Frühling für Portlets

Spring Portlet MVC

Reza Nazarian

Durch die zunehmende Verbreitung von Portal-Servern beschäftigen sich Entwickler auch zunehmend mit der Entwicklung von auf Portlets basierenden Applikationen. Der Artikel stellt das Framework Portlet MVC aus der weiten Spring-Welt vor und zeigt, wie sich Portlets damit realisieren lassen.

Wie üblich in der Welt der Webapplikationen teilt sich auch die der Portlets in eine auf Requests basierende und eine auf Komponenten basierende Welt auf.

Entscheidet man sich für die letztere, so trifft man zurzeit vor allem auf JavaServer Faces (JSF), deren Einsatz in einer Portal-Umgebung eine Portlet-Bridge ermöglicht. Möchte man seine Portlets lieber basierend auf Requests umsetzen, so implementiert man direkt gegen die Spezifikation JSR 168 (Portlet 1.0, [JSR168]) bzw. JSR 286 (Portlet 2.0, [JSR286]) oder man wählt ein Framework wie Spring Portlet MVC [SpringPortlet].

Wie der Name vermuten lässt, basiert Spring Portlet Web MVC auf dem Framework Spring Web MVC [SpringWebMVC]. Somit setzt auch Portlet MVC ein konsequentes Model-View-Controller-Muster um und ermöglicht die Verwendung aller Features aus Spring und Web MVC sowie den Einsatz der allermeisten Spring-Frameworks wie Security und Web Flow für die Portlet-Entwicklung. Das ist schon mal eine ganze Menge, aber fangen wir erst mal mit einem Blick auf die Architektur an.

Models, Aussichten und Kontrolleure

Wer sich bereits mit Web MVC auskennt, dem wird dieser Abschnitt bekannt vorkommen. Der Grundaufbau ist in Portlet MVC nahezu identisch: Eine Portlet-Applikation in Spring besteht grundsätzlich aus einem zentralen Dispatcher-Portlet, einem Satz von Controllern und einem Satz von Views.

Alle Requests auf die Portlet-Applikation nimmt ein Dispatcher-Portlet entgegen und leitet an einen passenden Controller weiter (s. Abb. 1). Die Auswahl des Controllers basiert auf dem aktuellen Modus des Portlets (View, Edit, Help), der Portlet-Phase (Render, Action, Resource, Event) und/oder einem Parameter im Request (z. B. `action=addUser`). Ist der passende Controller ausgewählt, so liefert er, basierend auf seiner Logik, ein Paar aus Model- und View-Name an das Dispatcher-Portlet zurück. Dieses ermittelt anhand des Namens das Render-Template (z. B. eine JSP) und liefert Model- und Template-Pfad (`ModelAndView`) als Response zurück. Der Portlet-Container übernimmt mit diesen Informationen nun die Erzeugung des Portlet-Markups.

Das beschriebene Handling von Requests zur Erzeugung von Responses ist übrigens für alle Portlet-Phasen, also „Render“, „Action“, „Event“ und „Resource“, dasselbe. Allerdings liefern Controller in der Action- und Event-Phase keine `ModelAndView`-Instanz zurück.

Die oben beschriebene Auswahl des passenden Controllers für einen eingehenden Request übernimmt das sogenannte Handler-Mapping. Die Bestimmung des Render-Templates ermittelt eine in Kette geschaltete Liste von sogenannten View-Resolvern.

Abb. 1: Ablauf eines Requests mit Spring Portlet MVC

Kommen wir wieder zurück zu den Controllern. Für jede der zu unterstützenden Portlet-Phasen implementiert ein Controller ein entsprechendes Interface (`Controller` bzw. `AbstractController` für Render- und Action-Phase, `EventAwareController` für die Event-Phase und `ResourceAwareController` für die Resource-Phase). Des Weiteren können alle aus Spring Web MVC bekannten Controller wiederverwendet werden. Beispielsweise solche, mit denen man Formulare und deren Validierung bzw. Verarbeitung an einen Controller und eine Backing Bean bindet.

Schritt für Schritt zum MVC-Portlet

Im Folgenden wird ein einfaches Spring MVC Portlet (Spring 3.0.5, Spring-Portlet 2.0.8) bestehend aus einem Controller mit einer Bean und drei Views in ein Liferay-Portal (Community Edition, Version 6.0.6) integriert. Das Portlet bietet in der Render-Phase ein einfaches Formular an, um die für Fotografen spannende Blaue Stunde [BlaueStunde] für einige Städte zu erfragen. Zu-

sätzlich wird in der Resource-Phase ein Ajax-Request verarbeitet.

