Einsatz von Flex mit Maven in Java-Enterprise-Projekten

Flex Mavenized

Mit dem Flex-Mojos-Plug-in besteht die Möglichkeit der einfachen Integration von Flex in Maven-Projekte. Wie das funktioniert, werden wir anhand einer Beispielanwendung unter Einsatz von Flex 4 mit Spring 3 und Maven 3 demonstrieren.

von Michael Schütz und Marek Iwaszkiewicz

Wie den meisten sicherlich nicht entgangen ist, haben *Rich Internet Applications (RIA)* stark an Bedeutung gewonnen. Allen voran gilt das für eines der am meisten verbreiteten RIA-Frameworks: *Adobe Flex*. Im Zuge dieser Bedeutungszunahme wird im Projektgeschäft bei Kunden eben dieses Framework immer häufiger im Kontext professioneller Java-Enterprise-Anwendungen eingesetzt. Im Backend kann dabei sowohl Spring als auch EJB zum Einsatz kommen. Den Reiz dieser Kombination macht für unsere Kunden die saubere Trennung zwischen Front- und Backend sowie die visuellen und funktionalen Vorteile von RIAs aus. Leider mussten wir mehrmals feststellen, dass die Wahl von Flex als Oberflächentechnologie ausschlaggebend für den Einsatz von Apache Ant als Build-System war, obwohl diese Einschränkung gar nicht existiert. Wir sagen leider, da mit Maven eine modernere und sauberere Alternative eines Build-Management-Systems existiert, die mit dem Flex-Mojos-Plug-in [1] eine direkte Integration von Flex in Maven bereit stellt. Im Zuge dieses Artikels möchten wir vorstellen, wie eine auf Java-Technologien basierende professionelle Enterprise-Anwendung mit Flex im Frontend, Spring im Backend und Maven als Build-System umgesetzt werden kann. Da seit dem Release von Flex 4 mittlerweile ein Dreivierteljahr vergangen ist und diese Version in neu aufgesetzten Projekten überwiegend zum Einsatz kommt, möchten wir auch in unserer Demoanwendung auf Flex 4 setzen. Diese Entscheidung impliziert aber auch eine Konsequenz. So wird Flex 4 mit dem Flex-Mojos-Plug-in nur in Kombination mit Maven 3 unterstützt. Da Maven 3 zur Vorgängerversion abwärtskompatibel ist, ergeben sich hierdurch keine Folgen für gegebenenfalls andere eingebundene Projektteile, die zuvor mit Maven 2 implementiert wurden.

Die Pizza-Anwendung

Bevor wir aber in die technische Tiefe dieser Thematik einsteigen, soll die im Rahmen des Artikels vorgestellte Anwendung kurz beschrieben werden. Wohl ziemlich jeder gönnt sich in regelmäßigen Abständen mal eine gute Pizza – so auch wir. Warum aber die Pizza nicht direkt auch online bestellen? Auch wenn wir nicht die ersten

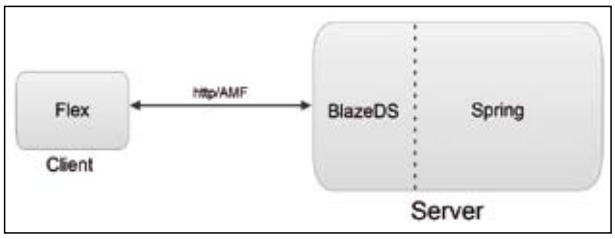

Abb. 1: Architektur/Technologie-Stack

mit dieser Idee sind, wollen wir in der Demoanwendung dennoch ein kleines Portal für das Bestellen von Pizzen umsetzen. Damit ist die Anwendung an sich schon komplett beschrieben. Wir halten die Anwendung bewusst sehr einfach, um uns auf das Wesentliche konzentrieren zu können: die Integration von Flex in auf Maven basierende Java-Projekte. Der Technologie-Stack der Anwendung besteht aus Flex 4, Maven 3, Flex Mojos 4, BlazeDS [2] sowie Spring 3. **Abbildung 1** zeigt ein einfaches Architekturbild der beteiligten Komponenten und Technologien.

