Das Web-Framework Apache Struts in Version 1.0: Power für Web-Anwendungen

von Muhammet Öztürk

Fliegen in der Nacht

Eine der sechs so genannten Best Practices in der Softwareentwicklung ist die Verwendung komponentenbasierter Architekturen [6]. Die komponentenbasierte Entwicklung ist die beste Vorgehensweise in der Softwareentwicklung, da diese Architektur ein effizientes Maß an Wiederverwendbarkeit, eine klare Arbeitsaufteilung zwischen den Entwicklungsteams und die unabhängige Entwicklung der Komponenten ermöglicht und nicht zuletzt die Wartung der Software stark vereinfacht.

Mit der three-tier-architecture bzw. n-tier-architecture trennt man die verschiedenen Komponenten (Präsentations-, Geschäftslogik- und Persistenzschicht) voneinander, sodass die Komponenten miteinander über eine definierte Schnittstelle kommunizieren. Die oben genannten Komponenten können weitere Komponenten beinhalten, die wiederum miteinander über eine definierte Schnittstelle kommunizieren – ein Beispiel hierfür ist das bekannte Model-View-Controller-Prinzip (MVC).

Als Sun die Technik der Java Servlets der Weltöffentlichkeit vorstellte, waren viele Programmierer hoch erfreut, weil diese schneller und mächtiger war als der bis dahin vorherrschende CGI-Ansatz. Aber das Problem war, dass das Schreiben von HTML-Seiten mit endlosen *print()*-Anweisungen in der Servlet-Klasse zum einen sehr lästig, zum anderen für die Weiterentwicklung der Views extrem mühsam war. Es gab nur eine Servlet-Klasse, die die Aufgabe von drei Komponenten (MVC) ausführte. Die Antwort auf dieses Problem kam prompt mit den JavaServer Pages. Durch JSP haben die Entwickler die Möglichkeit, HTML und Java in einer JSP-Seite zu programmieren. Dadurch befreit JSP die Entwickler von der Implementierung der unschönen *print()*-Anweisungen in der Servlet-Klasse. Eine JSP ist indessen nichts anderes als ein Servlet: die JSP-Dateien werden zunächst in eine *.java*-Datei konvertiert, dann zu einer *.class*-Datei kompiliert und anschließend ausgeführt.

Quellcode auf CD!

Innerhalb kurzer Zeit wurde JSP für Web-Entwickler äußerst populär und hat sich im Web-Bereich durchgesetzt. Für kleine Anwendungen wäre das neue JSP-Konzept, das MVC Model 1 genannt wird, in dem sich Daten, Geschäftslogik und View in einer JSP befinden, sicherlich eine Lösung. Wenn die Views und die Geschäftslogik der Anwendung allerdings etwas komplexer werden, erreichte man sofort die Grenzen dieses Ansatzes. Für die Entwicklung von JSP-Seiten braucht man Entwickler, die sowohl GUI-Design- als auch Java-Programmierkenntnisse haben, da sonst die Ergebnisse entweder schlechter Java-Code oder unschöne HTML-Seiten sind. Außerdem sind JSP-Seiten, die HTML-, JavaScript- und Java-Code enthalten, ziemlich unüberschaubar. Zudem würde eine Änderung in der Geschäftslo-

gik eine neue Implementierung aller betroffenen JSP-Seiten erfordern.

Aus oben genannten Gründen war es für viele Programmierer klar, dass man aus beiden Komponenten (JSP und Servlets) ein neues Konzept erstellen musste. Die Servlets als Controller steuern den Control-Workflow, während die JSPs als Views nur die Daten darstellen. Durch die Benutzung beider Komponenten (JSP und Servlets) entstand das MVC Model 2, welches für Web-Applikationen vom SmallTalk Model-View-Controller-Entwurfsmuster [5] abgeleitet wurde. Bei dem MVC Model 2 delegiert der Controller (Servlets) die Requests, in unserem Fall die HTTP Request, zum jeweiligen Handle. Das Handle ist ein Adapter zwischen Request und Model. Der Controller gibt als Response auf die Requests die jeweilige JSP-Seite zurück, welche die Daten aus dem Model bekommt und präsentiert. Die bekannte *stateless*-Verbindung zwischen Client und Server in einer Web-Anwendung macht es schwer für das Model, im Falle einer Änderung die View zu benachrichtigen. Um die Änderungen im Model festzustellen und im View anzuzeigen, muss die View bzw. der Browser neue Requests an den Server schicken, statt vom Model benachrichtigt zu werden.

