

Eine Einführung in die neuen Features von Spring 3.1

Spring 3.1 – Was gibt's Neues?



Spring 3.1 steht vor der Tür – und mit ihm eine Reihe neuer, interessanter Features. Zwei davon stellen wir in diesem Artikel vor: die neue Environment Abstraction, die mit ihren Bean Definition Profiles umgebungsspezifische Bean-Definitionen ermöglicht, und die Cache Abstraction, die ein allgemeines API für Cache-Funktionalitäten mit Unterstützung verschiedener Implementierungen anbietet.

von Tobias Flohre und Pascal Czollmann

Jede Anwendung durchläuft von der Entwicklung über die Testphase bis zum produktiven Einsatz unterschiedliche Lebensphasen. In diesen stehen in der Regel unterschiedliche Ressourcen zur Verfügung, seien es Hard- und Software, Persistenzmedien oder Fremdkomponenten. Für die Konfiguration einer Anwendung bedeutet das zumindest unterschiedliche Konfigurationsdaten wie Username/Password bei einer *DataSource*. Es können sich in den Umgebungen aber auch Infrastrukturkomponenten unterscheiden. So kann es sein, dass man in einer Entwicklungsumgebung einen *DataSourceTransactionManager* verwendet, während in der Produktion ein JTA-Transaktionsmanager Anwendung findet.

Umgebungsspezifische Spring-Konfigurationen werden meistens per *PropertyPlaceholderConfigurer* oder austauschbarer *ApplicationContext*-Dateien realisiert. Per *PropertyPlaceholderConfigurer* können einzelne Zeichenketten aus der Spring-Konfiguration in eine Properties-Datei ausgelagert werden. So können beispielsweise Verbindungsdaten für eine Datenbank für unterschiedliche Umgebungen konfigurierbar gehalten werden (Listing 1). Was aber, wenn man in der einen Umgebung die *DataSource* direkt erstellen möchte, sie in der anderen Umgebung aber per JNDI aus einem JEE-Container holen möchte? Hier enden die Möglichkeiten des *PropertyPlaceholderConfigurer*, da keine Bean-Definitionen ausgetauscht werden können. Mit austauschbaren *ApplicationContext*-Dateien ist das Problem lösbar, aber bei einer Zuststeuerung per Build-Prozess ist das Anwendungsartefakt auf eine Umgebung spezialisiert. Das *import*-Tag kann Platzhalter auflösen, die entweder in JVM-System-Properties oder in Umgebungsvariablen definiert sind:

```
<import resource="{env}-applicationContext.xml" />
```

Spring 3.1 konsolidiert und erweitert die bisherigen Möglichkeiten des Umgebungsmanagements. Zunächst betrachten wir die neuen Bean Definition Profiles.

Bean Definition Profiles

Wir gehen im Folgenden von einer Anwendung aus, die im Infrastrukturbereich für den Datenzugriff einen Transaktionsmanager und eine *DataSource* benötigt. In der Entwicklungsumgebung läuft unsere Anwendung auf einem Tomcat. Eine lokale *DataSource* wird erzeugt, ein *DataSourceTransactionManager* regelt das Transaktionsverhalten. Als Integrations- und Produktionsumgebung läuft unsere Anwendung auf einem Application Server, daher werden Transaktionsmanager und *DataSource* per JNDI aus dem Verzeichnis des Java-EE-Containers geholt.

Natürlich können diese beiden Konfigurationen nicht gleichzeitig geladen werden, allein schon die identischen Bean-IDs schaffen Probleme. Andererseits wollen wir aber ein Anwendungsartefakt schaffen, sei es ein *jar*, *war* oder *ear*, das in allen Umgebungen lauffähig ist. Wir brauchen also einen Mechanismus, der es erlaubt, bei Bedarf bestimmte Spring Beans zu aktivieren oder zu deaktivieren.

Genau dafür werden in Spring 3.1 die Bean Definition Profiles [1] eingeführt. Das *beans*-Tag, also das Wurzel-Tag einer Spring-XML-Konfiguration, wird um

Listing 1

```
<context:property-placeholder location="classpath:jdbc.properties" />
<bean id="dataSource" class="{jdbc.dataSourceClassName}">
  <property name="driverClassName" value="{jdbc.driverClassName}" />
  <property name="url" value="{jdbc.url}" />
  <property name="password" value="{jdbc.password}" />
  <property name="username" value="{jdbc.username}" />
</bean>
```

das Attribut *profile* erweitert. Ist dieses Attribut gesetzt, werden die innerhalb des Tags definierten Bean-Definitionen nur hinzugefügt, wenn ein Umgebungsprofil mit dem entsprechenden Namen aktiviert ist.

