Spring Framework
tvorba obchodních aplikací v Javě bez J2EE

Firma Sun Microsystems dodává již několik let programovou platformu J2EE (Java 2 Enterprise Edition). Jedná se o standardizovanou distribuovanou komponentovou technologii doplněnou o mnoho knihoven a API, které jsou potřebné při tvorbě enterprise aplikací. Jedná se zejména o technologie Servlet, JSP/JSF, JNDI, JTA, JMS a webové služby. Jádro tvoří komponentová technologie, kterou si představíme.

Enterprise JavaBeans

Nejprve je nutno osvětlit, co je to vlastně enterprise aplikace – jedná se o softwarovou aplikaci, kterou firma vyvinula, zakoupila nebo převzala, poskytující strategické služby dané firmě. Může se jednat o podnikové libovolné procesy (účetnictví, sklad), ERP (Enterprise Resource Planning – plánování a správa výrobního procesu), CRM (Customer Relatationship Management – správa vztahu se zákazníky). Tento pojem nevymyslela firma Sun, ale je zde již řadu let. Dříve byly tyto aplikace úzce svázány s mainframy a například jazykem Fortran.

EJB jsou komponenty, které se nasazují na aplikačním serveru (tzv EJB kontejneru) poskytujícím bezpečné a spolehlivé prostředí nutný pro jejich běh. Tyto komponenty jsou dostupné a instalovatelné přes síť a jejich úlohou je poskytovat nástroj pro tvorbu enterprise aplikací. Jen připomenu, že softwarová komponenta má, na rozdíl od obyčejné třídy (např. v jazyce Java), jasně dané rozhraní, explicitně dané závislosti na jiných komponentách a musí být možné ji nasadit nezávisle jako produkt třetí strany.

Komponenty je možné představit si jako černé skříňky, které lze v systému (aplikačním serveru) vyměnit za jiné implementace (které si můžeme třeba zakoupit od jiného dodavatele). Může se jednat o primitivní komponentu, která umí odeslat e-mail, nebo o složitou komponentu schopnou předpovídat počasí – ta by se mohla skládat z několika (stovek) jiných komponent. EJB komponenty tohle všechno splňují a přidávají některé věci navíc, jako jsou popisy nasazení, napojení na stávající komponentové technologie (CORBA) a hlavně myšlenku jazyka Javy - „napiš to jednou a provozuj kdekoli“. Nutno říci, že toto Sunem prosazované heslo („Write once, run everywhere“) začala skutečně naplňovat až technologie J2EE a právě EJB.

Jak později uvidíme, enterprise aplikace se obvykle tvoří jako vícevrstvé. Právě vrstva aplikační logiky (business logic) je tvořena (obvykle více) EJB komponentami. Tyto komponenty mezi sebou pomocí rozhraní komunikují a je vhodné rozlišovat více typů komponent. Business objekt se obvykle svazuje s řešenou doménou a jedná se o nějakým způsobem perzistentní element (uložený například v databázi). Určitě by nebylo vhodné do business objektu psát aplikační logiku (i když by objektově orientované programování mohlo svádět k tomu prostě vytvořit metodu nad objektem). Abyste mohli business objekty znovu používat i v jiných aplikacích, je dobré aplikační logiku zapouzdřit v jiných komponentách. Těmito komponentami jsou takzvané aplikační kontrolery (vykonávají nějakou akci). Pokud se budeme bavit v termínech EJB, právě jsem zmínil entity beans a session beans. V aktuální verzi J2EE 1.4 ještě existují message driven beans, které nás v tuto chvíli nezajímají.

J2EE rozlišuje dva typy entity beans a dva typy session beans. V prvním případě dělíme business objekty na CMP (kontejner se stará o perzistenci sám) a BMP (programátor komponenty se stará o ukládání). Komponenty sessions beans s aplikační logikou dělí J2EE na stavové (stateful – pamatují si klienty) a bezstavové (stateless – nepamatují si své klienty). Kontejnery J2EE jsou navíc schopny transakčního zpracování operací session beanu. Příkladem prvního typu je nákupní košík, druhého typu pak komponenta pro zasílání SMS zpráv (klienta nezajímá, kterou z instancí kontejner vybere,

pokud jich má dostupných víc).

Tím bych ukončil základní přehled technologie Enterprise Java Beans, která tvoří jádro J2EE. K dalším tématům se dostaneme při rozboru Springu.

Motivace pro Spring Framework

Původní myšlenka padla v knize Roda Johnsona Expert One-on-One J2EE Design and Development, která bohužel nevyšla v češtině ani slovenštině. Rod v ní probírá sice základy J2EE, ale poukazuje na možné problémy v technologii jako samotné, ale zejména ve špatném aplikování těchto postupů.

