu u konkrétní beany:

```
@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Profile("dev")

public @interface DevProfile {

}
```

- Použití u konkrétní beany:

```
@Service @DevProfile public class MyTestService { }
```

Spring Profiles III.

- Nastavení aktivního profilu:
  - U konzolové aplikace:

```
applicationContext.getEnvironment().setActiveProfiles("dev");
```

  - Při startu aplikace:

```
-Dspring.profiles.active="dev"
```

  - U JUnit testu:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ActiveProfiles(profiles = "dev")
@ContextConfiguration(locations = {"classpath:context.xml" })
```

Spring Profiles IV.

- Existuje speciální profil, který má název „`default`“. Pokud není specifikováno který profil má být aktivní, pak je aktivní tento profil.
- Změna defaultního profilu:
  - U konzolové aplikace:

```
applicationContext.getEnvironment().setDefaultProfiles("prod");
```

  - U webové aplikace ve `web.xml`:

```
<context-param>
   <param-name>spring.profiles.default</param-name>
   <param-value>prod</param-value>
</context-param>
```

@Conditional

- @Conditional (od Spring 4) je ještě flexibilnější mechanismus zapojení Spring bean než @Profile. Logika toho, jestli bude beana zapojena do Spring kontejneru je totiž řešena programově:
  - http://blog.codeleak.pl/2015/11/how-to-register-components-using.html

Scope Spring beany

- Definice beany je předpisem, pomocí něhož mohou být vytvářeny instance v různých oborech platnosti (scopes):
- **singleton** – výchozí scope. V rámci Spring contextu je vytvořena pouze jediná instance dané třídy. Ve většině případů je toto nastavení plně vyhovující (je logické mít v aplikaci jeden typ servisních tříd, repository tříd a po jednom datasource pro každou databázi, do které se aplikace připojuje).
- **prototype** – při každém požadavku na vytvoření instance třídy v rámci kontejneru se vytvoří nový objekt.
- **web scopes (request, session, global session)** – tyto obory platnosti je možné využívat pouze ve webové aplikaci. V případě requestu bude pro každý request vytvořena nová instance třídy a obdobně tomu je i v případě session. Global session má význam u portletové aplikace, kde jsou dva typy session scopů: request a application.