Struts Framework mit MVC Model 2

Struts ist ein MVC Model 2 basiertes Open-Source-Framework für die Präsentationsschicht von Web-Anwendungen. Es besteht aus kooperierenden Klassen, Servlets und JSP-Tags. Das Framework läuft auf einem JSP 1.1- bzw. Servlet 2.2-tauglichen Web Container wie z.B. Tomcat. Durch die Einführung des MVC Model 2-Konzepts trennt das Framework die Komponenten Model, View und Controller transparent voneinander und stellt dem Entwickler komfortable und einfache Schnittstellen zur Realisierung von Web-Applikationen zur Verfügung. Neben dieser komponentenbasierten Architektur ermöglicht das Framework, die Ressourcen effektiv einzusetzen. Zum einen benötigen die JSP-Entwickler keine Java-Kenntnisse, um die JSP-Seiten zu implementieren, zum anderen übernimmt der Standard-Controller die lästigen Konvertierungen des HTTP-Protokolls, sodass sich die Entwickler auf die Kernaufgabe konzentrieren können. Abbildung 2 zeigt eine Übersicht der Struts-Komponenten.

Client/Browser

Der Browser, in dem der Benutzer mit der View Interaktionen ausführt, sendet einen HTTP-Request an den Web-Server. Der Web-Server leitet diesen HTTP-Request an den Applikations-Server. Der Applika-

Klassisches MVC-Konzept

Sinn und Zweck des MVC-Konzepts ist es, ein Entwurfsmuster bereitzustellen, welches die View (Präsentation), das Model (Datencontainer) und den Controller (Ablaufsteuerung) transparent voneinander trennt. Das MVC-Design-Entwurfsmuster [5] stammt aus SmallTalk und ist mittlerweile Standard in der Softwarearchitektur. Das Konzept eignet sich besonders dann, wenn verschiedene Views dieselben Daten darstellen. Die einzelnen Komponenten besitzen die folgenden Aufgaben:

- Model: Als Model wird die Komponente bezeichnet, die die Datenstruktur der Anwendung definiert. Das Model speichert die Daten und somit den Zustand der Anwendung und stellt die Methoden zur Änderung der Daten zur Verfügung.
- View: Als View wird die Komponente bezeichnet, die die Daten des Models auf dem Bildschirm darstellt. Der Benutzer führt auf der View die Aktionen aus, die durch den Controller an das Model weitergeleitet werden.
- Controller: Als Controller wird die Komponente bezeichnet, die auf die Interaktionen der Anwender in der View reagiert. Sie überprüft die Benutzereingabe und ruft die jeweilige Methode des Models auf.

Die drei Komponenten Model, View und Controller müssen sich gegenseitig unterstützen, um die oben genannten Funktionen ausführen zu können. In Abbildung 1 werden die Relationen zwischen Model, View und Controller als UML-Diagramm dargestellt. Die blauen Linien stellen die Relationen und die grauen Linien die Kommunikationsrichtungen zwischen den Komponenten dar.

Das Model kennt keine der Komponenten (View und Controller), weshalb es unabhängig von beiden realisiert werden kann. Es wird auch als Kern der Anwendung bezeichnet und, in der Regel, als erstes implementiert. Die Daten des Models können von verschiedenen Views dargestellt werden. Ändern sich die Daten des Models, so werden alle Views über diese

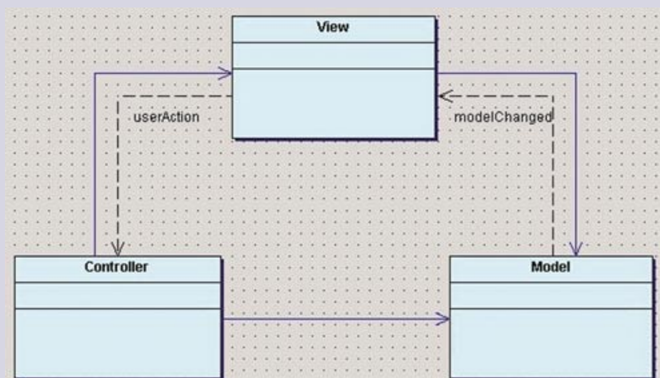

Abb. 1: Die Relationen zwischen Model, View und Controller

Änderung benachrichtigt. Diese Benachrichtigung erfolgt nach dem Observer-Entwurfsmuster [5]. Die Views werden beim Model als Observer registriert, sodass alle Views, im Falle einer Änderung der Daten (im Model), vom Model benachrichtigt werden können.

Die View kennt das Model und hat Zugriff auf die *getter*-Methoden des Models. Sie bietet dem Benutzer die Möglichkeit, Eingaben vorzunehmen. Diese Eingaben bewirken keine direkten Veränderungen im Model, sondern werden an den Controller weitergeleitet, welcher die Änderungen im Model ausführt. Die View kennt den Controller nicht. Der Controller wird mittels Events über die Benutzereingaben informiert. Er ist als EventListener bei der View registriert, wertet die Benutzereingabe aus und ruft die entsprechenden Methoden im Model auf.