Zuerst legen wir in der Datei portlet.xml unser Dispatcher-Portlet an, das die zentrale Komponente unseres Portlets ist:

```
<portlet>
  <portlet-name>SpringPortletExample</portlet-name>
  <portlet-class>org.springframework.web.portlet.DispatcherPortlet
  </portlet-class>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>view</portlet-mode>
  </supports>
  <portlet-info>
    <title>Spring Portlet MVC - Beispiel</title>
  </portlet-info>
</portlet>
```

Mit diesem Dispatcher-Portlet wirkt das folgende **`ViewRendererServlet`** zusammen, das die nahtlose Anbindung an Spring Web MVC vornimmt, um all die dort verfügbaren Komponenten zu verwenden. Das **`ViewRendererServlet`** wird von allen Portlets applikationsweit verwendet:

```
<servlet>
  <servlet-name>ViewRendererServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.ViewRendererServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>ViewRendererServlet</servlet-name>
  <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

Es folgt unser View-Resolver in der Datei applicationContext.xml, der das Mapping von View-Namen auf JSPs vornimmt:

```
<bean id="viewResolver" class=
  "org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="cache" value="true" />
  <property name="viewClass"
    value="org.springframework.web.servlet.view.JstlView" />
  <property name="prefix" value="/WEB-INF/jsp/" />
  <property name="suffix" value=".jsp" />
</bean>
```

Soweit so gut. Nehmen wir uns als Nächstes den Controller vor. Dieser wird durch Annotationen auf einen eindeutigen Controller-Namen und den Portlet-Modus „View" konfiguriert. Für jede Portlet-Phase, die wir durch den Controller bearbeiten wollen, implementiert man die entsprechende Handle-Methode. In diesem Fall werden die Render- und die Resource-Phase unterstützt:

```
@RequestMapping(value = "VIEW")
@Controller(value="blaueStundeController")
public class BlaueStundeController extends AbstractController
    implements ResourceAwareController {

  public ModelAndView handleRenderRequest(RenderRequest renderRequest,
      RenderResponse renderResponse) throws Exception {
    if (renderRequest.getParameter("stadt") != null) {
      ModelAndView modelAndView = new ModelAndView();
      BlaueStundeBean bean =
        new BlaueStundeBean(renderRequest.getParameter("stadt"));
      modelAndView.addObject("cityBean", bean);
      modelAndView.setView("blaueStundeResult");
      return modelAndView;
    } else {
      return new ModelAndView("blaueStundeForm");
    }
  }

  @Override
  public ModelAndView handleResourceRequest(
      ResourceRequest resourceRequest,
      ResourceResponse resourceResponse) throws Exception {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("wetterDaten", "<p>Sonnig, 21 Grad.</p>");
    modelAndView.setView("blaueStundeAjax");
    return modelAndView;
  }
}
```

Wird das Portlet im Modus „View" aufgerufen, so wird – bevor der Benutzer etwas zu sehen bekommt – in der Render-Phase die Methode **`handleRenderRequest()`** aufgerufen. Da sich noch keine Parameter im Request befinden, liefert die Methode eine **`ModelAndView`**-Instanz zurück, in der sich lediglich ein View-Name (**`blaueStundeForm`**) befindet.

Unser View-Resolver löst den Namen **`blaueStundeForm`** auf die JSP „/WEB-INF/jsp/blaueStundeForm.jsp" auf. Das Formular erfragt nun vom Benutzer die Stadt und zeigt als Action-Ziel mittels einer Render-URL wieder auf unser Portlet. Der Request wird nach dem Abschicken des Formulars somit wieder von der Methode **`handleRenderRequest()`** in unserem Controller bearbeitet:

```
<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
<%@taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet"%>
<portlet:defineObjects />

<h2>Finde die Blaue Stunde für meine Stadt</h2>
<form action="<portlet:renderURL/>" method="POST">
  <aui:input name="stadt" label="Stadt"/>
  <br />
  <aui:button name="submitButton" type="submit"
            value="Abschicken" last="true"  />
</form>
```

Da der Parameter **„stadt"** nun mit dem über das Formular eingegebenen Wert gefüllt ist, erzeugt die Methode **`handleRenderRequest()`** eine Bean mit den Daten für die übergebene Stadt und liefert diese zusammen mit dem Namen der View (**`blaueStundeResult`**), die für die Darstellung der Ergebnisse zuständig ist, als **`ModelAndView`**-Instanz wieder zurück. Der Quellcode der Bean ist hier nicht weiter aufgeführt.

Die JSP blaueStundeResult.jsp wird über das Mapping von **`blaueStundeResult`** durch den View-Resolver gefunden und rendert die Daten aus der Bean. Durch das Klicken des Benutzers auf einen angebotenen Link für die Abfrage des aktuellen Wetters wird ein Ajax-Request auf den Controller ausgeführt. Der Ajax-Request findet in der Resource-Phase statt und hat daher die Methode **`handleResourceRequest()`** als Ziel:

```
<%@taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
<%@taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet"%>
<portlet:defineObjects />

<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.min.js">
</script>
<script type="text/javascript">
$(document).ready(function(){
  $("a.weatherLink").click(function(event){
    $.ajax({
      url: "<portlet:resourceURL/>",
      cache: false,
      success: function(html){
        $("#weatherDisplay").append(html);
      }
    });
  });
```