Seit Spring 3.1 ist es auch möglich, *beans*-Tags ineinander zu schachteln. So können Spring-Beans verschiedener Umgebungen in einer Datei definiert werden. Die Konfiguration unserer Anwendung sieht nun so aus wie in Listing 2. Dabei haben wir jeweils für die Entwicklungs-, Integrations- und Produktionsumgebung ein Profil erstellt. Die Konfigurationen für Integrations- und

Produktionsumgebung sind zwar identisch, trotzdem mag es außerhalb dieser Konfigurationsdatei Unterschiede in der Konfiguration geben.

Mit der Annotation *@Profile* kann dieses Feature übrigens auch außerhalb von XML-Konfigurationen verwendet werden [2].

So weit, so gut. Aber wir haben keines der Profile aktiviert, also werden keine Spring Beans erzeugt. Wie aktiviert man also ein Umgebungsprofil?

Environment

Hier kommt die neue Umgebungsabstraktion ins Spiel. Ab Spring 3.1 bringt jeder *ApplicationContext* ein Objekt mit, das das Interface *Environment* implementiert. Dieses Objekt muss alle aktiven Profile kennen.

Schauen wir zuerst auf den programmatischen Ansatz (Listing 3). Haben wir den *ApplicationContext* zur Hand, so können wir das aktive Profil direkt am *Environment*-Objekt setzen. Das Setzen des aktiven Profils muss vor dem Erzeugen der Beans, also vor dem Aufruf der *refresh*-Methode, stattfinden.

In der Regel wollen wir die Entscheidung für ein bestimmtes Profil nicht im Code treffen. Daher ist es möglich, Profile über die Property *spring.profiles.active* zu aktivieren. Diese kann kommasepariert Profilenames der zu aktivierenden Profile enthalten und über viele mögliche Quelle eingelesen werden. Hier kommt die zweite Aufgabe des *Environment* ins Spiel: das Management von *PropertySources*.

PropertySources

Eine *PropertySource* [3] ist eine Quelle für Properties. Das *Environment* beinhaltet eine Hierarchie von *PropertySources*. Wenn unsere Anwendung eine Property anfragt (Listing 4), werden die *PropertySources* der Reihe nach gefragt, ob sie die Property kennen.

Das *DefaultEnvironment* registriert automatisch *PropertySources* für die JVM-System-Properties und die Systemumgebungsvariablen. Dabei steht die *PropertySource* für die JVM System Properties in der Hierarchie weiter oben, da sie in der Regel spezialisierter sind als die Umgebungsvariablen, die ja für alle JVMs auf der Maschine gelten.

Wollen wir also nun das Profil 'dev' aktivieren, so können wir unsere JVM mit dem Parameter *Dspring.profiles.active=dev* starten oder eine Umgebungsvariable mit dem Namen *spring.profiles.active* und dem Wert *dev* erzeugen.

In einer Webumgebung werden automatisch weitere *PropertySources* für Daten aus der *web.xml* registriert. Wahlweise können auch eine *JndiPropertySource* aktiviert und auch eine eigene *PropertySource* implementiert und der Umgebung hinzugefügt werden.

Nutzung der Properties

Bisher haben wir einen Anwendungsfall für Properties gesehen, nämlich die Property *spring.profiles.active*, die von Spring genutzt wird, um die aktiven Umgebungs-

Listing 2

```
<beans ...>
  <beans profile="dev">
    <bean id="transactionManager" class="org.springframework.jdbc.datasource.
DataSourceTransactionManager">
      <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driverClassName" value="org.SomeDriverClassName"/>
      <property name="url" value="someUrl"/>
      <property name="password" value="somePassword"/>
      <property name="username" value="someUsername"/>
    </bean>
  </beans>

  <beans profile="int,prod">
    <tx:jta-transaction-manager/>
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
  </beans>
</beans>
```

Listing 3

```
GenericXmlApplicationContext applicationContext =
  new GenericXmlApplicationContext();
applicationContext.getEnvironment().setActiveProfiles("dev");
applicationContext.load("META-INF/applicationContext.xml");
applicationContext.refresh();
```

Listing 4

```
GenericXmlApplicationContext applicationContext =
  new GenericXmlApplicationContext();
String propertyActiveProfiles = applicationContext.getEnvironment().getProperty("spring.
profiles.active");
```

Listing 5

```
@Cacheable("books")
public Book findBook(ISBN isbn) { ... }

@Cacheable({"media", "dvds"})
public DVD findDVD(String title) { ... }
```

profile zu setzen. In Listing 1 haben wir gesehen, wie man in einer Spring-Konfiguration mithilfe des Tags *property-placeholder* Platzhalter mit Werten aus einer Properties-Datei ersetzen kann. Ab Spring 3.1 wird nun bei Verwendung des Tags der *PropertySourcesPlaceholderConfigurer* registriert, der zunächst in den definierten Dateien sucht und danach die *PropertySources* des *Environment* durchgeht. Auch bei der Platzhalterersetzung im *import*-Tag werden nun alle *PropertySources* durchsucht. Wir haben insgesamt also eine höhere Flexibilität, da die Werte aus beliebigen Quellen stammen können.