Der Controller kennt dagegen die beiden Komponenten View und Model. Er reagiert auf die Benutzereingabe und hat Zugriff auf die *setter*-Methoden des Models. Mittels dieser Methoden führt der Controller die Änderungen im Model aus.

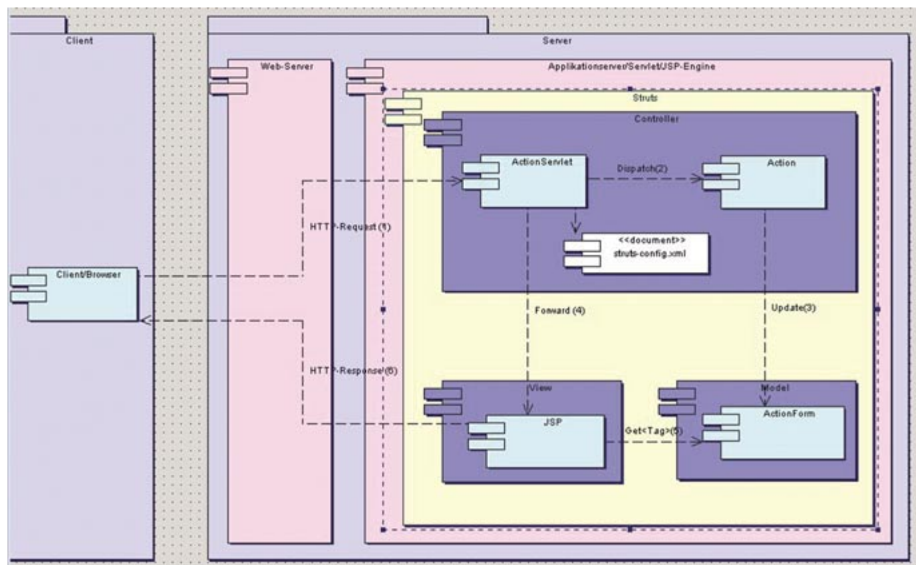

Abb. 2: Struts im Überblick

tions-Server ist so vorkonfiguriert, dass er diesen HTTP-Request direkt an das *ActionServlet* weiterleitet (Abb. 2).

Controller

Das *ActionServlet* und die *Action* bilden zusammen die Controller-Komponente. Diese beiden Klassen steuern die Abläufe der Anwendung mit klar definierten Aktivitäten. Während sich das *ActionServlet* um das Mapping der Requests zu den spezifizierten Actions kümmert, steuert die Action die Abläufe der Applikation und dient als Wrapper der Geschäftslogik.

ActionServlet: Das *org.apache.struts.action.ActionServlet* ist der wichtigste Baustein des Controllers, der die HTTP-Requests empfängt, umwandelt und diese dann zur jeweiligen Action-Klasse schickt. Das ActionServlet benutzt dabei die *struts-config.xml*-Konfigurationsdatei (siehe Listing 5), in der das Mapping zwischen HTTP-Requests und den Action-Objekten definiert ist. In dem folgenden Codeabschnitt aus der *struts-config.xml*-Datei wird die Action *startAppAction* mit der Action-Klasse *com.oisoft.view.action.Start AppAction* gemappt, die diesen Request ausführt. Das ActionServlet erzeugt beim ersten Starten die *org.apache.struts.action.ActionMapping*-Objekte aus den in der *struts-config.xml*-Datei definierten Informationen und speichert diese Objekte in einem Container-Object der Klasse *org.apache.struts.action.ActionMappings*. Mittels dieser gespeicherten ActionMapping-Objekte entscheidet das ActionServlet, welches ActionMapping für den jeweiligen Request verantwortlich ist. Danach wird die *perform()*-Methode des jeweiligen Action-Objekts ausgeführt (siehe Listing 4). Das ActionServlet ist hier als Command-Entwurfsmuster [5] implementiert, welches die jeweiligen HTTP-Requests an die gemappte Action-Klasse weiterleitet. Neben dem ActionServlet beinhaltet das ActionMapping-Objekt zusätzliche Informationen aus der *struts-config.xml*-Datei. Der Inhalt des Attributs *name* beschreibt den Variablennamen, unter dem das Model (ActionForm) im entsprechenden Scope gespeichert wird, *input* beschreibt die Input HTML/JSP-Form und der Inhalt des Attributs *forward* steuert die Abläufe nach der Ausführung der Action-Klasse. Jeder Action können beliebige *forward*-Attribute zugeordnet werden. Dem Attribut *forward* sollte ein logischer Name, wie z.B. *success* oder *failed*, zugeordnet werden. Die Methode *perform()* der Action-Klasse bestimmt durch den Rückgabewert *return (mapping.findForward("success"))* den Ablauf der Anwendung. Das ActionServlet führt die im *path* angegebene Anweisung aus. Sie haben die Möglichkeit, neben dem Action-spezifischen *forward*-Eintrag auch globale *forwards* zu definieren (siehe Listing 1). Diese sind dann für alle Actions global verfügbar. In vielen Fällen ist diese Anweisung die Ausführung einer JSP-Seite. Es ist aber möglich, die Kontrolle durch die Definition neuer Actions *<forward name="something"path="/something.do"/>* wieder an eine andere Action weiterzuleiten. Um den Controller mit zusätzlicher Funktionalität zu erweitern, kann die Klasse *org.apache.struts.action.Action-Servlet* abgeleitet werden. Aber die Standardimplementierung dieser Klasse wäre in der Regel ausreichend.

```
<action  path="/startAppAction"

type="com.oisoft.view.action.StartAppAction"
 name="formModel"
 scope="session"
 validate="true"
 input="/FBNApplication.jsp">
 <forward name="success" path="/FBNApplication.jsp"/>
 <forward name="something" path="/something.do"/>
 <forward name="failed" path="/Logon.jsp"/>
</action>
```

