JAVA Mag inkl. CD

Open Source BPM: Tools und Strömungen ▸86

S&S

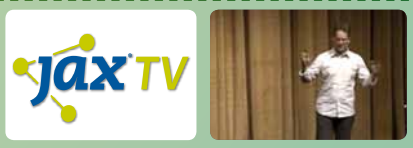
1.2012

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

CD-INHALT



DEVOPS-KEYNOTE
von Matthias Marschall

Video von der W-JAX 2011

NIO2

Erweiterungen der I/O-Funktionalität ▸27

Product Line Engineering mit OSGi

Ein sinnvolles Paar? ▸105

jax 2012

Erste Sessions ab Seite ▸43

DevOps

Sonderdruck für
www.codecentric.de

codecentric

WEITERE INHALTE

- H-Ubu
- Apache POI 3.8 beta 4
- Activiti 5.8

Alle CD-Infos ab Seite 3

Infrastructure as Code mit Chef ▸▸ 53

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

GWT & HTML5
Das große Web-Tutorial ▸S. 60

Google Dart
Die neue Programmiersprache ▸S. 75

Spring ohne XML
Java-basierte Konfiguration ▸S. 96

Spring komplett ohne XML – geht das?

Java-basierte Konfiguration

Mit Spring 3.1 wird die Java-basierte Konfiguration erwachsen, da es mit den @Enable-Annotationen nun Äquivalente zu den XML-Namespaces gibt. Höchste Zeit also, sich diese auf den ersten Blick ungewohnte Art und Weise, eine Spring-Anwendung zu konfigurieren, genauer anzusehen und mit den bisher etablierten Varianten XML und Annotationen zu vergleichen.

von Tobias Flohre

Seit Spring 3.0 gibt es nun schon im Kernbereich des Frameworks die Möglichkeit, direkt per Java-Code eine Spring-Anwendung zu konfigurieren. Dass mir diese Funktionsweise bisher in Projekten kaum begegnet, mag verschiedene Gründe haben. Zum einen existieren mit XML- und Annotations-basierter Konfiguration gleich zwei Alternativen, die ebenfalls ihre Vorteile haben, die man kennt und die funktionieren – was will man mehr? Und zum anderen war es bis jetzt nicht wirklich bequem möglich, ganz auf XML zu verzichten. Das ändert sich nun mit Spring 3.1 – höchste Zeit also, sich noch einmal intensiver mit Java-basierter Konfiguration (kurz: JavaConfig) zu befassen, denn sie bietet einige Vorteile gegenüber den herkömmlichen Varianten. Um eine gute Vergleichbarkeit zu gewährleisten, will ich in diesem Artikel die drei Varianten jeweils an demselben Beispiel vorstellen und Vor- und Nachteile dieser Variante benennen. Während XML und Annotationen relativ kurz abgehandelt werden, wird auf die JavaConfig genauer eingegangen.

Die Beispielanwendung

Was muss eine gute Beispielanwendung leisten? Sie sollte realistisch sein, Features beinhalten, die in einer echten, produktiven Anwendung unerlässlich sind, aber ihre Konfiguration sollte auch mit einem Blick erfassbar sein. Meine Anwendung besteht zunächst aus drei Komponenten: einer *DataSource*, einem Repository, das mithilfe der *DataSource* Schreib- und Leseoperationen auf einer Datenbank ausführt, sowie einem Service, der Businesslogik auf den vom Repository gelieferten Daten durchführt (**Abb. 1**, Listing 1).

Zwei weitere Standardanforderungen habe ich noch an die Anwendung: Die Verbindungsdaten zur Datenbank sollen aus einer Properties-Datei ausgelesen werden, außerdem sollen ausgeführte Aktionen transaktional abgesichert werden. Das sind sicherlich Basisanforderungen, die fast jede Anwendung hat, und doch sind sie ausreichend, um die relevanten Spring-Mechanismen darzustellen. Schauen wir zunächst auf die Konfiguration mit XML.