Zwischenfazit

Die Environment Abstraction fasst bereits bestehende und neue Funktionalitäten für das Umgebungsmanagement auf sinnvolle und erweiterbare Art und Weise zusammen. Die Bean Definition Profiles ermöglichen es, Beans bestimmten Profilen zuzuweisen. Nur wenn die Profile aktiviert werden, werden auch die Bean-Definitionen geladen und die Beans erstellt. Eine *PropertySource* ist eine beliebige Quelle von Properties. Jedes Environment beinhaltet eine Hierarchie von *PropertySources*, die frei konfigurierbar und erweiterbar ist, aber sehr sinnvolle Defaults enthält. Zusammenfassend ist festzustellen, dass dieses Feature eine bisher vorhandene

Lücke in der Konfiguration von Spring-Anwendungen mit der für Spring typischen Offenheit für Erweiterungen schließt.

Cache Abstraction

Der Einsatz eines Caches lohnt vor allem dann, wenn Daten häufig angefragt werden, sich jedoch selten ändern. Wiederholte Anfragen werden dann aus dem Cache bedient und können somit schneller beantwortet werden. Beispiele hierfür sind aufwändige Datenbankabfragen, Anfragen an langsame Drittsysteme, zeitintensive Berechnungen etc.

Im Java-Umfeld gibt es verschiedene Cache-Implementierungen, z. B. Ehcache, JBoss Cache und dessen Nachfolger Infinispan. Der Einsatz von Caches ist bislang codeinvasiv, d. h., die Anbindung des Caches erfolgt programmatisch durch Nutzung des jeweiligen Cache-API. Die daraus folgende Vermischung von Business- und Infrastrukturcode resultiert in schlechterer Test- und Lesbarkeit des Codes.

Die neue Cache Abstraction in Spring 3.1 implementiert Caching als Aspekt. An welchen Stellen Caching zum Einsatz kommen soll, wird auf Methodenebene deklarativ mittels Annotations bestimmt, analog zu deklarativem Transaktionsmanagement. Spring fängt die Methodenaufrufe mittels AOP ab und fügt das Caching-

Listing 6

```
@Cacheable(value="books", key="#isbn.rawNumber")
public Book findBook(ISBN isbn, boolean checkWarehouse) { ... }

// Schlüssel wird mittels statischer Methode aus beiden Parametern gebildet
@Cacheable(value={"media", "dvds"},
    key="T(my.media.KeyGen).createKey(#p0, #p1)")
public DVD findDVD(String title, int year) { ... }
```

Listing 7

```
@Cacheable(value="books", condition="#name.length < 32")
public Book findBook(String name) { ... }
```

Listing 8

```
@CacheEvict(value="books", key="#p0.rawNumber")
public void deleteBook(ISBN isbn, Book book) { ... }

@CacheEvict(value="books", allEntries=true)
public void dropAndReloadAllBooks() { ... }
```

Listing 9

```
<beans ...
  xmlns:cache="http://www.springframework.org/schema/cache"
  xsi:schemaLocation="...
    http://www.springframework.org/schema/cache
    http://www.springframework.org/schema/cache/spring-cache-3.1.xsd">

  <cache:annotation-driven/>

  <bean id="cacheManager" class=
    "org.springframework.cache.support.SimpleCacheManager">
    <property name="caches">
      <set>
        <bean class=
          "org.springframework.cache.concurrent.ConcurrentCacheFactoryBean">
          <property name="name" value="books"/>
        </bean>
        <bean class=
          "org.springframework.cache.concurrent.ConcurrentCacheFactoryBean">
          <property name="name" value="media"/>
        </bean>
      </set>
    </property>
  </bean>
```

Listing 10

```
<bean id="cacheManager" class=
  "org.springframework.cache.ehcache.EhCacheCacheManager">
  <property name="cacheManager" ref="ehcache"/>
</bean>

<bean id="ehcache" class=
  "org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
  <property name="configLocation" value="ehcache.xml"/>
</bean>
```

Verhalten hinzu. Für den Businesscode ist das völlig transparent – weder der aufrufende Code einer Methode noch die Methode selbst enthalten Cache-spezifischen Code.