Action: Die *Action*-Klasse ist ein Bestandteil des Controllers und eine Wrapper-Klasse der Geschäftslogik. Die *Action*-Klasse soll den Workflow und die Fehlerbehandlung der Applikation kontrollieren, aber keine Geschäftslogik beinhalten. Die Geschäftslogik der Anwendung soll in anderen Komponenten (Business-Komponenten) ausgeführt werden. Dadurch erreicht man eine saubere Trennung zwischen Präsentations-, Business- und Persistenzschicht, sodass die Business-Komponente mit beliebigen Präsentations-Komponenten verwendet werden kann. Man muss beachten, dass das Struts-Framework nur in der Präsentationsschicht einzusetzen ist. Bei den einfachen Anwendungen könnte man die Geschäftslogik in der *Action*-Klasse realisieren, wobei man die komponentenbasierte Architektur opfert. Für komplexe Anwendungen ist diese Trennung unbedingt notwendig. Um die *Action*-Klasse zu benutzen, muss diese von der Klasse *org.apache.struts.action.Action* abgeleitet und die Methode *perform()* überschrieben werden (Listing 4 – am Ende des Artikels bzw. auf der beiliegenden CD). Damit das *Action-Servlet* die *Action*-Klasse ausführen kann, muss die *Action*-Klasse in der *struts-config.xml*-

Konfigurationsdatei (Listing 5) angegeben werden. Das *ActionServlet* übergibt das initialisierte Objekt der Klassen *ActionMapping*, *Action- Form*, *HttpServletRequest*, *HttpServletResponse* an die Methode *perform()*:

```
public ActionForward perform
 (ActionMapping mapping,
  ActionForm form,
  HttpServletRequest request,
  HttpServletResponse response)
```

Der Rückgabewert dieser Methode *perform()* ist ein Objekt der Klasse *org.apache.struts.action.ActionForward* und dieses bestimmt, was als nächstes vom *ActionServlet* ausgeführt werden soll (Listing 5). Die in der *struts-config.xml*-Datei definierten *forward*-Einträge beschreiben die weiteren Abläufe der Anwendung. In einer Drei-Schichten-Architektur wären für die *Action*-Klasse die folgenden Anweisungen auszuführen:

- Business-Schicht erzeugt eine Verbindung zu der Datenbank
- Business-Schicht gibt die Ergebnisse zurück
- Die *Action*-Klasse aktualisiert das Model
- Die *Action*-Klasse gibt die Kontrolle an den Controller (ActionServlet) zum Anzeigen einer spezifizierten JSP-Seite, die auf Daten des Models mittels Struts-Tags zugreift und diese anzeigt.

Die *Action*-Klasse ist nach dem Adapter-Entwurfsmuster [5] implementiert, welches die Schnittstelle einer Klasse an eine andere vom Client erwartete Schnittstelle anpasst – in diesem Fall ist das *ActionServlet* also unser Client. Die von der Klasse *org.apache.struts.action.Action* abgeleitete *Action*-Klasse passt die Geschäftslogik an die Schnittstelle an, welche von Struts erwartet wird. Das ActionServlet erzeugt nur ein Objekt der *Action*-Klasse, welches nicht *thread safe* ist, und man darf daher auch keine Instanzvariable in den abgeleiteten Klassen anlegen.