Jako základní problém se jeví použití J2EE za situacích, kdy to vůbec není vhodné. Takové situace lze zjistit snadno – v případě, že nepoužití této technologie výrazně sníží složitost problému, a tedy i délku projektu. Nemusí se jednat nutně o komplexnost samotného programování, ztíženo může být i nasazení a přenositelnost (některé aplikační servery rozšiřují specifikaci, problémy s databázovou přenositelností). V následujících odstavcích představím hlavní faktory pro vznik tohoto projektu.

Asi největším problémem je **přílišná komplikovanost** J2EE, ačkoliv to nebylo hlavním motorem při vývoji Springu. Technologie jako taková obsahuje velké množství rozhraní, které jsou dále rozšiřovány dodavateli aplikačních serverů. Mnoho softwarových společností nedodržuje standardy a vznikají tak problémy při přenositelnosti. Technologie J2EE také trpěla některými nedostatky v návrhu, které sice byly eliminovány v dalších verzích, ale musela být zachována zpětná kompatibilita. Vývoj J2EE také nebyl nikterak rychlý a změny, které firmy požadovaly, se projevovaly příliš pomalu.

Spring umožňuje **snadné věci dělat jednoduše**, avšak při zachování vysoké škálovatelnosti a rozšiřitelnosti. Dále celé rozhraní klade velký **důraz na dobré praktiky** v programování (best practices) – například možnosti v testování. Spring je stále ve vývoji a rychle reaguje na nové techniky ve vývoji obchodních aplikací, které třeba jinde ještě nejsou aplikovány.

Možnosti **jednoduchých zásahů do samotného frameworku** jsou dalším kritériem, které by mohlo rozhodovat při volbě mezi J2EE a Springem. Spring je nesmírně modulární, jedná se o sadu rozhraní s mnoha různými implementacemi. V neposlední řadě je možné zasahovat přímo do zdrojového kódu (Spring jako takový je open-source softwarem), včetně možnosti vytvořit si vlastní aplikační rámec nad Springem a dále jej distribuovat, nebo se přímo účastnit vývoje a participovat se na dalších verzích.

Spring **nevyžaduje žádné závislosti**. Ačkoli má distribuce Springu kolem 100 MB, vlastní jádro není závislé na žádné knihovně a až s postupem projektu vývojový tým přidává nové závislosti. Není potřeba žádného aplikačního serveru, avšak ta možnost tady je.

Díky vpíchnutí závislostí (viz dále) **aplikace nejsou přímo závislé** na Springu a nic programátorovi tak nebrání aplikaci zprovoznit nad jinými aplikačními rozhraními. V neposlední řadě nabízí Spring **jednotnou konfiguraci**, která je také snadno pochopitelná.

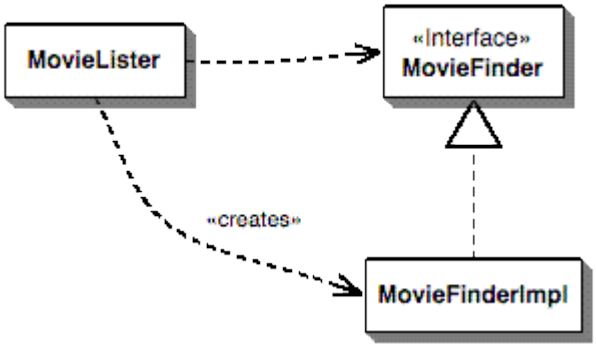
Vpíchnutí závislostí

Spring samotný je postaven na myšlence Martina Flowera, špičkového experta na poli objektově orientovaných přístupů, kterou původně nazval *Inversion Of Control* převrácení závislostí), ale vzápětí tento návrhový vzor přejmenoval na *Dependency Injection* (vpíchnutí závislostí). Poprvé tento návrhový vzor publikoval na internetu [2]. Jelikož pochopení pojmu vpíchnutí závislostí je pro další výklad nezbytné, rád bych jej zde uvedl. Na okraj bych ještě uvedl, že jsem se rozhodl

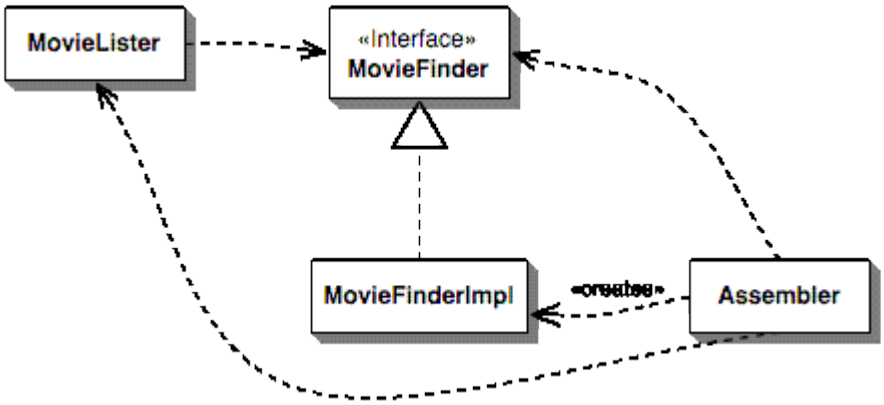
tento návrhový vzor pojmenovat jako „vpíchnutí“, protože se s českým výrazem daleko lépe pracuje.