XML-basierte Konfiguration

Insgesamt besteht unsere Konfigurationsdatei aus sieben Einträgen (Listing 2). Dazu gehören die drei schon beschriebenen Komponenten und deren Verdrahtung, zusätzlich wird ein *DataSourceTransactionManager* benötigt, der für das Management der Transaktionen auf der *DataSource* zuständig ist. Das Tag *property-placeholder* sorgt dafür, dass die Platzhalter, die sich in der *DataSource*-Konfiguration befinden, durch die Daten aus der angegebenen Properties-Datei ersetzt werden. Und der Transaktionsaspekt ist hier so konfiguriert, dass er über Aufrufe auf allen Komponenten, die sich im Package *de.codecentric* befinden, Transaktionsklammern setzt.

In einem echten Projekt kommt man natürlich nicht mit einem Service und einem Repository aus, womit wir zum ersten Nachteil kommen: *XML-Konfigurationsdateien neigen dazu, sehr lang und unübersichtlich zu werden*. Best Practice ist es, die Konfigurationen themenbezogen zu splitten, also beispielsweise eine Datei nur für Datenzugriff und Transaktionen, eine nur für Sicherheitsaspekte und mehrere für die eigentliche Anwendung zu erstellen. Problematisch ist dabei aber, dass die *Navigierbarkeit in XML-Dateien nicht gegeben* ist, und woher will man wissen, wo genau die Spring Bean namens *xyService* definiert ist, die hier referenziert wird? Häufig bleibt da nur die Volltextsuche, und schon befinden wir uns wieder in der Steinzeit des Programmierens. Auch *Tippfehler* und die anschließende Fehlersuche sind nicht zu vernachlässigen, wie ich in Schulungen häufig bemerken musste. *Typsicherheit* ist in einem XML-File natürlich auch *nicht gegeben*; niemand hindert einen daran, Objekte vom Typ X in ein Feld vom Typ Y zu injizieren – hässliche Exceptions sind garantiert. All diese Nachteile bekommt man auch mit dem entsprechenden Tool nur halbwegs in den Griff, und schön wird das dadurch noch lange nicht.

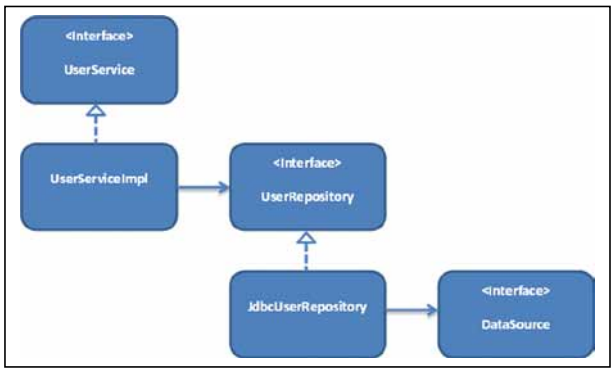


Abb. 1: Komponentenschaubild Beispielanwendung

Einen Vorteil sehe ich jedoch auch in XML-Konfigurationen: die *Konfiguration der Anwendung befindet sich an einer dedizierten Stelle*. Gibt es Probleme mit der Konfiguration, weiß ich, wo ich nachsehen muss.

Annotations-basierte Konfiguration

XML ist out, das haben auch die Spring-Jungs schon vor längerer Zeit erkannt und eine Möglichkeit geschaffen, mit der man zumindest seine eigenen Komponenten per Annotationen konfigurieren kann. Ganz auf XML verzichten kann man dabei aber nicht, denn Komponenten, die man nicht selbst in der Hand hat, müssen weiterhin per XML konfiguriert werden. In unserem Fall sind das die *DataSource* und der *DataSourceTransactionManager* (Listing 3). Ebenfalls erhalten bleibt uns der *property-placeholder*. Neu hinzu kommen das Tag *component-scan*, das dafür sorgt, dass alle Klassen im Package *de.codecentric* nach Spring-Komponenten gescannt werden, und das Tag *tx:annotation-driven*, das dafür sorgt, dass Informationen zur Konfiguration von Transaktionalität ebenfalls in den Klassen gesucht werden.

