KaffeeKlatsch
Das Magazin rund um Software-Entwicklung
ISSN 1865-682X
09/2010

Ein Teil des großen Ganzen

Java Portlets

von Werner Eberling

Copyright © 2010 Bookware 1865-682x/10/09/001

Ein Teil des großen Ganzen

Java Portlets

von Werner Eberling

Die Erstellung von Web-Anwendungen ist in der heutigen Projektlandschaft scheinbar zur Routine geworden. Doch für sich allein stehende Web-Anwendungen reichen inzwischen nicht mehr aus. Die Schlagwörter heissen: *Personalisierung*, *SingleSignOn* oder *Content-Management*. Anders ausgedrückt: Portale werden immer beliebter. In diesem Zusammenhang bietet der *JavaPortlet-Standard* eine solide Basis zum Aufbau von Web-Anwendungen im Portal. Um diesen, teilweise noch recht unbekannten Standard aus seinem vermeintlichen Schattendasein herauszuholen, soll er in diesem Artikel etwas genauer betrachtet werden.

Man mag dazu stehen, wie man möchte, Web-Anwendungen sind aus unserem Leben nicht mehr wegzudenken. Wer kauft nicht seine Bücher bei Amazon, holt sich die neueste Wettervorhersage bei *wetter.com* oder fragt, bevor er (oder sie) lange Bücher wälzt, nicht einfach Google nach Informationen zum gerade zu lösenden Problem. Der Browser ist inzwischen zu der „Allround-Applikation" auf den meisten Desktops aufgestiegen.

Doch damit nicht genug. Hat es vor einigen Jahren noch ausgereicht, einfach eine Dienstleistung ins Internet zu stellen, so heisst das Zauberwort heutzutage *Personalisierung*. Kaum angemeldet, schlägt Amazon scheinbar passende Neuerscheinungen vor und anstatt sich mühsam zur Wettervorhersage für den eigenen Wohnort durchzuklicken, genügt ein Login auf der passenden Seite und schon stehen die Vorhersagen der persönlichen Lieblingsregionen zur Verfügung. Der Preis des Ganzen: eine Flut von Benutzernamen und Passwörtern. Wäre es nicht viel schöner, anstatt vieler einzelner Seiten eine Seite mit all den gewünschten Daten zur Verfügung zu haben? Nur einmal Benutzername, Passwort und Internetadresse eingeben und schon hat man Zugriff auf alle personalisierten Dienste, die man sich mit der Zeit aufgebaut hat.

Genau hier kommen Portale ins Spiel. Um sich beispielsweise eine Seite zusammenzustellen, die oben genannte Dienste umfasst, reicht es aus, sich bei Google zu registrieren und seine Suchseite zu personalisieren [1]. Kleine Web-Anwendungen, denen Google den Namen *Gadgets* verpasst hat, die nun die Wettervorhersage, die Suche bei Amazon oder die Darstellung der neuesten Weltnachrichten übernehmen, sind dabei gleich inklusive und können zur Gestaltung der eigenen Goggle-Startseite herangezogen werden.

Portale und Portlets

Was braucht es nun, um solche Arten von Seiten zu realisieren? Erst einmal eine Web-Anwendung, die es dem Benutzer ermöglicht, sich anzumelden und ggf. den Inhalt der ihm präsentierten Web-Seiten zu ändern. Kurz gesagt: ein Portal. In diesem Portal sollen wiederum eine Reihe kleiner Web-Anwendung betrieben werden, die der Anwender (oder ein Administrator) verwenden kann, um sich seine persönliche Web-Seite zusammenzustellen.

Web-Anwendungen in Java bedeuten in der Regel *JavaServlets*. Diese Technologie kann hier allerdings nicht verwendet werden, da ein Servlet seines Zeichens immer eine komplette Internetseite darstellt, die gerade beschriebenen Anwendungen aber nur Teil einer grösseren Seite sind. Aus diesem Grund wurde bereits im Jahre 2003 eine neue Java Web-Komponente aus der Taufe gehoben: das *Portlet* [1] [2].