Při návrhu jakékoliv komponentové technologie narazíte hned na začátku na jeden klíčový problém. Je nutné vytvářené komponenty nějak efektivně propojit do funkčních celků. Už z definice samotné komponenty vyplývá, že se musí jednat o samostatné a snadno vyměnitelné jednotky. V dnešní době, kdy počítačové sítě mají hlavní roli téměř ve všech odvětvích IT průmyslu, je také vhodné, aby byly komponenty dostupné také přes síť (popř. internet).

Je tedy nutné vymyslet způsob, jak komponenty mezi sebou skládat do aplikací. Většina komponentových technologií využívá vyhledávacích služeb (Service Locator, viz [3]) – každá komponenta se zaregistruje do určitého kontextu a pokud nějaká komponenta chce použít jinou, musí si ji explicitně přes předem dohodnutou vyhledávací službu najít. Všechny komponenty musejí obvykle implementovat různorodá rozhraní, aby je bylo možné v kontextu vyhledat a použít. Na následujícím obrázku je vidět, že komponenta MovieLister vytvoří instanci komponenty implementující rozhraní MovieFinder (například zmiňovaným způsobem, nebo jakkoli jinak).



Vpíchnutí závislostí vše obrací naruby, existuje jakýsi sestavovatel, který komponenty podle určitých pravidel inicializuje a nějakým způsobem jim předá odkazy na ostatní komponenty. Příklad je vidět na obrázku.



Zde vidíme komponenty MovieLister (což je patrně komponenta generující seznam filmů na základě nějakého dotazu), MovieFinder (komponenta určená pro vyhledávání v databázi včetně implementace) a Assembler (prvek zajišťující správné propojení komponent). Assembler tedy závislosti do objektů vloží, doslova vstříkne (to inject – vpíchnout jehlou).

***Poznámka**: Nejedná se tedy o žádnou zásadní změnu oproti Service Locatoru, jen je to jiný pohled na věc. Jako u všeho ostatního v programování se nedá říci, že je tento přístup lepší nebo horší – to je relativní. Stejně tak nelze říci, jestli je lepší Spring nebo J2EE – k různým účelům jsou vhodná různá řešení.*

Rozlišujeme tři typy realizace vstříknutí:

- interface injection (pomocí rozhraní)
- setter injection (pomocí vlastnosti objektu)
- constructor injection (pomocí konstruktoru)

Popis začnu trošku netradičně – dvěma posledními. Vlastní předání odkazů na komponenty se prování přímo v konstruktoru komponenty, respektive přes set metody (přes vlastnosti objektů). Martin Flower vše ukazuje na jednoduché knihovně PicoContainer, kterou vytvořili jeho kolegové z firmy, ve které Martin působí. Jedná se o malou java knihovnu, která nabízí všechny popisované typy vstříknutí závislostí, příklady a dokumentaci. Je dostupná jako open-source software na adrese *http://www.picocontainer.org/*, kde jsou odkazy i na implementace v jiných jazycích.

Představme si, že máme třídu

```
class MovieLister {
     ...
     public MovieLister(MovieFinder finder) {
          this.finder = finder;
     }
}
```

Tato třída očekává ve svém konstruktoru komponentu MovieFinder, jejíž implementace by mohla vypadat následovně

```
class SQLMovieFinder implements MovieFinder {
     ...
     public SQLMovieFinder(String dbServer) {
          this.server = dbServer;
     }
}
```

Parametrem je jakési spojení do databáze, kde jsou uloženy informace o filmech. Nyní stačí assembleru vytvořit instance těchto tříd a v konstruktoru třídy MovieLister uvést odkaz na SQLMovieFinder. Nebudu se zde pouštět do popisu, jak to dělá PicoContainer, na kterém vše vysvětluje Martin Flower ve své práci, protože to není cílem mého textu. Zůstaneme jen u toho, že je nutné toto dělat za běhu, takže PicoContainer je schopný pracovat se třídami, které vůbec „nezná“.