Model

Das Model in Struts ist eine von der abstrakten Klasse *org.apache.struts.action.ActionForm* abgeleitete Klasse. Sie repräsentiert den Zustand der jeweiligen Views in einer Session oder einem Request und verwaltet die Daten der Views. Die von der abstrakten Klasse *ActionForm* abgeleiteten Klassen beinhalten in einfachster Form *Getter*- und *Setter*-Methoden, jedoch keine Geschäftslogik. In dieser Klasse sind zwei wichtige Methoden implementiert, die bei jedem Request (je nach Konfiguration) von dem *ActionServlet* aufgerufen werden. Mit der Methode *validate()* wird die Gültigkeit der Benutzereingabe in der HTML-Seite geprüft. Wenn das Attribut *validate* in der *struts-config.xml*-Datei auf *true* gesetzt ist, wird die Methode *validate()* vom *ActionServlet* automatisch vor dem Setzen der gemappten Felder in der Klasse *ActionForm* aufgerufen, um die Plausibilitätsprüfungen auszuführen. Die Methode *validate()* gibt ein Objekt der Klasse *ActionErrors* zurück, welchem als Container die Objekte der Klasse *ActionError* hinzugefügt werden, falls ein Fehler aufgetreten ist (Listing 1). Diese Fehler werden von Struts in der View (JSP) an der Stelle angezeigt, an der der Tag *<html:errors/>* definiert ist (Listing 3).

Eine andere wichtige Methode ist *reset()*, die bei jedem Request vom ActionServlet aufgerufen wird, um die Defaultwerte der Klasse zu setzen. Diese beiden Methoden können in der abgeleiteten Klasse überschrieben werden (Listing 1). Die Klasse der *ActionForm* ist in der *struts-config.xml*-Datei mit dem Eintrag *name= "formModel"* einer Action bzw. einem JSP zugewiesen. Durch das Benutzen der Struts-Tags hat JSP Zugriff auf das Model (Listing 3). Die folgenden Schritte werden vom Struts ausgeführt, wenn das ActionServlet einen Request empfängt:

- Wenn das *ActionServlet* einen Request erhält, überprüft der Controller in der *user-session*, ob ein Objekt der Klasse *ActionForm* für den aktuellen User erzeugt wurde. Wenn nicht, wird ein Objekt erzeugt und mit dem jeweiligem Key (*user*) und Value (*ActionForm*) in einer HashTable gespeichert.
- Für die Requests, die mit dieser Form (HTML-Seite) gemappt sind, werden die jeweiligen *setter*-Methoden der Klasse *ActionForm* aufgerufen. Zum Beispiel wird mit dem folgenden Tag in einer JSP-Datei ein Eingabefeld erzeugt. Dessen Inhalt wird von dem *ActionServlet* im *property bookedSeats* der Klasse *Form Model* durch den Aufruf der Methode *setBookedSeats* gesetzt:

```
<td><html:text name="formModel" property=
        "bookedSeats" size="15" maxlength="3" /></td>
```

- Die vom ActionServlet aktualisierte ActionForm wird als Übergabeparameter beim Aufruf an die Methode *perform()* der Klasse *Action* weitergegeben. Die Methode *perform()* holt die Daten aus der *ActionForm* und führt die jeweiligen Aktionen in der Geschäftslogik aus. Anschließend aktualisiert die *Action*-Klasse die *ActionForm* (Model) und gibt die Kontrolle an das *ActionServlet* ab, das je nach Rückgabewert (ein Objekt der Klasse *ActionForward*) die nächste Aktion ausführt (Listing 4). In den meisten Fällen wird eine JSP-Seite ausgeführt, welche die Daten des Models präsentiert.

View

Die View im Struts ist eine einfache JSP-Datei, welches die HTML- und Struts-Tags beinhaltet. Struts befreit die JSP-Seiten von Java-Code (scriptlet) durch die eigenen Tags. Eine detaillierte Dokumentation über die Struts-Tag-Library wird mit Struts mitgeliefert. Alle HTML-Tags sind mit Struts-eigenen Tags abgebildet, die JSP-Seiten beinhalten dabei weder Workflow noch Geschäftslogik. So benutzt Struts verschiedene Tags zum Zugriff auf die Daten des Models, zum Beispiel befindet sich in dem folgenden Codesegment ein Eingabefeld für die Benutzereingabe, in dem der Java-Code (scriptlet)

```
<%= loginBean.getUsername() %>
```

im HTML-Tag steht:

```
<input type="text" name="username" value="<%=
                    loginBean.getUsername() %>"/>
```

Struts benutzt hier das Konzept Custom Tag Library facility von JSP 1.1, um eigene Tag-Libraries zu definieren. Das Codeseg-

ment oben kann in Struts folgendermaßen mit Struts-Tags kodiert werden:

```
<html:text property="username"/>
```

Wie oben zu sehen ist, benötigt man durch Struts keinen Java-Code in JSP. Struts übernimmt diese Arbeit und ersetzt *username* mit der Property der ActionForms. JSP braucht nur zu wissen, wie die Property der *ActionForm* heißt. Struts bindet diese Properties von *ActionForms* mit den HTML-Komponenten, um den Inhalt von Properties zu zeigen bzw. die Benutzereingabe in diesen HTML-Komponenten in ActionForms zu speichern.