[1] Der Vollständigkeit halber sei hier erwähnt, dass es sich bei *Portlets* um eine *server*-seitige Technologie handelt, wobei die zu Beginn erwähnten Google-Gadgets eine *client*-seitige Lösung darstellen. Beide Ansätze adressieren aber (wenn auch auf technisch unterschiedliche Art und Weise) das selbe Thema: die Personalisierung von Web-Seiten durch kleine Teil-Applikationen.

Anders als ein Servlet ist ein Portlet nur für einen Teil der darzustellenden Seite verantwortlich. Dies macht sich unter anderem dadurch bemerkbar, dass HTML-, HEAD- oder BODY-Tags nicht von jedem einzelnen Portlet, sondern ausschliesslich von der Portlet-übergreifenden Portal-Anwendung erzeugt werden. Das hört sich erst einmal nicht all zu kompliziert an. Allerdings stellt sich darüber hinaus die Frage, was passiert, wenn ein Benutzer nun mit einem der dargestellten Portlets interagiert, in dem er einen Link drückt oder ein Formular ausfüllt und absendet. Hier sollte nur das betroffene Portlet die Informationen über den gedrückten Link bzw. die Daten aus dem ausgefüllten Formular erhalten. Die anderen Portlets der selben Seite könnten mit diesen in der Regel auch relativ wenig anfangen.

Anzeige oder Interaktion ?

Damit sind wir an einer zentralen Stelle des Portlet-Standards angekommen: der Unterscheidung zwischen Aktion und Darstellung. Die logische Unterscheidung zwischen der Reaktion auf eine Interaktion des Anwenders und die Darstellung der entsprechenden Antwort ist bereits aus dem MVC-Pattern [3] bekannt. Der Portlet-Standard geht allerdings noch einen Schritt weiter, in dem er diese Trennung in zwei verschiedene Phasen innerhalb eines Anfrage-/Antwort-Zyklus giesst (siehe Abbildung 1).

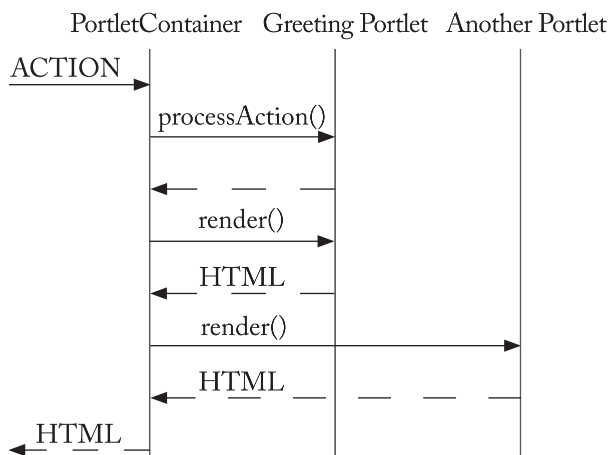

Abbildung 1

Der Portlet-Container, der die Anfrage entgegen nimmt, wird diese nur an das Portlet weiterleiten, das von der Interaktion des Benutzers betroffen ist. Dieses Portlet bekommt die entsprechenden Daten im Rahmen eines *Action-Requests* übermittelt (es befindet sich jetzt in der sogenannten *Action-Phase*). Bei der Abarbeitung dieses Requests ändert es seinen Zustand gemäß der empfangenen Daten. Dabei wird keine Ausgabe in Form von HTML produziert. Nach erfolgreicher Abarbeitung des *Action-Requests* erhalten alle Portlets der aktuellen Seite (auch das Portlet, das gerade den *Action-Request* erhalten hat) über einen sogenannten *Render-Request* die Aufforderung sich neu darzustellen. Damit wird die sogenannte *Render-Phase* eingeleitet. Als Antwort auf diese Anfrage erzeugt jedes Portlet ein, seinem Zustand entsprechenden, HTML-Schnipsel. Die so entstehenden HTML-Einzelteile werden vom Portlet-Container zu einer Gesamtseite zusammengesetzt, die dem Benutzer als Antwort auf seine Interaktion präsentiert wird.