Jistě si dokážete představit, jak by vypadalo vpíchnutí pomocí set metody. Konstruktor by v tomto případě neměl žádné parametry a odkazy na komponenty by se předávaly až po vytvoření objektu pomocí vlastností. Hlavním přínosem je zde možnost vytvářet instance komponent pomocí skriptovacích nebo značkovacích jazyků, například XML. Mohlo by to vypadat například následovně:

```
     <component id="MovieLister" class="package.MovieLister">
         <property name="finder">
             <ref local="MovieFinder"/>
         </property>
     </component>
     <component id="MovieFinder" class="package.ColonMovieFinder">
         <property name="server">
             <value>phoenix:4398</value>
         </property>
     </component>
```

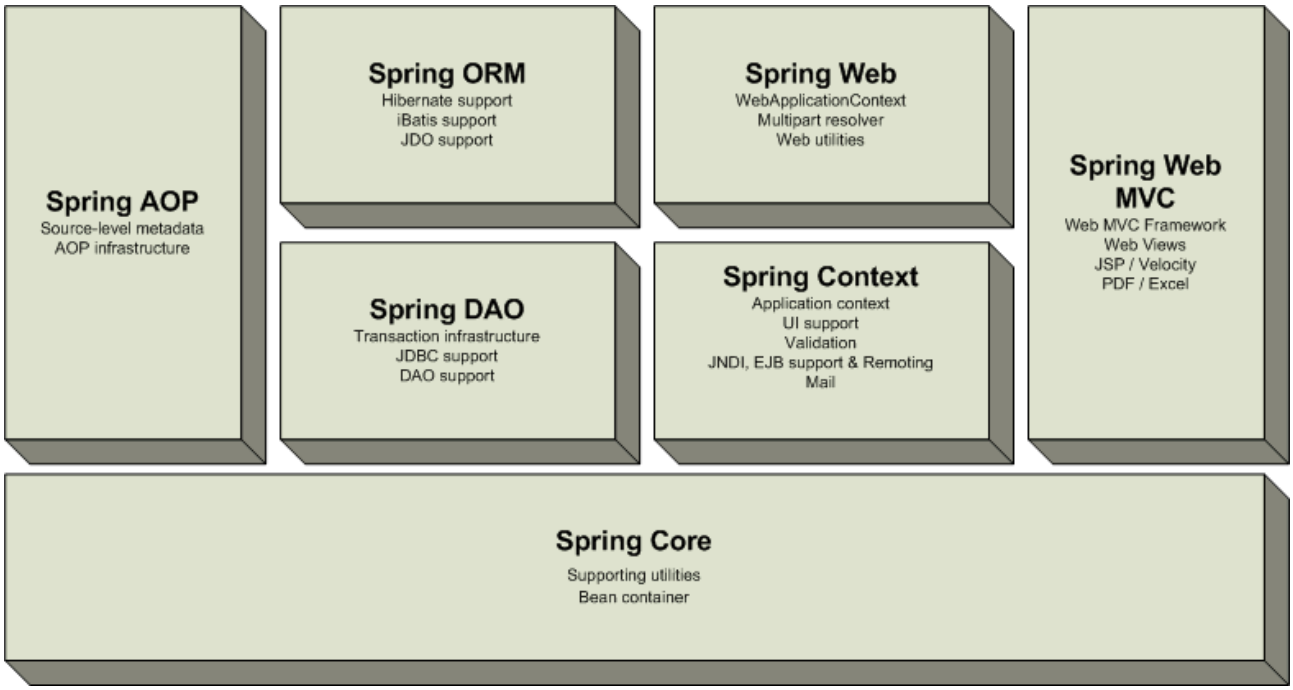
Třetí metodou je vpíchnutí pomocí rozhraní, kdy se pro vložení závislosti vytvoří speciální rozhraní, což by v tomto našem příkladě mohla být například metoda **injectFinder(MovieFinder)**. Tento způsob umožňuje o něco větší abstrakci, což je vykoupeno poněkud vyšší režií.

Existují dva hlavní (open-source) projekty založené na vpíchnutí závislostí a to je Apache Avalon a NanoContainer. První ze jmenovaných se stal základem pro několik produktů nadace Apache Free Foundation a využívá vpíchnutí pomocí rozhraní. NanoContainer rozšiřuje možnosti PicoContaineru o skriptování (závislosti lze definovat pomocí libovolného jazyka, včetně XML), nasazení (deployment, composition skripty), vzdálené volání (remoting), perzistenci objektů, AOP a webové služby. Na rozdíl od Avalonu je NanoContainer je ale stále malou knihovnou a nevyžaduje téměř žádné závislosti.

Představení Spring Frameworku

Hlavní myšlenka této knihovny (jedná se vlastně o sadu knihoven – spolu s dokumentací se jedná o více jak 100 MB dat) je ve vytvoření jednoduché, ale účinné komponentové technologie založené na moderních postupech, a podobně jako u J2EE na jazyku XML. Spring se snaží pokrýt veškeré oblasti spojené s vývojem obchodních aplikací s důrazem na databáze, web a aspektově orientovaného programování.

Spring je tedy jakési lepidlo mezi mnoha open-source knihovnami a snaží se co nejlepším způsobem zobecnit základní API sloužící pro programování obchodních aplikací. Základem je komponentová technologie, dále pak kontext aplikace (podpora konfigurace, prostředků, UI, validace, adresářových služeb či zasílání pošty), databázová vrstva (DAO, transakce), O/R mapování a webové programování (modely, pohledy, způsob propojení modelů). K tomu všemu ještě Spring přidává AOP.