Will man mit seiner Web-Anwendung eine internationale Kundschaft erreichen, ist eine Internationalisierung unumgänglich. Struts bietet hier ein sehr einfaches Konzept, basierend auf den in Java eingebauten Internationalisierungs-Features, um mehrsprachige Web-Anwendung zu entwickeln. Es wird für das jeweilige Land eine *ApplicationResources_xx.properties*-Datei erstellt, die die Texteinträge mit den entsprechenden Keys und Values beinhaltet. Die Zeichen *xx* im Dateinamen bedeuten die Länderkürzel, für Deutschland ist der Dateiname *ApplicationResources_de.properties*. Struts ruft den zu dem Key gehörenden Wert mittels eines *message*-Tags ab. Zum Beispiel wird dieses Tag

```
<h3><bean:message key=
                    "bookingok.confirmation"/></h3>
```

mit dem Wert ersetzt, der zum *key="bookingok.confirmation"* gehört. Wenn die Variable *locale* in der JSP-Datei auf *true* gesetzt wird, wird für den jeweiligen Benutzer automatisch ein Local-Objekt aus den Client-Informationen erzeugt (Listing 4). Mittels dieses Local Object initialisiert Struts die HTML-Komponenten mit der jeweiligen Properties-Datei. Die *ApplicationResources.properties*-Datei ohne Länderabkürzung ist die Default-Datei, wenn die passende Property-Datei nicht gefunden wird. Struts unterstützt das *java.text.MessageFormat*, welches es erlaubt, Platzhalter mit dem *String {}* zu definieren, der während der Laufzeit ersetzt wird. Zum Beispiel wird dieser Eintrag

```
error.noavailableseat=<li>Es gibt max. {0} Sitztplätze
                                          zu buchen.</li>
```

in die Property-Datei eingetragen. In dem folgenden Codesegment wird *placeholder {0}* mit dem String *ex.getMessage()* während der Laufzeit ersetzt.

```
catch(BOException ex){
 errors.add( ActionErrors.GLOBAL_ERROR, new
      ActionError("error.noavailableseat",ex.getMessage()));
}
```

Das Verzeichnis der Properties-Dateien wird in der *web.xml*-Datei mit dem folgenden Eintrag spezifiziert:

```
<init-param>
   <param-name>application</param-name>
   <param-value>
com.oisoft.util.ApplicationResources</param-value>
 </init-param>
```

Was ist neu in Struts 1.0

Die neue Version Struts 1.0 enthält viele neue Features im Vergleich zur Version 0.5. Die Änderungen betreffen sowohl die Tag-Library als auch die APIs des Frameworks. Mit der Version 1.0 wurden des Weiteren viele Bugs, die sich in der Version 0.5 befanden, behoben. Im Folgenden werden die wichtigsten Features bzw. Änderungen aufgelistet. Weitere Informationen über die neuen Features sind in der Struts-Dokumentation zu finden:

- Die gesamte *struts.tld* Tag-Library der Version 0.5 ist deprecated. Einige Funktionalitäten dieser Tag-Libary wurden verbessert und in andere Tag-Libraries verteilt
- Die *struts-form.tld* Tag-Library wurde in *struts-html.tld* umbenannt.
- Die Klasse *ValidatingActionForm* ist deprecated, statt dessen wird die Methode *validate()* der Klasse *ActionForm* eingeführt.

Anzeige

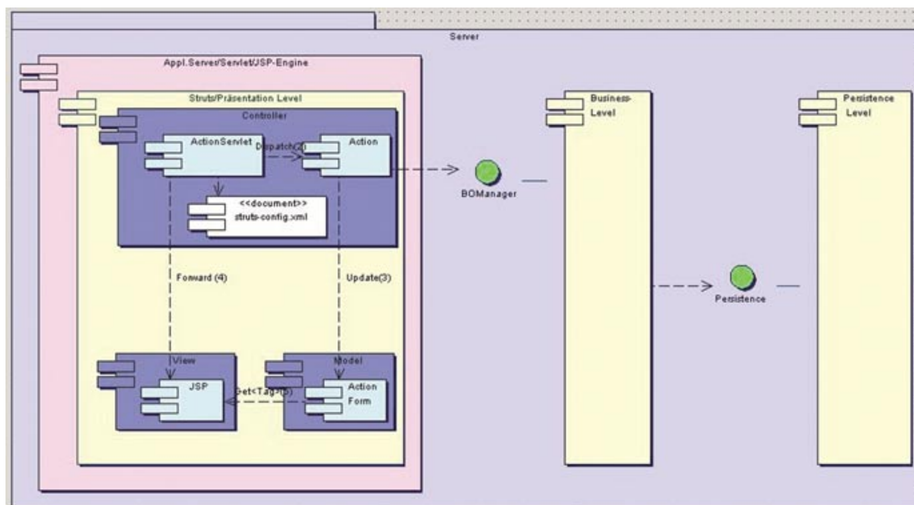

Abb. 3: Die Architektur der Anwendung „Fliegen in der Nacht"