So sieht's in Java aus

Soweit der konzeptionelle Ablauf. Aber wie sieht das Ganze nun in Java aus? Der Portlet-Standard definiert das Interface *javax.portlet.Portlet*, das jedes Portlet implementieren muss. Hier sind, neben Lebenszyklus-Methoden zur Initialisierung und zur Zerstörung des Portlets, die Methoden zur Abarbeitung von *Action-* und *Render-Requests* definiert:

```
public void processAction (
  ActionRequest request, ActionResponse response
)
public void render (
  RenderRequest request, RenderResponse response
)
```

Um nun nicht jedes mal alle Methoden des Interfaces implementieren zu müssen, stellt der Standard mit *javax.portlet.GenericPortlet* eine Adapter-Klasse bereit, die von konkreten Portlet-Implementierungen genutzt werden kann. Das klassische HelloWorld Beispiel sähe damit wie folgt aus:

GreetingPortlet.java

```
import javax.portlet.*;
import java.io.IOException;
public class GreetingPortlet extends GenericPortlet {
  @Override
  public void render(
    RenderRequest request, RenderResponse response
  ) throws PortletException, IOException {
    request.setAttribute(
      "name",
      request.getPortletSession().
      getAttribute("name")
    );
    getPortletContext()
      .getRequestDispatcher("/hello.jsp")
      .forward(request, response);
  }
```

```
    @Override
    public void processAction(
        ActionRequest request, ActionResponse response
    ) throws PortletException, IOException {
        request.getPortletSession().setAttribute(
            "name",
            request.getParameter("name")
        );
    }
}
```

Wie aus der Welt der *JavaServlets* bekannt, kann auch bei der Implementierung von Portlets eine *Java Server Page* (JSP) zur Darstellung der Seite verwendet werden. Da allerdings, wie bereits erwähnt, Links und Formulare vom *Portlet-Container* entgegengenommen und an das jeweils betroffene Portlet weitergereicht werden, müssen die verwendeten URLs vom Container generiert werden. Hierzu steht eine eigene Tag-Bibliothek[2] zur Verfügung, die innerhalb der JSP genutzt werden kann.

hello.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="p" uri="http://java.sun.com/portlet_2_0" %>
<div>
  <div style='padding: 5px;'>
    <h2>Hallo Portlet-Welt!</h2>
    <p>Hier gibt's kostenlose Gr&uuml;sse aus dem Portal:</p>
    <p>
        <p:actionURL var="action"/>
        <form method='POST' action='${action}'>
          Wie heisst Du?   
          <input type='text' name='name'>  
          <input type='submit' value='Gr&uuml;sse bitte!'>
        </form>
    </p>
  </div>
  <c:if test="${not empty name}">
    <div style='padding: 5px;'>
      <p>
        <h4>
          <c:out value="Hallo ${name}, willkommen im Portal !" />
        </h4>
      </p>
    </div>
  </c:if>
</div>
```

Natürlich kommt auch der Portlet-Standard nicht ohne den obligatorischen Deskriptor (hier: *portlet.xml*) aus. In dieser XML-Datei werden unter anderem die Portlet-Klasse, der Name des Portlets und evtl. vorhandene Initialisierungsparameter angegeben. Zusätzlich wird definiert, welche Anzeige-Modi das Portlet unterstützt. Hier wird zwischen *view* (der normalen Anzeige), *edit* (dem Bearbeitungsmodus) und *help* (einer Hilfeansicht) unterschieden.

[2] http://java.sun.com/portlet_2_0

Ein Portlet sollte sinnvollerweise mindestens den *view*-Modus unterstützen. In wieweit die beiden anderen Anzeige-Modi möglich sind, obliegt der Entscheidung des Portlet-Entwicklers. Die Umschaltung zwischen den einzelnen Anzeige-Modi erfolgt über den Portlet-Container. Von Seiten des Portlets ist hier also kein weiterer Implementierungsaufwand notwendig. Die Adapterklasse *GenericPortlet* bietet für jeden der angesprochenen Modi eine eigene Darstellungsmethode an. Portlets, die diese Klasse erweitern, überschreiben daher in der Regel nicht die Methode *render(...)*, sondern, je nach unterstütztem Portlet-Modus, die Methoden *doView(...)*, *doEdit(...)* oder *doHelp(...)*.

portlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app
  xmlns="http://java.sun.com/xmlns/portlet/portlet-app_2_0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/
    portlet-app_2_0.xsd/etc/opt/SUNWportal/dtd/portlet.xsd"
  version="2.0"
>
  <portlet>
    <portlet-name>greetingPortlet</portlet-name>
    <portlet-class>
      de.mathema.portlet.GreetingPortlet
    </portlet-class>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>VIEW</portlet-mode>
    </supports>
    <portlet-info>
      <title>greeting.portlet</title>
    </portlet-info>
  </portlet>
</portlet-app>
```