Jak vidíme na obrázků, klíčovou vrstvou je jádro nazvané **Spring Core**, které představuje vlastní komponentovou technologii (paralela s EJB), dále kontejner pro tyto komponenty a potřebné nástroje. K tomuto účelu se využívá popsaná technologie vpíchnutí závislostí. Nad tímto jádrem se nacházejí balíky (dokumentace je doslova takto nazývá, protože vlastní implementace v Javě je

rozdělena do právě těchto balíčků), které bych detailněji popsal.

Asi nejdůležitějším je **Spring Context**, který obsahuje aplikační kontext a podpůrné komponenty pro tvorbu uživatelského rozhraní, pro validaci vstupů, posílání zpráv a e-mailů a v neposlední řadě se zde realizuje velmi důležité napojení na EJB. Spring je totiž kompatibilní s EJB a můžete v něm tyto komponenty jak vytvářet, tak používat.

Spring DAO zastřešuje všechny klíčové databázové technologie a ačkoliv by se mohla další DB vrstva zdát zbytečná, velmi brzy poznáte, že i JDBC je nutno nějakým způsobem sjednotit (například velmi nejednotné jsou výjimky a chybové zprávy jednotlivých databázových dodavatelů). Nemluvě o to, že tento balík přidává jednotnou podporu pro transakce a mnoho dalších nástrojů zjednodušujících práci s databází.

Nad tímto balíkem stojí **ORM** – tedy objektově relační mapování. Spring nepoužívá žádnou obdobu entity java beans, jako je tomu u EJB. Je to výhodné, protože mnoho softwarových společností vůbec entity EJB komponenty nepoužívalo a psalo si vlastní DB vrstvy. Spring tedy vkládá abstraktní vrstvu mezi jádro (resp. DAO) a dodavatele ORM technologií. Se Springem se distribuuje několik adaptérů na různé open-source ORM knihovny včetně nejpoužívanější – projektu Hibernate.

Balík pro **AOP** nabízí napojení na aspektově orientované programování v Javě. Tomuto tématu se ve svých přednáškách budou věnovat moji kolegové, nicméně jen bych dodal, že AOP jde ruku v ruce právě s Dependency Injection přístupem a AOP je tedy základním stavebním kamenem celého popisovaného frameworku.

Balíky **Spring Web** a **Web MVC** společně zobecňují webové aplikační rozhraní. V Javě existuje obrovské množství webových technologií, tedy postupů, jakým způsobem obhospodařit webový (např. http) požadavek, poslat ho příslušným komponentám (tzv akcím), vykonat nezbytnou činnost (nad nějakými business) a korektně vrátit klientovi výsledek. Jelikož většina webových protokolů je bezstavová, k tomuto se ještě přidává nutnost uchovávat sezení. Díky tomuto balíku můžete sjednotit práci s webovými frameworky a nic vám nemůže bránit snadno nahradit technologii Servlet technologií Struts nebo WebWork, technologii JSP technologií JSF a podobně.

Spring jako komponentová technologie

V následující části si přiblížíme samotné komponenty ve Springu. Komponenta je ve Springu obyčejná javovská třída (POJO – Plain Old Java Object – starý dobrý obyčejný objekt). Díky dostatečné podpoře platformy Java (reflexe, RMI...) není nutno zavádět žádná specifika a jelikož je celý framework určen právě pro tuto platformu Java, není divu, že se komponenty nazývají názvem Beans (resp. dlouze: Java Beans). Jako Bean se totiž označuje libovolný java objekt, který má jednu nebo více vlastností.

Jak jsem se již zmínil, Spring využívá vpíchnutí závislostí a úlohu assembleru přebírá jádro Springu. Závislosti komponent se načítají z XML souboru, zde je krátký příklad:

```
<bean id="exampleBean" class="examples.ExampleBean">
  <property name="database"><ref bean="db"/></property>
  <property name="numConnections"><value>32</value></property>
</bean>

<bean id="db" class="examples.DatabaseBean"/>
```

K výpisu netřeba žádný komentář a pomalu si dovolím odkazovat na dokumentaci, která je ve Springu poměrně rozsáhlá, úplná a přehledná (což u mnoha open-source projektů bohužel nebývá

zvykem). Spring nabízí také vpíchnutí pomocí konstruktoru, stačí nahradit značku *property* značkou *constructor-arg*.

Definiční soubory XML mají promyšlenou strukturu a je zde myšleno na sebemenší detail – máte možnost předávat veškeré možné hodnoty (čísla, řetězce), odkazy na jiné objekty či null hodnotu. Samozřejmá je podpora java kolekcí (List, Set, Map, Properties), singletonů, abstraktních tříd a abstraktních továren. V definičních souborech se dají používat i pokročilé techniky jako jsou runtime delegace a výměna metody, bližší informace jsou v dokumentaci.