@Cacheable

Das Caching wird mittels der Annotation `@Cacheable` an Methoden deklariert (Listing 5). Die Annotation markiert, dass der Rückgabewert der Methode gecached werden soll, und kann daher logischerweise nicht an *void*-Methoden eingesetzt werden. Der Schlüssel, unter dem der Wert im Cache abgelegt wird, wird per Default aus den `hashCode()`-Werten der Parameter der Methode gebildet. Bei einem erneuten Aufruf prüft die Cache Abstraction, ob es zu den Parameterwerten bereits einen Eintrag im Cache gibt. Falls ja, wird direkt der Wert aus dem Cache zurückgegeben, ohne dass die Methode erneut aufgerufen wird [4].

Listing 5 zeigt die Anwendung der Annotation. In dem gezeigten Fall werden die Rückgabewerte der Methode `findBook` in dem Cache `books` abgelegt – in einem Cache liegen in der Regel gleichartige Daten. Es ist auch möglich, mehrere Caches anzugeben. Sie werden dann der Reihe nach geprüft, und sobald ein Cache einen Treffer liefert, wird dieser zurückgegeben. Aktualisierungen werden hierbei in allen angegebenen Caches durchgeführt.

Das Ergebnis der `hashCode()`-Methode für zwei unterschiedliche Objekte muss laut der JDK-Dokumentation aber nicht unbedingt unterschiedliche Werte ergeben. In einem solchen Fall ist der per Default aus den `hashCode()`-Werten der Parameter gebildete Schlüssel unbrauchbar, da es dann zu Kollisionen der Schlüssel kommt, d. h., es würde für unterschiedliche Parameter der gleiche Schlüssel erzeugt. Aus diesem Grund bietet die Annotation `@Cacheable` das Attribut `key`, mit dem sich die Schlüsselerzeugung steuern lässt (Listing 6). Es erlaubt die Verwendung von Ausdrücken der Spring Expression Language (SpEL) und ist somit sehr flexibel einsetzbar. Beispielsweise können bestimmte Methodenparameter anhand ihres Namens oder per Index – z. B. `#p0` für den ersten Parameter – für die Schlüsselerzeugung ausgewählt werden. Letzteres ist sinnvoll, da die Parameternamen nur genutzt werden können, wenn entsprechende Debug-Informationen in den kompilierten Klassen vorliegen. Abweichend von der Referenzdokumentation muss den Parametern ein `#`-Zeichen vorangestellt werden [5]. Mittels Punktnotation ist es möglich, auf Properties der ausgewählten Parameter zuzugreifen. Komplexere Schlüsselberechnungen, auch über mehrere Parameter, lassen sich in eine statische Methode auslagern, die dann mittels SpEL in der `@Cacheable`-Annotation eingebunden werden kann.

Ein weiteres nützliches Feature ist das Conditional Caching. Dabei wird nur dann das Caching aktiviert, falls eine definierte Bedingung erfüllt ist, die mittels SpEL im Attribut `condition` der Annotation angegeben werden kann (Listing 7).

@CacheEvict

@CacheEvict ist die zweite Annotation der Cache Abstraktion [4]. Sie wird an Methoden eingesetzt, die dazu führen, dass Informationen im Cache nicht mehr aktuell sind und daher aus dem Cache entfernt (evicted) werden müssen. Die Annotation bietet die gleichen Attribute wie @Cacheable und darüber hinaus lässt sich mittels *allEntries=true* angeben, dass alle Daten aus dem spezifizierten Cache verworfen werden sollen, nicht nur der zu dem Schlüssel passende Eintrag. Im Gegensatz zu @Cacheable kann @CacheEvict auch an void-Methoden eingesetzt werden und im Fall *allEntries=true* benötigt die Methode keine Parameter.

Konfiguration

Spring bietet derzeit Unterstützung zur Einbindung von zwei Cache-Implementierungen: den Concurrent Cache, basierend auf der JDK-Klasse *ConcurrentHashMap*, sowie die Open Source Library Ehcache [4]. Concurrent Cache lässt sich einfach einbinden, erfordert keinen zusätzlichen Konfigurationsaufwand und ist dank *ConcurrentHashMap* auch in Multithreading-Umgebungen sehr performant. Andererseits werden wichtige Enterprise-Anforderungen nicht abgedeckt, beispielsweise Verteilung und Transaktionssicherheit. Daher bietet sich Concurrent Cache vor allem für einfache Anwendungsfälle wie Standalone-Anwendungen oder zur lokalen Entwicklung und zum Unit Testing an. Ehcache deckt hingegen Anforderungen ab, die im produktiven Java-Enterprise-Umfeld wichtig sind, wie verteiltes Caching, Transaktionssicherheit, Persistenz, Eviction Policies etc.