```
});
</script>
<h2>Die blaue Stunde für ${cityBean.stadt}</h2>
Morgens: ${cityBean.blaueStundeMorgens} <br />
Abends:  ${cityBean.blaueStundeAbends} <br />
<br />
<a class="weatherLink">Das Wetter für ${cityBean.stadt} anzeigen.</a>
<div id="weatherDisplay">
</div>
```

Der Aufruf von **handleResourceRequest()** über einen Ajax-Request erzeugt eine **ModelAndView**-Instanz, die den Namen der View zur Anzeige des Wetters für die ausgewählte Stadt und die eigentlichen Wetterdaten (z. B.„Sonnig, 21 Grad.“) in einem String enthält.

Die JSP blaueStundeAjax.jsp ermittelt wie bei den anderen JSPs der View-Resolver. Diese JSP gibt lediglich den String mit den Wetterdaten aus, damit dieser in der eigentlichen Ergebnisseite an das noch leere **div**-Element angehängt werden kann:

```
<p>${wetterDaten}</p>
```

Fertig. Fast. Für die Verwendung in Liferay benötigen wir noch einen Eintrag in die Dateien liferay-portlet.xml:

```
<liferay-portlet-app>
  <portlet>
    <portlet-name>SpringPortletExample</portlet-name>
    <instanceable>true</instanceable>
  </portlet>
</liferay-portlet-app>
```

und liferay-display.xml

```
<display>
  <category name="javaspektrum">
    <portlet id="SpringPortletExample"></portlet>
  </category>
</display>
```

Das Portlet lässt sich nun im Liferay-Portal verwenden, wie Abbildung 2 zeigt. Nach dem Ausfüllen des Formulars, dem Abschicken und der Ausführung des Ajax-Request, stellt das Portlet das Ergebnis dar (s. Abb. 3).

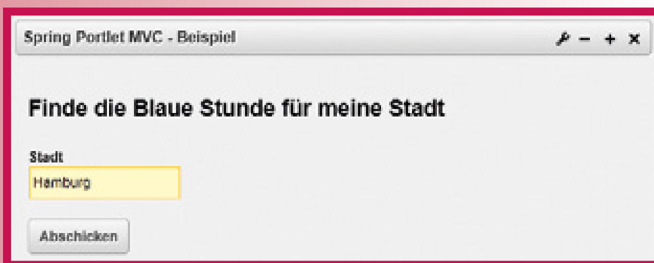

Abb. 2: Eingabemaske im Portlet

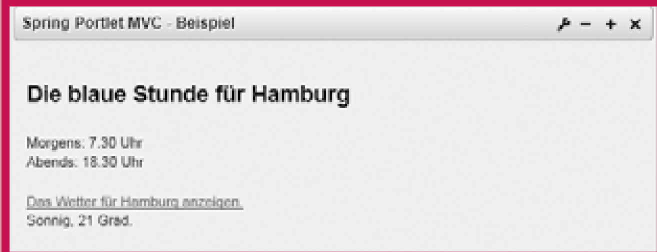

Abb. 3: Ausgabe im Portlet

Fazit

Spring Portlet MVC eignet sich sehr gut für den Einsatz in einer Spring-Umgebung. Sollen beispielsweise bestehende Applikationen basierend auf Spring Web MVC als Portlet über einen Portal-Server bereitgestellt werden, so lässt sich dies mit Portlet MVC rasch bewerkstelligen.

Grundsätzlich sollte man sich Portlet MVC auch anschauen, bevor man Portlets basierend auf den nackten Spezifikationen umsetzt. Die Kernfeatures von Spring und die von Web MVC, wie Dependency Injection, Formularverarbeitung, Validierung usw., nehmen einem Portlet-Entwickler viele Alltagsprobleme ab und verhindern, dass das Rad immer wieder aufs Neue entwickelt wird. Bringt man Spring-Frameworks à la Web Flow ins Spiel, kommt man um Portlet MVC nur schwer herum.

Obwohl die Dokumentation zu Portlet MVC von recht guter Qualität ist, so ist sie in Teilen leider nicht vollständig. Dies betrifft insbesondere die Features, die Portlet 2.0 betreffen (z. B. Event- und Resource-Handling). Diese werden nämlich noch überhaupt nicht beschrieben.

Links

[BlaueStunde] Wikipedia: Blaue Stunde,
http://de.wikipedia.org/wiki/Blaue_Stunde

[JSR168] Java Specification Request 168,
Portlet Specification (1.0),
http://www.jcp.org/en/jsr/detail?id=168

[JSR286] Java Specification Request 286,
Portlet Specification 2.0,
http://www.jcp.org/en/jsr/summary?id=286

[SpringPortlet] Portlet MVC Framework,
http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/portlet.html

[SpringWebMVC] Web MVC Framework,
http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/mvc.html

Reza Nazarian ist Senior Consultant bei der Unternehmensberatung Acando in Hamburg. Seine fachlichen Schwerpunkte liegen in den Bereichen Portal-Technologien und Content-Management-Systeme. Reza Nazarian ist seit 2002 in der IT-Branche tätig und hat Erfahrungen als Berater, Entwickler und Technischer Projektmanager gesammelt.
E-Mail: reza.nazarian@acando.de