- Mehrere Klassen im Package *org.apache.struts.util* sind deprecated. Sie werden durch das Package *beanutils* aus dem Jakarta Commons Project ersetzt. Die alten Klassen werden mit der Version 1.0 mitgeliefert, bei der Version 1.0.1 durch neue Packages ersetzt.
- Die *Action*-Klasse wird mit einem Lifecycle-Feature unterstützt, welches dafür sorgt, dass beim Erzeugen des Objekts die Methode *setServlet()* mit einem Übergabeparameter ungleich *null* aufgerufen wird. Wenn das Objekt gelöscht werden soll, wird die Methode mit dem Übergabeparameter *null* aufgerufen.
- Das Interface *ActionForm* in der Version 0.5 wurde als abstrakte Klasse definiert und mit einigen nützlichen Methoden vorimplementiert. Die Methode *validate()* dieser Klasse (ohne Parameter) ist deprecated. Diese Methode wurden durch zwei neue Methoden *validate(ActionMapping mapping, ServletRequest request)* und *validate(ActionMapping mapping, HttpServletRequest request)* ersetzt.
- Die Klasse *ActionForm* hat zwei zusätzliche Methoden *reset(ActionMapping mapping, HttpServletRequest request)* und *reset(ActionMapping mapping, ServletRequest request)* erhalten, um die Defaultwerte der Klasse *ActionForm* zu setzen.
- Die Methode

```
public ActionForward perform
  (ActionServlet servlet,
   ActionMapping mapping,
   ActionForm form,
   HttpServletRequest request,
   HttpServletResponse response)
```

der Klasse *Action* ist deprecated und wird mit der folgenden Methode ersetzt:

```
public ActionForward perform
  ( ActionMapping mapping,
   ActionForm form,
   HttpServletRequest request,
   HttpServletResponse response)
```

- Die *Action*-Klasse implementiert die Methode *getServlet()*, die den Zugriff auf das *ActionServlet* ermöglicht.
- Alle vom Controller erzeugten Objekte implementieren das Interface *java.io.Serializable*.

Neben diesen Änderungen in der Struts-API wurden die folgenden Features den Tag-Libraries zugefügt und auch etliche Änderungen in der *struts-bean*-Bibliothek vorgenommen:

- *<bean:page>* wurde hinzugefügt, um auf die Informationen aus dem *Page-Context* zuzugreifen.
- *<bean:struts>* wurde hinzugefügt, um auf die Struts-spezifischen Informationen zuzugreifen.
- *<bean:size>* erzeugt ein Bean, welches die Elemente der Klassen *Collection* oder *Map* speichert.
- Alle Tags akzeptieren ein optionales *scope*-Attribut. Im *scope* werden die Bean-Objekte gesucht. Wenn man dieses Attribut nicht spezifiziert, wird das Bean-Objekt in allen Bereichen *(page, request, session* und *application)* gesucht.

Diese Änderungen wurden in der *struts-html*-Bibliothek vorgenommen:

- Um auf *struts-html.tld* zuzugreifen, müssen Sie den Tag *<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>* in JSP hinzufügen.
- Alle JavaScript EventHandler werden gemäß XHTML-Konvention klein geschrieben *(onClick —> onclick)*.
- Alle *final*-Modifikationen der Tag-Klasse wurden gelöscht und *private*-Modifikationen zu *protected* umgewandelt, um diese Klassen ableiten zu können.
- *<html:link>* wurde so geändert, dass es Zugriff auf Context-relative URLs bietet.

Was ist neu in Struts 1.0.1

Kurz vor Fertigstellung dieses Artikels wurde Struts 1.0.1 freigegeben. Im Folgenden werden die wichtigsten Features dieser Version kurz erläutert. Neben diesen Änderungen sind manche Fehler aus Struts 1.0 behoben. Sie können die ausführlichen Dokumentationen über die neue Version unter jakarta.apache.org/struts/release-notes.html herunterladen.