Spring je schopný závislosti vytvářet automaticky (tzv. autowiring mode). Nabízí hned několik typů autowiringu, díky kterým je dokonce možno aplikaci nastavit tak, že není nutné při vytváření dalších a dalších komponent (a nutných závislostí) znovu zasahovat do XML konfigurace (což je někdy značně zdržující). Nutno podotknout, že při rozsáhlejších aplikacích pak nemusí být zřejmé, která komponenta obsahuje jinou. Autowiring mód lze nastavovat pro každý bean zvlášť, což je velká výhoda.

Komponenty v Beanu nemusejí povinně implementovat žádné metody (init, destroy a podobně). Teprve až když je nutné explicitně kontrolovat životní cyklus komponenty, je možno implementovat definovaná rozhraní (InitializingBean, DisposableBean). Podobné je to s komponentou BeanFactory, což je objekt zajišťující čtení vlastní konfiguraci – až když ji komponenta potřebuje, implementuje určité rozhraní. Tady je zásadní rozdíl oproti J2EE.

Všechny aspekty jádra lze dobře rozšiřovat, takže pokud nebudete mít možnost pomocí XML souboru nastavit nějaký objekt, který nebude primitivním typem nebo základní třídou Javy, můžete si snadno vytvořit takzvaný PropertyEditor.

Spring Context

Součástí frameworku je kontext aplikace usnadňující některé operace, které musí programátor často řešit. Prvním je API pro jednotný přístup k prostředkům (soubor na disku, v kontextu webové aplikace, na síti nebo z CLASSPATH). Podpora snadné lokalizace je také součástí tohoto balíku.

Podpora validace vstupů jde ruku v ruce s navázáním vstupních hodnot na doménové objekty. K těmto účelům se používají rozhraní DataBinder a Validator. K těmto účelům se využívá standardního API definovaného přímo firmou Sun pro JavaBeans, pomocí kterého si Spring „hlídá“ změny v datových objektech a tyto hodnoty validuje pomocí Validator API.

Validace probíhá tak, že v XML souboru stačí u vlastnosti zadat daný validátor a ten implementovat. V něm můžete použít předdefinované validátory pro jednotlivé vlastnosti objektu (JavaBeanu) . Pokud nám nebude vyhovovat některý z vestavěných (například celé číslo, řetězec o minimální délce, nebo regulární výraz), můžeme definovat vlastní:

```
public class PersonValidator implements Validator {

        public boolean supports(Class clzz) {
                return Person.class.equals(clzz);
        }

        public void validate(Object obj, Errors e) {
                ValidationUtils.rejectIfEmpty(e, "name",
"name.empty");
                Person p = (Person)obj;
```

```
                if (p.getAge() < 0) {
                        e.rejectValue("age", "negativevalue");
                } else if (p.getAge() > 110) {
                        e.rejectValue("age", "tooold");
                }
        }
}
```

Aspektově orientované programování

Ačkoliv AOP tvoří nedílnou součást Spring Frameworku a jeho autor navrhoval všechna rozhraní tak, aby byly „AOP-kompatibilní“, tak se nebudu této části příliš věnovat. Důvodem je zejména fakt, že bych kopíroval kolegy, kteří mají přednášky na toto téma a jistě popíší celou problematiku podrobněji.

Spring nevyužívá asi nejpoužívanější knihovnu AspectJ, ale rozhraní AOP Alliance, které je taktéž poskytováno jako open-source. Spring se totiž nesnaží pokrýt všechny možnosti, které AOP nabízí, ale zaměřuje se zejména na tu oblast, která je dobře využitelná v postupech, které Spring využívá. Ovšem v plánech pro verzi 1.1 je kompletní podpora pro AspectJ.

Transakční zpracování

V Javě existuje velké množství různých rozhraní a standardizovaných API pro transakční zpracování (ať už pouze o databázové jako například JDBC, Hibernate či iBatis, nebo více generické, jako například JTA). Spring opět plní úlohu jakéhosi „lepidla“ a poskytuje obecné rozhraní, aby bylo možné použít libovolné API a později jej vyměnit (například JDBC a v případě nutnosti vyměnit za JTA).

V EJB je nutné striktně rozlišovat globální a lokální transakce (transakce například na úrovni databázového spojení). Také se zde používá rozhraní JNDI a transakce jsou obvykle závislé na aplikačním serveru (vyžadují jeho nasazení – CMT – Container Management Transacitons). V případě Springu používáte vždy jednotné rozhraní a nevyžaduje použití aplikačního serveru, pokud to není nutné.