Was müssen wir tun, damit eine Klasse als Spring-Komponente erkannt wird? Sie muss entweder mit

Listing 1

```
public class UserServiceImpl implements UserService {
    private UserRepository userRepository;
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
    // business code here
}

public class JdbcUserRepository implements UserRepository {
    private JdbcTemplate jdbcTemplate;
    public JdbcUserRepository(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    // data access code here
}
```

Listing 2

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

<bean id="userService" class="de...service.impl.UserServiceImpl">
   <property name="userRepository" ref="userRepository"/>
</bean>

<bean id="userRepository" class="de...repository.impl.JdbcUserRepository">
   <constructor-arg ref="dataSource"/>
</bean>

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

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

<tx:advice id="txAdvice">
   <tx:attributes>
      <tx:method name="*" propagation="REQUIRED"/>
   </tx:attributes>
</tx:advice>

<aop:config>
   <aop:advisor advice-ref="txAdvice" pointcut="execution(* de.codecentric..*.*(..))" />
</aop:config>
```

Listing 3

```
<context:component-scan base-package="de.codecentric"/>

<tx:annotation-driven/>

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

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

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

@Component, *@Service*, *@Repository* oder *@Controller* annotiert werden. Im Prinzip bewirken alle vier Annotationen dasselbe, die letzten drei bringen aber noch ein wenig Semantik mit, sodass ich sie im Beispiel verwende (Listing 4). Die *@Autowired*-Annotationen sorgen dann für die Verdrahtung der Komponenten. Sie funktionieren auf Methoden-, Konstruktor- und Attributebene und lassen per Default nach typgleichen Komponenten im ApplicationContext suchen und injizieren. Die *@Transactional*-Annotation funktioniert auf Klassen- und Methodenebene und sorgt dafür, dass alle Methoden der annotierten Klasse bzw. die annotierte Methode transaktional abgesichert werden.

Der erste offensichtliche Vorteil ist wohl, dass die *Konfigurationsdatei erheblich kürzer* geworden ist. *Typsicherheit* wird ebenfalls erreicht, außerdem verliert man viele der oben genannten Nachteile der XML-basierten Konfiguration, allerdings natürlich nur für den Teil, der nicht in der XML-Datei definiert wird.

Dafür ist die *Konfiguration der Anwendung nun über die ganze Codebasis verstreut*. Der ursprüngliche Ansatz der Dependency Injection, dumme Komponenten zu haben, die nicht wissen, mit wem sie zusammenarbeiten müssen, wird dadurch zumindest teilweise unterlaufen, denn jetzt ist es doch wieder die Komponente, die weiß, wer wie injiziert werden muss. Dazu kommt, dass *Autowiring nicht* so *intuitiv* ist wie explizite Konfiguration. Und die Injection by Type bringt neue Probleme mit sich, wenn es mehrere Komponenten eines Typs geben soll.

Java-basierte Konfiguration

Code sagt mehr als tausend Worte, deswegen möchte ich direkt mit der Konfiguration für den Datenzugriff beginnen (Listing 5). Die Annotation *@Configuration* markiert eine Klasse als Quelle von Spring Beans. Jede Methode, die mit *@Bean* annotiert ist, produziert eine Singleton Spring Bean vom Rückgabetyp der Methode mit dem Methodennamen als Bean-Namen. Wird eine dieser Methoden mehrere Male aufgerufen, so liefert sie immer genau das gleiche Objekt zurück, das wie in den anderen Konfigurationsarten mit einem Proxy versehen

Listing 4

```
@Transactional
@Service
public class UserServiceImpl implements UserService {
   private UserRepository userRepository;
   @Autowired
   public void setUserRepository(UserRepository userRepository) {
      this.userRepository = userRepository;
   }
   // business code here
}
```

wird, falls Aspekte bei Methodenaufrufen auf diesem Objekt tätig werden sollen.

Um die Platzhalter in der *DataSource*-Konfiguration zu ersetzen, greife ich dieses Mal nicht auf den Property Placeholder zurück, sondern nutze das in Spring 3.1 neu eingeführte Environment [1], [2], [3]. Mit der Annotation *@PropertySource* kann dem Environment eine Property-Quelle hinzugefügt werden, das Environment selbst lässt sich per *@Autowired*-Annotation automatisch injizieren. Die Datenbank-Properties frage ich dann an der entsprechenden Stelle ab. Diese Konfiguration für den Datenzugriff importiere ich per *@Import*-Annotation in meine Applikationskonfiguration (Listing 6).