Wie erwähnt, fällt die Wahl der eingesetzten Backend-Technologie in unserer Demoanwendung auf Spring 3. Das hat jedoch keine tiefergehenden Gründe. Man könnte auch EJB als Backend-Technologie wählen. Ein Vorteil von Spring ist aber, dass es für die Anbindung von Flex-Clients ans Backend Erweiterungen bereitstellt, mit denen die Integration schnell und komfortabel realisiert werden kann. Ein weiterer wohl genereller Vorteil von Spring ist, dass die für die Demoanwendung benötigten Projekte etwas schlanker ausfallen. BlazeDS ist eine von Adobe bereitgestellte Messaging-Lösung für die Anbindung von Flex-Clients an Java Backends und ist in den Spring-Erweiterungen enthalten.

Projektstruktur

Wie in der Einleitung erwähnt, setzen wir Maven 3 als Build-System ein. Die Demoanwendung besteht aus insgesamt drei Maven-Projekten: einem Parent-Projekt und zwei weiteren von diesem verwalteten Maven-Projekten: *jm-flex* und *jm-war*. **Abbildung 2** zeigt einen Screenshot mit den genannten Projekten.

Bei den vom Parent verwalteten Artefakten handelt es sich um ein Flex-Projekt (*jm-flex*) und um eine Java-Webanwendung (*jm-war*). Im Parent erfolgt wie üblich die Angabe der Kindmodule, die Angabe einiger globaler Abhängigkeiten (Dependencies) sowie die Definition von Versions-Properties und des Dependency- und Plugin-Managements.

Das Flex-Projekt

Das Flex-Projekt beinhaltet allen voran die Flex Sources, die die Clientanwendung implementieren. Zum

Listing 1

```
<packaging>swf</packaging>
<dependencies>
    <dependency>
      <groupId>com.adobe.flex.framework</groupId>
      <artifactId>flex-framework</artifactId>
      <version>${flex.version}</version>
      <type>pom</type>
    </dependency>
</dependencies>
```

Listing 2

```
<build>
    <plugins>
      <plugin>
      <groupId>org.sonatype.flexmojos</groupId>
      <artifactId>flexmojos-maven-plugin</artifactId>
        <configuration>
          <services>
            ${basedir}/src/main/config/services-config-local.xml
          </services>
          <contextRoot>/${web.context.root}</contextRoot>
          <locales>
            <locale>en_US</locale>
          </locales>
        </configuration>
      </plugin>
    </plugins>
</build>
```

Listing 3

```
<packaging>war</packaging>
<build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-war-plugin</artifactId>
          <version>2.0.2</version>
        </plugin>
      </plugins>
    </pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.sonatype.flexmojos</groupId>
        <artifactId>flexmojos-maven-plugin</artifactId>
        <version>${flexmojos.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>copy-flex-resources</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
</build>
```

Bauen des Projekts mit Maven wird von diesem Projekt das Flex-Mojos-Maven-Plug-in eingesetzt. Die benötigten Dependencies fallen sehr kurz aus – genau genommen benötigen wir hier zwei Artefakte: das Flex-Framework als Dependency und das bereits angekündigte Maven-Plug-in Flex Mojos zum Kompilieren. In Listing 1 ist die einzige einzutragende Dependency auf das Flex-Framework enthalten. Die von uns genutzte Version ist 4.0-beta-3. In Listing 1 ist ebenfalls die Angabe des Packaging des Projekts angegeben. Durch den Packaging-Typ *swf* wird festgelegt, dass es sich beim resultierenden Artefakt um eine Flash-Datei handelt.

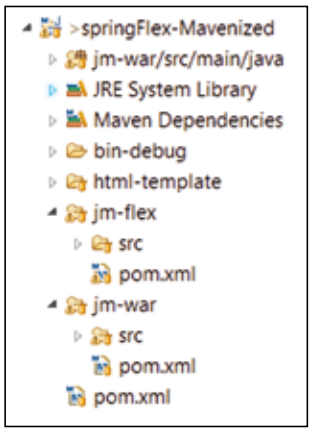

Abb. 2: Projektstruktur der Pizza-Anwendung

Einen weiteren Ausschnitt aus der Konfigurationsdatei *(pom.xml)* des Flex-Projekts sehen Sie hier:

```
<build>
    <finalName>Main</finalName>
    <sourceDirectory>src/main/flex</sourceDirectory>
</build>
```

In diesem wird mit *finalName* der Name des Artefakts, also der Name der erzeugten SWF-Datei festgelegt. Im *sourceDirectory*-Element wird dem gleich vorgestellten Flex-Mojos-Plug-in der Pfad zu den Flex Sources angegeben. Wie bei Maven üblich, befinden sich auch hier die Quellen im *src/main*-Verzeichnis.