Abbildung 2

Einpacken bitte

Nun haben wir also eine Portlet-Klasse, eine JSP und einen Deskriptor. Wie kommt das Ganze nun in den Portlet-Container? Da Portlets, wie bereits erwähnt, im Prinzip kleine Web-Anwendungen sind die innerhalb einer großen Web-Anwendung (dem Portal) laufen, ist es naheliegend, dass sie in Form eines *Web-Archivs* (WAR) verpackt und verteilt werden. Die Portlet-Klasse lan-

det dabei im Verzeichnis */WEB-INF/classes*, die JSP im Wurzelverzeichnis und der Deskriptor wie gewohnt unter */WEB-INF*.

Da ein Web-Archiv immer einen Web-Deploymentdeskriptor (*web.xml*) enthalten muss, kommen wir nicht umher einen solchen, leeren Deskriptor zu erstellen und mit in das Portlet-WAR zu packen.

Portlet 2.0 oder: Events und Ressourcen

Nun können wir also eine Web-Seite aufbauen, die aus einer Vielzahl kleiner, nebeneinander existierender Web-Anwendungen besteht. Dabei ist es nicht schwer sich ein Szenario auszudenken, in dem es wünschenswert wäre, dass die einzelnen Anwendungen miteinander kommunizieren können. Unter anderem hat sich die Version 2.0 des Portlet-Standards dieser Anforderung angenommen [4], der seit 2008 verfügbar ist. Herausgekommen ist dabei das Konzept der *Portlet-Events*. Möchte ein Portlet mit anderen Portlets kommunizieren, so kann es, innerhalb der Action-Phase, einen *Event* erzeugen, der wiederum von anderen Portlets empfangen werden kann. Ein solcher *Event* wird dabei durch einen Namen gekennzeichnet und kann neben quasi primitiven Java-Typen (*Integer, String, Date*, …) beliebige Klassen enthalten, solange diese serialisierbar und über das *Java XML Binding* (*JAXB*) [5] in XML umwandelbar sind.

Beispiel einer per Event transportierbaren Klasse.

```
@javax.xml.bind.annotation.XmlRootElement
public class Registration implements java.io.Serializable{
    private String name;
    private String firstName;
    private String email;
    public Registration() {
    }
   // … evtl. weitere Konstruktoren …

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    //... weitere Getter und Setter ...
}
```

Portlets, die einen *Event* verschicken wollen, müssen diesen in ihrem Deskriptor definieren und dort auch bekannt geben, dass sie Erzeuger dieses *Events* sind. Ist ein *Event* zwar definiert, aber das entsprechende Portlet nicht als Erzeuger dieses *Events* markiert, kann das Portlet den *Event* zwar erzeugen und versenden, der Container kann (und wird) diesen *Event* aber verwerfen, da er nicht über den Deskriptor bekanntgegeben wurde.

Versenden eines Events in der Action-Phase

```
public class RegistrationPortlet extends GenericPortlet{
    //... Attribute und weitere Methoden ...
    @Override
    public void processAction(
        ActionRequest request, ActionResponse response
    ) throws PortletException, IOException {
        String name = request.getParameter("name");
        String firstname = request.getParameter("firstname");
        String email = request.getParameter("email");
        Registration registration = new Registration(
            name, firstname, email
        );
        response.setEvent("registration", registration);
        setState(request, State.ACK);
    }
}
```