Neu in Spring 3.1 ist die Annotation *@EnableTransactionManagement*, die ein Äquivalent zum XML-Tag *<tx:annotation-driven/>* darstellt. Es werden also in dieser Konfiguration die *@Transactional*-Annotationen im Code verwendet, um zu bestimmen, wo und wann Transaktionen geöffnet und geschlossen werden müssen. Wer Component Scanning und Autowiring schon ins Herz geschlossen hatte, muss darauf auch bei der Java-basierten Konfiguration nicht verzichten (Listing 7), denn dafür gibt es die ebenfalls in Spring 3.1 neu eingeführte Annotation *@ComponentScan*.

Welche Vorteile bietet nun die Java-basierte Konfiguration? In einem Satz: *Wir werden XML los, ohne irgendwelche Nachteile in Kauf nehmen zu müssen*. Java-Code ist selbstverständlich *typsicher* und wird direkt zur *Compile-Zeit* überprüft. Jede Java-IDE bietet eine Vielzahl von Hilfsmitteln, die wir in XML-Dateien nicht selbstverständlich zur Verfügung haben: *Code-Completion*, *Auflösen von Referenzen*, *Refactoring* und viele mehr. Es ist sehr einfach möglich, direkt vom Businesscode in die Konfiguration zu springen. Benennt man eine Klasse per Refactoring um, so wird sie auch automatisch in der Konfiguration umbenannt. Baut man eine verteilte Konfiguration wie in Listing 5 und 6 auf, so kann man mit IDE-Mitteln einfach zwischen diesen *navigieren*: Möchte ich in der *AppConfig* wissen, wie genau nun die *DataSource* in der *DataConfig* definiert ist, springe ich mit einem Klick in die Deklaration der Methode *dataSource()*. Und natürlich kann in der Konfiguration beliebiger *echter Java-Code ausgeführt* werden, ohne dass man über die Expression Language gehen muss. Nachteile gibt es keine.

Auch in der JavaConfig haben wir die Wahl, unsere eigenen Komponenten explizit zu konfigurieren oder per Component Scanning und Autowiring finden zu lassen (Listing 6 vs. Listing 7). Der Vorteil von Component Scanning ist, dass man eben keine explizite Konfiguration mehr braucht und die Konfigurationsklassen dadurch schmaler werden. Der Nachteil ist, dass es nicht ganz so intuitiv und die Konfiguration im Prinzip über die ganze Codebasis verstreut ist.

Weitere Namespace-Äquivalente

Neben den bereits angesprochenen Annotationen *@EnableTransactionManagement* und *@ComponentScan* bringt Spring 3.1 einige weitere Code-Äquivalente für Namespaces mit [4]:

- *@EnableAsnyc* für *<task:*>*
- *@EnableScheduling* für *<task:*>*

Listing 5

```
@PropertySource("classpath:database.properties")
@Configuration
public class DataConfig {

    @Autowired
    Environment env;

    @Bean
    public DataSource dataSource(){
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setUrl(env.getProperty("jdbc.url"));
        dataSource.setUsername(env.getProperty("jdbc.username"));
        dataSource.setPassword(env.getProperty("jdbc.password"));
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager(){
        return new DataSourceTransactionManager(dataSource());
    }

}
```

Listing 6

```
@EnableTransactionManagement
@Import(DataConfig.class)
@Configuration
public class AppConfig {

    @Autowired
    DataConfig dataConfig;

    @Bean
    public UserRepository userRepository(){
        return new JdbcUserRepository(dataConfig.dataSource());
    }

    @Bean
    public UserService userService(){
        UserServiceImpl userService = new UserServiceImpl();
        userService.setUserRepository(userRepository());
        return userService;
    }

}
```

- *@EnableLoadTimeWeaving* für *<context:load-time-weaver>*
- *@EnableAspectJAutoProxy* für *<aop:aspectj-auto-proxy>*
- *@EnableWebMvc* für *<mvc:annotation-driven>*

Insbesondere der letzte Punkt bietet einige Vorteile gegenüber der XML-Variante [5].