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition
definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws
TransactionException;

    void rollback(TransactionStatus status) throws
TransactionException;
}
```

Na výpisu vidíme API (jedná se opět o rozhraní a připomínám, že Spring je vlastně hlavně o rozhraních, abstraktních pomocných třídách a přehledné dokumentaci). Metoda getTransaction přímo vyhledá vhodnou transakci na základě její definice, kde specifikujeme izolování, timeout a podobně.

Následující příklad ukazuje použití Hibernate Transaction Manageru. Nejprve si zadefinujeme datový zdroj (spojení na SQL databázi).

```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
  <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>
```

Dále je třeba mít nastavený Hibernate O/R mapper, o kterém budeme ještě mluvit.

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="mappingResources">
    <list>
      <value>samples/petclinic/hibernate/petclinic.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
    </props>
  </property>
</bean>
```

Vytvoření komponenty *txManager*, která bude poskytovat transakční služby, je pak velmi jednoduché.

```
<bean id="txManager"
class="org.spring.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

To je vše. Všimněte si, že vše jsou komponenty (z pohledu Springu), což v EJB tak úplně neplatí (tam se pracuje přímo s rozhraními JNDI, JTA a podobně). Stejně jako v EJB však Spring rozlišuje transakce na úrovni aplikace (programmatic transaction support), kde je přístup podobný rozhraní JTA (Java Transaction API) s tím rozdílem, že můžete implementaci nahradit jinou knihovnou.

V případě, že budete potřebovat využít služeb některého z aplikačních serverů, pak Spring poskytuje implementace konektorů k serverům BEA WebLogic a IBM Websphere.

Značkování v komentářích

Jazyk Java verze 1.5 zavedl popisování zdrojových kódů pomocí metadat. Jelikož Spring je vyzrálý produkt (vznikl v době, kdy byla produkční verze 1.4) a jelikož autor Springu nechce nutit programátory, aby stávající projekty převáděli do nové verze jazyka Java (které v některých případech **není** zpětně kompatibilní a vyžaduje zásahy do zdrojových kódů), využívá Spring služeb projektů XDoclet a commons-attributtes, což jsou open-source implementace Source-level Metadat.

Spring vše zobecňuje do té úrovně, že je možno přistupovat k těmto implementacím zcela nezávisle, navíc dodává podporu i pro Javu 1.5. Pokud se tedy jedná o starší projekt, může využít služeb XDocletu, v případě nového pak například nativní podpory v Javě 1.5. Například platforma .NET má svoji implementaci metainformací již od svých raných verzí.

A jaká je hlavní výhoda source-level metadat? Programátor získává možnost přidávat ke třídám, metodám, vlastnostem a atributům zvláštní informace (takříkajíc informace „navíc“ - metainformace). Typickým příkladem může být označení určité třídy jako DAO objekt (databázový doménový objekt). Spring pak může vyhledat všechny DAO objekty v aplikaci a provést například jejich převod, vytvořit SQL strukturu a podobně. Na ukázce vidíme atribut PoolingAttribut, který označuje danou třídu pro znovupoužití.

```
 /**
 * @@org.spring.aop.framework.autoproxy.target.PoolingAttribute(10)
 * @author Rod Johnson
 */
public class MyClass {
```

Zde je vidět jedna z nevýhod implementace metadat v Javě 1.5 – tam totiž jsou všechna metadata uložená ve zkompilovaném class souboru a není možné je dynamicky měnit. Pokud bychom například chtěli bez rekompilace změnit maximální počet objektů v poolu (na příkladě 10), museli bychom použít XDoclet. Tato vlastnost by se měla objevit v Javě 6.0.

DAO a objektově-relační mapping

Najde se jen málo obchodních aplikací bez databází, proto Spring nabízí několik metod pro přístup k nim. Použitá technika DAO je nezávislá na tom, který přístup si vyberete. Spring se nesnaží jako J2EE vytvořit vlastní vrstvu pro ukládání objektových dat, ale používá k tomu špičkové open-source projekty, které jsou ověřené trhem i časem.

Ve Springu můžete používat přímo rozhraní JDBC, přičemž se Spring snaží zastřešit a sjednotit některé věci. Jednak je to vlastní přístup k databázi (otevření spojení) a dále obsluha výjimek, která je v současné verzi JDBC nedostatečně standardizovaná a liší se od dodavatele databáze. Spring také definuje API zastřešující SQL dotazy pro zjednodušení vytváření mapujících tříd, což jsou třídy převádějící výsledky volání JDBC vrstvy na databázové objekty.

Hlavní síla Springu je bezesporu ve výborné integraci s knihovnami Hibernate, Oracle TopLink, Apache OJB nebo iBatis. Poskytují spojení mezi doménovými objekty a relační databází. Princip spočívá v tom, že JavaBeans komponenty a jejich vlastnosti jsou namapovány (pomocí konfiguračních souborů) do databázových tabulek. Knihovna samotná se postará o jejich správné vyzvednutí, ošetřuje kolekce (spojování dotazů), výjimky a cachování.

Zde je ukázka použití Hibernate, což je defacto standard mezi O/R mapovacími systémy. Firma Sun Microsystems sice vytvořila standardizovanou API JDO, ale ta přišla jednak dost pozdě (projekt Hibernate je jeden z nejstarších) a hlavně verze 1.0 nedosahovala takových kvalit a možností, jako Hibernate.

Nejdříve je nutno nadefinovat JDBC spojení (Hibernate pracuje nad JDBC), pak velmi důležitou komponentu – SessionFactory. Je to vlastně taková „instance“ knihovny Hibernate jako takové. V ní se provádí hlavní konfigurace (včetně nastavení mapovacích XML souborů, které lze pomocí různých nástrojů generovat automaticky).