Die Konfiguration der Cache Abstraktion erfolgt in der *applicationContext.xml* unter Verwendung des neuen XML Namespace <http://www.springframework.org/schema/cache>. Das Element `<cache:annotation-driven/>` aktiviert die Auswertung der Caching-Annotationen. Über Attribute kann hierbei festgelegt werden, ob Spring AOP oder AspectJ zum Einsatz kommt und ob im Fall Spring AOP die Proxies auf Interface- oder Klassenebene erzeugt werden. Die Standardeinstellung ist Spring AOP und die Nutzung Interface-basierter Proxies.

Die Einbindung eines Cache erfolgt nun durch Konfiguration einer *CacheManager*-Instanz. Dieses Spring-Interface abstrahiert von konkreten Cache-Implementierungen und bietet Zugriff auf die konfigurierten Caches. `<cache:annotation-driven/>` erwartet den *CacheManager* per Default unter der Bean-ID *cacheManager*.

Die Concurrent-Cache-Implementierung wird über die Klasse *SimpleCacheManager* eingebunden. Sie dient lediglich als Container für ein oder mehrere Caches, die in der Property *caches* definiert werden. Instanzen des Concurrent Cache werden mittels der Factory Bean *ConcurrentCacheFactoryBean* konfiguriert. Der Name des Cache wird über das Property *name* der Factory Bean gesetzt. Unsere Konfiguration sieht nun aus wie in Listing 9.

Der Einsatz von Ehcache erfordert nur wenige Zeilen in der *applicationContext.xml* (Listing 10). Als Erstes erstellen wir die Ehcache-Instanz unter Verwendung der Spring-Factory-Bean *EhCacheManagerFactoryBean*, die einen Ehcache-Manager erzeugt. Die Ehcache-Konfigurationsdatei wird hierbei über die Property *configLocation* der Factory Bean gesetzt. Die in den vorherigen Beispielen verwendeten Caches *media* und *books* werden nun in der Konfigurationsdatei von Ehcache definiert. Anstelle des *SimpleCacheManager* aus Listing 9 verwenden wir die Spring-Klasse *EhCacheCacheManager* und geben ihr in der Property *cacheManager* die Referenz auf den zuvor konfigurierten Ehcache-Manager mit.

Fazit

Mit der Cache Abstraktion führt Spring 3.1 eine wichtige Abstraktion für Enterprise-Java-Anwendungen ein. Caches haben eine hohe Relevanz für Clustering und Failover und können – richtig eingesetzt – den Datendurchsatz vieler Anwendungen massiv steigern. Analog zu deklarativem Transaktionsmanagement etabliert die Cache Abstraktion deklaratives Caching mittels Annotationen und ermöglicht damit eine effektive Trennung von Infrastruktur- und Businesscode. Die Cache Abstraktion ist so gestaltet, dass Adapter für weitere Caches mit überschaubarem Aufwand entwickelt werden können. Derzeit werden nur zwei Cache-Implementierungen angeboten, doch dabei wird es in Zukunft nicht bleiben. Im Rahmen des Spring-GemFire-Projekts ist für das nächste Major Release 1.1 bereits die Integration des SpringSource GemFire Cache geplant.



Tobias Flohre arbeitet als Senior Software Engineer beim IT-Dienstleistungs- und Beratungsunternehmen adesso AG. Seine Schwerpunkte sind Java-Enterprise-Anwendungen und Architekturen mit JE/Spring.



Pascal Czollmann ist als Senior Software Engineer bei dem IT-Dienstleistungs- und Beratungsunternehmen adesso AG beschäftigt. Er arbeitet dort seit mehreren Jahren in unterschiedlichen Kundenprojekten im Java-Enterprise-Umfeld.

Links & Literatur

- [1] <http://blog.springsource.com/2011/02/11/spring-framework-3-1-m1-released/>
- [2] <http://blog.springsource.com/2011/02/14/spring-3-1-m1-introducing-profile/>
- [3] <http://blog.springsource.com/2011/02/15/spring-3-1-m1-unified-property-management/>
- [4] <http://static.springsource.org/spring/docs/3.1.0.M1/spring-framework-reference/html/cache.html>
- [5] <http://blog.springsource.com/2011/02/23/spring-3-1-m1-caching/>