Konfiguration einer Event-Erzeugung im portlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app
  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/
    portlet-app_2_0.xsd/etc/opt/SUNWportal/dtd/portlet.xsd"
  version="2.0"
>
  <portlet>
    <portlet-name>portlet-registrationPortlet</portlet-name>
    <portlet-class>
      de.mathema.portlet.RegistrationPortle
    </portlet-class>
    ...
    <supported-publishing-event>
      <name>registration</name>
    </supported-publishing-event>
  </portlet>
  <event-definition>
    <name>registration</name>
    <value-type>
      de.mathema.portlet.Registration
    </value-type>
  </event-definition>
</portlet-app>
```

Abbildung 3

Interessiert sich ein Portlet für bestimmte *Events*, so muss es das Interface *javax.portlet.EventPortlet* implementieren, das die Methode *public void processEvent*(*EventRequest request, EventResponse response*) definiert und sich über seinen Portlet-Deskriptor für den *Event* registrieren. Wird im Rahmen einer Action-Phase nun ein entsprechender *Event* erzeugt, so wird, nach Ende der Phase, jedes Portlet, das sich für diesen *Event* registriert hat, über die oben genannte Methode benachrichtigt (siehe Abbildung 4). Auf den empfangenen *Event* kann das Portlet über die Methode *getEvent()* am *EventRequest* zugreifen. Dieser bietet u.a. Methoden zum Auslesen der übertragenen Daten.

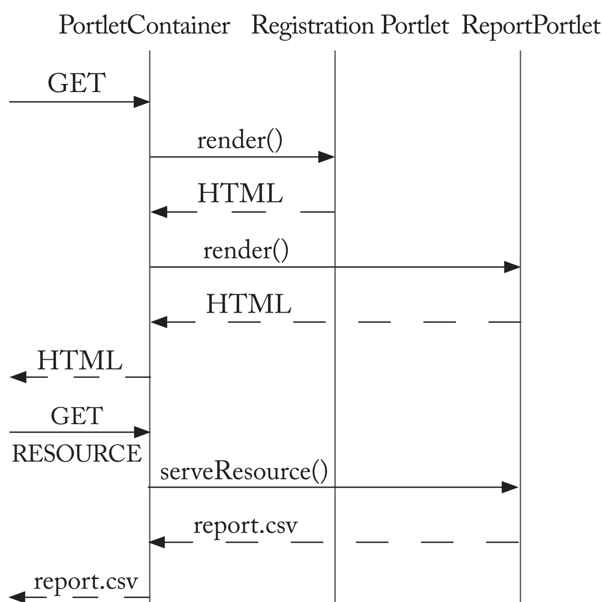

Abbildung 4

Empfang und Verarbeitung eines Portlet-Events

```
public class ReportPortlet extends GenericPortlet {
  //... Attribute und weitere Methoden ...
  @Override
  public void processEvent(
    EventRequest request, EventResponse response
  ) throws PortletException, IOException {
    if (
      "registration".equals(request.getEvent().getName())
      &&
      request.getEvent().getValue() != null
    ) {
      List<Registration> registrations =
        getRegistrations(request);
      registrations.add(
        (Registration) request.getEvent().getValue()
      );
    }
  }
}
```

Konfiguration eines Event-Empfangs im portlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app
  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-
    app_2_0.xsd/etc/opt/SUNWportal/dtd/portlet.xsd"
  version="2.0"
>
  <portlet>
    <portlet-name>reportPortlet</portlet-name>
    <portlet-class>
      de.mathema.portlet.ReportPortlet
    </portlet-class>
    ...
    <supported-processing-event>
      <name>registration</name>
    </supported-processing-event>
  </portlet>
  <event-definition>
    <name>registration</name>
    <value-type>
      de.mathema.portlet.Registratio
    </value-type>
  </event-definition>
</portlet-app>
```

Abbildung 5

Ich habe hier etwas für Dich

Neben dem *Portlet-Eventing* adressiert die Version 2.0 des Portlet-Standards auch das Ausliefern von Ressourcen, eine Problemstellung, die dem Web-entwicklungserfahrenen Leser mit Sicherheit bekannt vorkommen dürfte. Wie oft stand man schon vor dem Problem dynamisch erzeugte Daten, wie z.B. ein Bild, eine PDF-Datei oder eine *csv*-Datei über einen Link zur Verfügung stellen zu müssen. Der Ausweg war normalerweise das klassische *Streaming*- oder *Download*-Servlet, das sich um die Übertragung der Daten kümmerte und über eine spezielle URL ansprechbar war.