```
<beans>

  <bean id="myDataSource"
class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName"
value="org.hsqldb.jdbcDriver"/>
    <property name="url"
```

```
value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>

  <bean id="mySessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop
key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop
>
      </props>
    </property>
  </bean>

   ...
</beans>
```

Vlastní dotaz v DAO je pak následující:

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where
product.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

Transakce můžeme na v nejvyšší úrovni řešit také zajímavým způsobem – anonymními třídami:

```
public class ProductServiceImpl implements ProductService {

    private PlatformTransactionManager transactionManager;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager
transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
```

```
    }

    public void increasePriceOfAllProductsInCategory(final String
category) {
        TransactionTemplate transactionTemplate = new
TransactionTemplate(this.transactionManager);
        transactionTemplate.execute(
            new TransactionCallbackWithoutResult() {
                public void
doInTransactionWithoutResult(TransactionStatus status) {
                    List productsToChange =
productDAO.loadProductsByCategory(category);
                    ...
                }
            }
        );
    }
}
```

Na Hibernate (i jiné mapovační rozhraní) může programátor přistupovat i na nižší úrovni (tj. ručně zahájit transakci, vrátit seznam objektů z dotazu, ošetřit výjimky, commitnout transakci).

Ačkoli O/R mapovací rohzraní tvoří základ obchodních aplikací, nebudu se tomuto tématu nadále věnovat a přejdu k závěrečně a asi nejzajímavější fázi.

Webová část

Webová část zastřešuje kompletní MVC přístup pro tvorbu webových aplikací. Ačkoli programátorovi nic nebrání použít jiný/vlastní aplikační rámec (Struts, Tapestry, Velocity/Freemaker, XLST, Tiles, JasperReports), použití implementace ve Springu má řadu výhod:

- *čistá separace všech rolí* – v mnoha MVC rámcích se slučují pojmy jako form a model, validator a model a podobně,
- *jednoduchá konfigurace* – Spring používá samozřejmě na všechno JavaBeans komponenty a s tím je spojena řada dalších výhod,
- *znovupoužití business tříd* – nemusíte vytvářet další objekty typu Form, ve Springu použijete přímo business komponenty,
- *nazávislé mapování* – handlery a pohledy jsou nezávislé a můžete použít více přístupů.

Spring obsahuje přímou podporu JSP a také jakousi ad-hoc podporu (zatím) pro novou technologii JSF.

Všechny klíčové komponenty (Controller, Model, View) jsou součástí Springu a ten navíc obsahuje mnoho připravených implementací. Typická aplikace pomocí Spring MVC je nakonfigurovaná tak, aby na straně J2EE kontejneru předávala všechny dotazy na jeden jediný servlet. Tento servlet pak načítá vlastní konfiguraci a mapování z XML souborů. Všechno jsou znovupoužitelné komponenty a velkou výhodou je velká provázanost s již vytvořenými business objekty ve Springu.

Další technologie

Vyčerpali jsme hlavní technologie, které se používají při budování obchodních aplikací. Spring toho však nabízí více. Dobrou podporu má pro webové služby (RMI, JAX-RPC), také zmíněná možnost vytvářet EJB komponenty patří do této části. Technologie JMS je navíc doplněna o Spring email infrastructure. Posledním článkem je JCA CCI (Java Connector Architecture – Common Client Interface), což je technologie pro výměnu obchodních informací v rámci EIS.

Literatura a odkazy:

[1] Rod Jonhson: **Expert One-on-One J2EE Design and Development**, Wiley Publ. 2003

[2] Martin Flower: **Inversion of Control Containers and the Dependency Injection pattern**, *http://www.martinfowler.com/articles/injection.html*

[3] Sun Microsystems, **Core J2EE Patterns**, Sun Press, *http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html*

[4] Das Spring Framework als Teil eines Paradigmenwechsels? Vergleich der leichtgewichtigen Alternative zur traditionellen J2EE Entwicklung, diplomová práce. *http://www.martinmaier.name/archives/5*

[5] Sum Microsystems, J2EE Portal, oficiální dokumentace technologie J2EE, *http://java.sun.com/j2ee/*