Abschließend zum Flex-Projekt sieht man in Listing 2 den letzten Teil der dazugehörigen *pom.xml*, in dem die Einbindung des Flex-Mojos-Plug-ins dargestellt ist. Hier müssen die sonst bei Flex auch üblichen Compiler-Einstellungen vorgenommen werden. Im *services*-Element erfolgt die Angabe des Pfads zur Flex-Konfigurationsdatei *services-config.xml*. In dieser Datei wird alles rund um die Kommunikation des Flex-Clients mit dem Backend vorgenommen. Dazu zählen zum Beispiel die Angaben des Protokolls, der Kommunikationskanäle (Channels) sowie der Endpoints. Channels bilden eine der zentralen Konfigurationseinheiten von BlazeDS. Sie definieren das für die Kommunikation genutzte Format (AMF [3], SOAP [4]), das Transportprotokoll (http) und die URL-Endpoints. Das Element *contextRoot* legt den Namen der Webanwendung fest, mit der der Client schließlich kommunizieren soll. Wird mit Internationalisierung gearbeitet, so können die unterstützten Sprachen im *locales*-Element angegeben werden.

Das Webprojekt

Im Webprojekt erwartet uns neben den Java-Quellen auch die Integration der im Flex-Projekt entwickelten Clientanwendung. Bei dem Projekt selbst handelt es sich um ein einfaches Standardwebprojekt, das in einem Webarchiv (WAR) gebündelt wird. In Maven wird für diesen Projekttyp als Packaging *war* in Kombination mit dem Maven-War-Plug-in [5] eingesetzt, das das Zusammenbauen der von WAR-Anwendungen geforderten Strukturen übernimmt. Im Webprojekt wird zudem erneut vom Flex-Mojos-Plug-in Gebrauch gemacht – dieses Mal jedoch nicht, um Code zu kompilieren, sondern um die in den Maven Dependencies eingebundenen Flex-Artefakte in die Webanwendung zu kopieren und somit in diese zu integrieren. Listing 3 enthält den dazugehörigen Konfigurationstext.

Das wären erst einmal die notwendigen Schritte, um mit Maven ein Webprojekt zu erstellen. Neben einer direkten Maven-Abhängigkeit zu

unseren Flex-Projekten beziehen sich die üblichen Abhängigkeiten allesamt auf Spring-Komponenten. Spring selbst ist modular aufgebaut, wodurch die benötigten Ressourcen ganz gezielt hinzugefügt werden können. So enthält das Artefakt *spring-flex* den erforderlichen Verbindungscode, um eine Spring-Komponente direkt für den Flex-Zugriff zu konfigurieren. Durch dieses Modul können Spring Beans fast ohne die Notwendigkeit weiterer Konfigurationen auf Backend-Seite nahtlos in die Flex-Kommunikation eingebunden werden.

Das Weiterleiten der vom Flex-Client eingehenden Anfragen an die entsprechende Bean übernimmt Spring. An dieser Stelle bietet Spring wie üblich zwei Integrationsmöglichkeiten: XML und Annotationen. Sollen Annotationen verwendet werden, so muss das in einer Spring-Konfigurationsdatei angegeben werden. Listing 4 enthält den gesamten Code der Demoanwendung, in der wir uns für Annotationen entschieden und hierfür mit dem *component-scan*-Element den Component Scan aktiviert haben, wodurch alle im angegebenen Java Package enthaltenen Spring-Komponenten erkannt werden. Dadurch fällt die dazugehörige Konfigurationsdatei entsprechend kurz aus.

In Listing 6 sieht man die gängige Deklaration einer Spring Bean. Durch die Serviceannotation wird die dazugehörige Klasse als Bean deklariert. Damit die Bean aber auch noch für Flex-Clientaufrufe zugänglich gemacht wird, muss zusätzlich wie im Listing eine *RemotingDestination*-Annotation auf Klassenebene hinzugefügt werden. Aus dem Namen der Annotation geht bereits hervor, dass die damit annotierten Klassen als „entfernte Ziele" von Flex-Clients fungieren und von diesen aufgerufen werden können. Es werden hierbei alle *public*-Methoden nach außen zum Aufruf zur Verfügung gestellt. In der *RemotingDestination*-Annotation muss zudem auch noch mindestens ein Channel angegeben werden. Dadurch erfolgt auch die Zuordnung einer Destination zu einem Channel, die zur Laufzeit von einem von Spring gesteuerten Message Broker für das Dispatching der Aufrufe herangezogen wird. Die hier angegebenen Channels müssen zweierlei Dinge erfüllen: erstens muss jeder hier angegebene Channel ebenfalls in der *services-config.xml* des Flex-