Erzeugung des JavaConfig-ApplicationContext

Was fehlt uns jetzt noch, um einen JavaConfig-ApplicationContext erfolgreich einzusetzen? Wir müssen wissen, wie er gebootet wird. In den meisten Fällen wird Spring in Webanwendungen verwendet und der *ContextLoaderListener* in der *web.xml* registriert, damit er beim Starten der Anwendung auch gleichzeitig den ApplicationContext mit hochfahren kann. Da dieser allerdings im Default-Fall einen *XmlWebApplicationContext* erzeugt, müssen wir über den Context-Parameter *contextClass* die Alternative *AnnotationConfigWebApplicationContext* angeben. Dann können wir unter den Parameter *contextConfigLocation* den vollqualifizierten Klassennamen unserer Konfigurationsklasse setzen (Listing 8).

Ein weiterer häufiger Anwendungsfall ist das Booten eines ApplicationContext durch Springs TestContext-Framework [6]. Auch hier wird über die Annotation *@ContextConfiguration* üblicherweise eine XML-Datei referenziert. Um die Konfiguration aus einer Klasse zu laden, muss noch ein alternativer *ContextLoader* angegeben werden (Listing 9).

JPA ohne XML

Ein häufig verwendetes Spring-Feature ist die Unterstützung für JPA. In Spring 3.1 wurde die *LocalContainerEntityManagerFactoryBean* um eine Property *packagesToScan* erweitert, sodass es jetzt möglich ist, die *persistence.xml* ganz wegzulassen [4].

Fazit

Wer mit Spring 3.1 noch eine Zeile XML schreibt, ist selbst Schuld oder hat einen guten Grund. Gute Gründe gibt es, so bieten einige Spring-Unterprojekte sehr hilfreiche und ausgefeilte XML-Namespaces an, für die es (noch) keinen ebenso bequemen Codeersatz gibt, z. B. Spring Security oder Spring Integration. Da aber SpringSource das Thema JavaConfig zurzeit im Fokus hat, kann es gut sein, dass es auch dort in nächster Zeit Lösungen gibt. Bis dahin können beide Varianten auch sehr einfach miteinander kombiniert werden. In Zukunft wird es mit dem *ServletContainerInitializer* aus der Servlet-3.0-Spezifikation auch möglich sein, eine Webanwendung ohne *web.xml* zu erstellen. In Kombination mit Springs JavaConfig ist es dann nicht mehr weit bis zu einer XML-freien Anwendung.

Listing 7

```
@EnableTransactionManagement
@ComponentScan("de.codecentric")
@Import(DataConfig.class)
@Configuration
public class AppConfig {
}
```

Listing 8

```
<context-param>
<param-name>contextClass</param-name>
<param-value>org...java.context.AnnotationConfigWebApplicationContext</param-value>
</context-param>
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>de.codecentric.config.AppConfig</param-value>
</context-param>
<listener>
<listener-class>org.sfw.web.context.ContextLoaderListener</listener-class>
</listener>
```

Listing 9

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(loader = AnnotationConfigContextLoader.class, classes =
                                                              AppConfig.class)
public class UserServiceTest {
  @Autowired
  private UserService userService;
  // tests here
}
```

Tobias Flohre arbeitet als Senior-Softwareentwickler bei der codecentric AG. Seine Schwerpunkte sind Java-Enterprise-Anwendungen und Architekturen mit JEE/Spring. Außerdem interessiert er sich für die aktuellen Entwicklungen rund um die Cloud: PaaS, NoSQL und Co. Kontakt: tobias.flohre@codecentric.de.

Links & Literatur

[1] Flohre, Tobias und Czollmann: „Pascal, Spring 3.1 – Was gibt's Neues", Java Magazin 7.2011

[2] http://blog.springsource.com/2011/02/11/spring-framework-3-1-m1-released/

[3] http://blog.springsource.com/2011/02/15/spring-3-1-m1-unified-property-management/

[4] http://blog.springsource.com/2011/06/10/spring-3-1-m2-configuration-enhancements/

[5] http://blog.springsource.com/2011/06/13/spring-3-1-m2-spring-mvc-enhancements-2/

[6] http://blog.springsource.com/2011/06/21/spring-3-1-m2-testing-with-configuration-classes-and-profiles/

Notizen

codecentric AG
Kölner Landstraße 11
40591 Düsseldorf

Tel: +49 (0) 211.9941410
Fax: +49 (0) 211.9941444

E-Mail: info@codecentric.de
www.codecentric.de
blog.codecentric.de