Der Portlet-Standard schafft hier Abhilfe über sogenannte *Resource-URLs*. Diese speziellen Links werden nicht als Benutzeraktion, sondern als Anfrage einer Res-

source interpretiert, und in Form eines *ResourceRequests* an das betroffene Portlet weitergereicht (siehe Abbildung 6). Dieses Portlet ist nun dafür zuständig, die angefragte Ressource auszuliefern. Um diese eindeutig zu identifizieren, können *Resource-URLs* mit sogenannten *Resource-IDs* versehen werden.

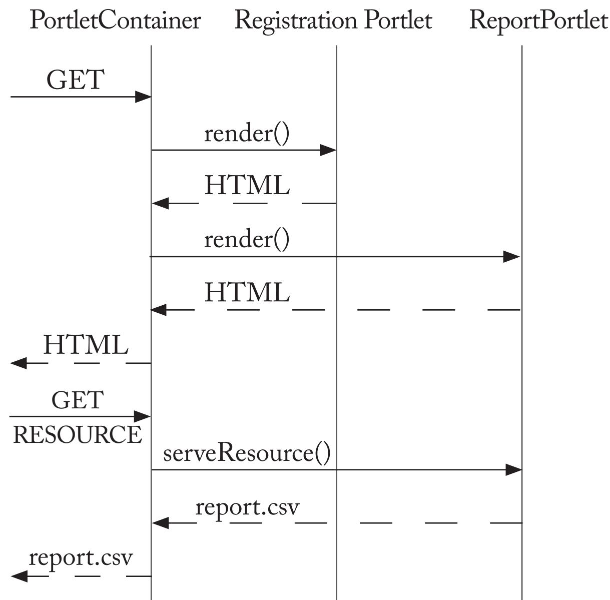

Abbildung 6

Um Ressourcen ausliefern zu können, muss ein Portlet das Interface *javax.portlet.ResourceServingPortlet* implementieren bzw. die Methode *serveResource(ResourceRequest request, ResourceResponse response)* der Adapter-Klasse *javax.portlet.GenericPortlet* überschreiben.

Auslieferung einer csv-Datei aus einem Portlet

```
public class ReportPortlet extends GenericPortlet {
  // … Zustand und weitere Methoden
  @Override
  public void serveResource(
    ResourceRequest request,
    ResourceResponse response
  ) throws PortletException, IOException {
    if (
      "ReportPortlet.Download"
        .equals(request.getResourceID())
    ) {
      response.setContentType("text/plain");
      PrintWriter writer = response.getWriter();
      writer.write(getRegistrationsAsCSVString(request));
      writer.flush();
      writer.close();
    }
    ...
}
```

Da *Resource-URLs* der Darstellung dienen, können sie nur während der Render-Phase erzeugt werden. Hierzu bietet die Klasse *RenderResponse* eine Methode *createResourceURL*. Das zurückgelieferte Objekt besitzt unter anderem eine Methode zum Setzen der Resource-ID.

Werden JSPs zur Realisierung des *Views* verwendet, so kann die *Resource-URL* auch über das entsprechende Tag, der schon im ersten Beispiel für die Erzeugung von Action-URLs verwendeten, *Portlet-Taglib* erzeugt werden.

Verwendung von Resource-URLs innerhalb einer JSP-Seite

```
<%@ taglib prefix="p" uri="http://java.sun.com/portlet_2_0" %>
...
<p style='margin-top:20px;'>
  <p:resourceURL
    var="downloadURL"
    id="ReportPortlet.Download"
  />
  <p:resourceURL
    var="iconURL"
    id="AbstractResourcePortlet.CsvIcon"
  />
  <a href='${downloadURL}' target='_BLANK'>
    <img
      src='${iconURL}'
      style='padding-right:5px;margin-top:-6px;'
    />
    Als Liste herunterladen
  </a>
</p>
```

Genug für's Erste

Der Portlet-Standard bietet noch eine Reihe weiterer interessanter Features (*Public Render-Parameter*, erweitertes *Request Dispatching*, …), für einen ersten Einblick soll es jetzt aber erst einmal genug sein. Es bleibt festzuhalten, dass sich die Entwicklung auf Basis von *JavaPortlets* sehr schnell vertraut anfühlt. Gleichzeitig können, dank der Möglichkeiten des Portals, die eigenen Anwendungen durch fremde Portlets erweitert werden, sodass das Rad nicht immer neu erfunden werden muss.