Listing 4

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

    <context:component-scan
       base-package="incubator.spring_flex.services" />

</beans>
```

Listing 5

```
import org.springframework.flex.remoting.RemotingDestination;
import org.springframework.stereotype.Service;

@Service("orderService")
@RemotingDestination(channels = { "my-amf" })
public class OrderService {
   ...
}
```

Listing 6

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:flex="http://www.springframework.org/schema/flex">

  <flex:message-broker>
    <flex:message-service default-channels="my-amf" />
  </flex:message-broker>
</beans>
```

Listing 7

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>

  <servlet>
    <servlet-name>flex</servlet-name>
  <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>flex</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Listing 8

```
<dependency>
  <groupId>com.adobe.flexunit</groupId>
  <artifactId>flexunit</artifactId>
  <version>4.0-rc-1</version>
  <type>swc</type>
  <scope>test</scope>
</dependency>

...

<build>
  <finalName>Main</finalName>
  <sourceDirectory>src/main/flex</sourceDirectory>
  <testSourceDirectory>src/test/flex</testSourceDirectory>
</build>
```

Projekts angegeben werden. Zweitens muss wie in Listing 6 der bereits erwähnte Message Broker mit diesem Channel als Spring Bean deklariert werden.

Abschließend darf zur Beschreibung des Webprojekts der entsprechende XML-Descriptor *web.xml* nicht fehlen. Auch hier decken sich die Konfigurationsangaben mit denen aus der *services-config.xml*-Datei des Flex-Projekts. So wird das *DispatcherServlet* an das URL-Pattern */messagebroker/** gebunden – ein Pattern, das in der Regel von allen URLs der defnierten Endpoints erfüllt wird. Dadurch wird das *DispatcherServlet* zum zentralen Verteiler der Clientanfragen. Das erneut sehr kurze Listing 8 enthält alle notwendigen Einträge, die im Kontext der Spring-Flex-Integration im Web-Descriptor anfallen.

Geschafft – bis jetzt haben wir vorgestellt, welche Konfigurationsschritte vorzunehmen sind, um eine Flex-Anwendung mit Spring 3 im Backend zu erstellen und diese mit Maven 3 zu bauen. Wie wir finden, stellt das eine spannende und vor allem robuste Technologiekombination dar.

Blick unter die Haube

Wir haben bisher eher weniger Bezug auf das Flex-Mojos-Plug-in genommen, das weitaus mehr kann, als lediglich eine Flex-Anwendung zu kompilieren und in eine Webanwendung einzubinden. So bietet das Flex-Mojos-Plug-in die Möglichkeit, eine Flex-Anwendung in verschiedenen Modulen zu organisieren und die Versionen und Konfigurationen mit Mechanismen aus Maven zu verwalten. Weiterhin ist der Einsatz des Plug-ins nicht auf Flex-Anwendungen beschränkt, sondern bietet auch die Möglichkeit, mit Flash- und Flex-Bibliotheken sowie mit AIR-Anwendungen zu arbeiten. In einem häufigen Einsatzszenario wird das ActionScript-Modell [6] für die mit dem Backend ausgetauschten Data Transfer Objects (DTOs) in einem separaten Maven-Projekt abgelegt und anschließend als Bibliothek in die Flex-Anwendung eingebunden. Das Plug-in bietet darüber hinaus auch die Möglichkeit, das ActionScript-Modell aus den Java-Klassen während des Build-Prozesses generieren zu lassen. Hierdurch müssen Entwickler nicht mehr beide Modelle pflegen und darauf achten, dass diese stets synchron sind.

Testintegration

Ein weiterer Aspekt guter Software ist die Möglichkeit, automatisierte Unit Tests ausführen zu können. In Maven wird dieser Aspekt generell großgeschrieben, sodass das automatisierte Ausführen von Tests einen festen Bestandteil des Build-Prozesses darstellt. Das gilt auch im Kontext von Flex-Anwendungen. Es existiert mit FlexUnit [7] ein an JUnit4 angelehntes Testframework für Flex, das mit wenigen Handgriffen in ein Flex-Projekt integriert werden kann. Die hierzu notwendigen Erweiterungen sind in Listing 8 dargestellt. In diesem wird zunächst eine Abhängigkeit auf FlexUnit eingebunden und danach das Verzeichnis ange-

Abb. 3: Screenshot der gebauten Anwendung

geben, in dem die Testfälle abgelegt sind. Das reicht bereits aus, damit beim nächsten Build die sich in diesem Verzeichnis befindenden Testfälle automatisiert ausgeführt werden.