Fazit und Ausblick

Natürlich sind Portlets nicht die Generallösung für jede Web-Anwendung. Sie bieten aber eine einfache Möglichkeit mit einer Anwendung Teil eines grösseren Portals zu werden. Gleichzeitig liefern sie eine Reihe interessanter Konzepte (*Eventing, Resource-Handling*), die den Aufbau größerer Web-Anwendungen vereinfachen können. So bietet es sich beispielsweise an, den Portlet-Ansatz in grösseren Projekten zur Modularisierung zu verwenden. Erfolgt die Entwicklung einer Web-Anwendung in einem größeren Team, so kann sich jedes Mitglied erst einmal auf seine Teilkomponente (*Portlet*) konzentrieren und diese, unabhängig von den anderen Komponenten, zu deployen und zu testen.

Wie bei *JavaServlets* ist auch bei *JavaPortlets* beim „puren“ Standard natürlich noch nicht Schluss. Auch hier wünscht man sich die Vorzüge moderner Web-Frameworks wie z.B. JSF oder Wicket. Mit einer dünnen Adapterschicht sind diese Frameworks auch im Portal einsetzbar. Im Falle von JSF gibt es diese Schicht als sogenannte *JSF-Portletbridge* sogar standardisiert [6][7][8]. Durch den Einsatz dieser Web-Frameworks wird die Portlet-Technologie zu einem technischen Detail, das bei der Entwicklung kaum noch ins Gewicht fällt, aber bei Bedarf seine technischen Fähigkeiten ausspielen kann.

PortalServer

Ein Thema, das in diesem Artikel bewusst ausgespart wurde ist das Thema *Portalserver*, da damit der Umfang dieses Artikels gesprengt worden wäre. Wer sich aber nach der Lektüre näher mit dem Thema *JavaPortlets* befassen möchte, dem seien an dieser Stelle die *PortalServer* von Liferay [9] oder JBoss [10] empfohlen. Beide Server sind frei verfügbar und relativ problemlos aufzusetzen.

Referenzen

[1] iGoogle,
http://www.google.com/ig

[2] Java Community Process *JSR 168 – Portlet 1.0*,
http://jcp.org/en/jsr/detail?id=168

[3] Wikipedia *Model View Controller – Pattern (MVC)*,
http://de.wikipedia.org/wiki/Model_View_Controller

[4] Java Community Process *JSR 286 - Portlet 2.0*,
http://jcp.org/en/jsr/detail?id=286

[5] GlassFish *JAXB – Java XML Binding*,
https://jaxb.dev.java.net

[6] Java Community Process *JSR 329 - Portlet 2.0 Bridge for JSF 1.2,*
http://www.jcp.org/en/jsr/detail?id=329

[7] JBoss Community *JBoss Portlet Bridge,*
http://jboss.org/portletbridge

[8] My Faces *Portlet Bridge,*
http://myfaces.apache.org/portlet-bridge/index.html

[9] Liferay *Portal Server,*
http://www.liferay.com

[10] JBoss Community *GateIn Portal Server,*
http://www.jboss.org/gatein

Kurzbiographie

Werner Eberling ist Principal Consultant und Technical Lead bei der MATHEMA Software GmbH in Erlangen. Seit 1999 beschäftigt er sich mit verteilten Systemen, sein Fokus liegt dabei auf CORBA und der JEE. Neben seiner Projekttätigkeit hält er Technologie-Trainings in Deutschland und der Schweiz und ist als Sprecher auf nationalen und internationalen Konferenzen anzutreffen. Er ist Autor des Buches *Enterprise JavaBeans 3. Das EJB3-Praxisbuch für Ein- und Umsteiger*, erschienen im Hanser Verlag.