Los geht's – das Demoprojekt selbst starten

Genug der Theorie. Jetzt möchten wir zeigen, wie leicht sich die in diesem Artikel beschriebene Demoanwendung selbst bauen und ausprobieren lässt. **Abbildung 3** zeigt das Ergebnis der installierten Applikation. Für die Installation der Demoanwendung müssen folgende Voraussetzungen erfüllt sein:

- Java Version >= 1.5
- Maven Version = 3.0.1

Zum Ausführen der Anwendung sind schließlich mehrere Schritte notwendig. Zuerst muss das Projekt heruntergeladen werden. Das Projekt befindet sich im Github unter dem URL https://github.com/michaelschuetz/springFlex-javamagazin:

- Git Checkout [8] oder Herunterladen des gepackten Archivs
- cd spring-flex
- mvn install

Nachdem alle notwendigen Maven-Artefakte heruntergeladen wurden, werden die Flex Sources kompiliert, mit der Java-Webanwendung verbunden und eine deploy-fähige Webanwendung erzeugt. Unsere Pizza-Anwendung erzeugt ein WAR-Archiv, das in gängige Servlet-Container installiert werden kann. Die Entscheidung, bei der Architektur auf Spring und nicht auf EJB zu setzen, ermöglicht das Deployment der Anwendung in einfache Servlet-Container, wie Tomcat und Jetty, sowie in Java-EE-konforme Applikationsserver.

Der schlanke Ansatz der Architektur wirkt sich somit auch direkt beim Deployment aus, da die Turnaround-Zeiten entsprechend gering ausfallen. Während der Entwicklung der Beispielanwendung haben wir gegen den Tomcat 6 und 7 getestet.

Fazit

Flex stellt schon länger eine ernsthafte und solide Lösung zur Erstellung professioneller Webanwendungen dar, allerdings mangelte es lange an der Integration in moderne, auf Maven basierende Java-Webanwendungen. In Kombination mit Spring im Backend und Maven als Build-System steht ein sehr leistungsfähiges und zukunftsträchtiges Dreigestirn zur Entwicklung von RIAs der nächsten Generation bereit. Eben dieses Zusammenspiel wurde hier aus technischer Architekturperspektive beleuchtet.

Trotz der positiven Resultate muss auch erwähnt werden, dass das Konfigurieren des Flex-Mojo-Plug-ins an einige Stellen etwas mühselig ist. Dies gilt speziell beim Einbinden weiterer Features wie der Codegenerierung. So hat es mehrerer Versuche bedurft, bis wir ein lauffähiges Setup hinbekommen haben. Grund hierfür ist die leider noch etwas unvollständige und veraltete Dokumentation des Plug-ins. Mit ein wenig Recherche und etwas Geduld ließen sich aber schließlich die gewünschten Funktionalitäten zum Laufen bringen. Das Wesentliche, nämlich das Kompilieren von Flex-Projekten und das Verwalten verschiedener Flex-Module mit Maven funktioniert jedoch einwandfrei – so wie im Rahmen des Artikels vorgestellt.

Marek Iwaszkiewicz arbeitet als Software Engineer beim IT-Dienstleistungs- und Beratungsunternehmen adesso AG. Seine Schwerpunkte dort sind EJB sowie Java-Enterprise-Architekturen mit Spring. E-Mail: marek.iwaszkiewicz@adesso.de.

Michael Schütz arbeitet als Software Engineer bei der akquinet AG in Berlin und beschäftigt sich seit Jahren mit Java-Enterprise-Architekturen und -Technologien. Er ist Gründer und Moderator der Seam-Plattform auf Xing und wirkt regelmäßig an JBoss-Open-Source-Projekten mit. Michaels berufliche Interessen liegen derzeit in den Bereichen Seam/CDI, Java EE Testing, Java EE Cloud Hosting und Git. Kontakt: Michael.Schuetz@akquinet.de.

Links & Literatur

[1] http://flexmojos.sonatype.org

[2] http://opensource.adobe.com/wiki/display/blazeds/BlazeDS

[3] http://opensource.adobe.com/wiki/download/attachments/1114283/amf3_spec_05_05_08.pdf

[4] http://www.w3.org/TR/soap

[5] http://maven.apache.org/plugins/maven-war-plugin

[6] http://www.actionscript.org

[7] http://flexunit.org

[8] http://git-scm.com