- BeanUtils Package (*org.apache.commons.beanutils*) wurde durch Collections Package (*org.apache.commons.collection*) ersetzt.
- Struts 1.0.1 arbeitet mit einem XML-Parser, der JAXP/1.1 unterstützt.
- Unterstützung von UNIT-Tests für Struts-Komponenten.
- Die neuen Tags *<logic:empty>* und *<logic:notEmpty>* wurden der *struts-logic* Tag-Library hinzugefügt.
- Der Service Manager fügt spezifische Services zum ActionServlet hinzu, ohne diese Klasse abzuleiten.

Abb. 4: Erstellen eine Abfrage mit der Anwendung „Fliegen in der Nacht“

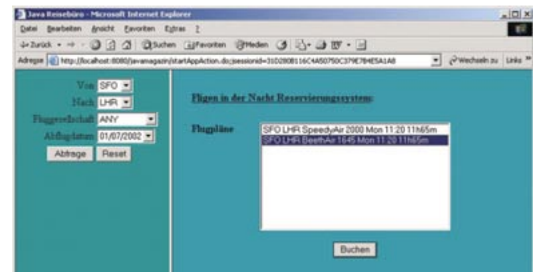

Abb. 5: Die Ergebnisse der Abfrage

- *<html:options>* unterstützt jetzt das *collection*-Attribut, welches das collection Bean-Objekt spezifiziert.
- *<html:img>* wurde hinzugefügt, um den HTML *<img>* Tag zu zeichnen.
- *<html:file>* wurde hinzugefügt, um Dateien vom Client auf den Server zu laden.

Jetzt etwas Praxis

Das folgende Beispiel beschreibt ein Projekt, das ich für meine „Sun Certified Developer for the Java 2“-Zertifizierung realisiert habe. Hierbei handelt es sich um eine Drei-Schicht-Client-Server-Anwendung für ein Flugreservierungs-System, dessen Client als Stand-Alone-Applikation (Swing) implementiert wurde. Für dieses Beispiel wurde die Präsentationsschicht durch die Web-Applikation (Browser und HTML) ersetzt und mit dem Struts-Framework realisiert. Die anderen zwei Komponenten (Geschäftslogik- und Persistenzschicht) wurden übernommen, ohne sie zu verändern. Abbildung 3 zeigt die Architektur der Anwendung.

Installation des Beispiels

Auf der beiliegenden CD wird die *flybynight.zip*-Datei mitgeliefert, die die Quellen und die *javamagazin.war*-Datei enthält. Sie können die Datei *javamagazin.war* in einem JSP 1.1- und Servlet 2.2-tauglichen Applikationsserver unter das Applikations-Verzeichnis kopieren und starten – ich selbst habe Tomcat 4 dafür eingesetzt.

Zur Weiterentwicklung bzw. zu Testzwecken werden die JSP-Dateien und die dazugehörigen Java-Quellcodes mitgeliefert. Um die Entwicklungsarbeit zu vereinfachen, wird eine *build.bat*-Datei mitgeliefert, welche das Open-Source Tool ANT aufruft. Um mit dieser Datei zu arbeiten, müssen Sie vorher ANT, jdk1.2.x. und Tomcat 4.0 installieren. Sie müssen dazu die Umgebungsvariablen JAVA_HOME und TOMCAT_HOME setzen, die auf das Installationsverzeichnis der jeweiligen Anwendung verweisen. Sie können mit dem Befehl *build deploy* in einem DOS-Shell das Projekt kompilieren. Es wird eine *javamagazin.war*-Datei erzeugt und in das Verzeichnis */webapps* vom Tomcat kopiert. Als nächstes müssen Sie den Applikationsserver neu starten und können danach mit der Adresse *localhost:8080/javamagazin* die Applikation starten (Abb. 4).

Business- und Persistenzschicht

In der Business-Komponente wird die Geschäftslogik der Anwendung bearbeitet, in der Persistenz-Komponente werden die Daten statt in eine Datenbank in eine Datei *(db.db)* gespeichert. Diese beiden fertigen Komponenten werden in diesem Projekt eingesetzt, ohne diese neu zu kompilieren (dank der komponentenbasierten Architektur). Da diese beiden Komponenten von mir für die Prüfung „Sun Certified Developer for the Java 2“ entwickelt wurden, werden die Quellcodes der beiden Komponenten allerdings nicht mitgeliefert.

Präsentationsschicht der Web-Anwendung „Fliegen in der Nacht“

Wenn Sie die Anwendung gestartet haben, können Sie Ihre Auswahl mit den *ComboBoxes* festlegen (siehe Abb. 4). Nach dem Anklicken des Buttons ABFRAGE werden die Ergebnisse dieser Abfrage auf der rechten Seite der Anwendung angezeigt. Wenn Sie nun einen Flug auswählen und den Button BUCHEN anklicken, wird eine Maske zur Eingabe weiterer Daten angezeigt (siehe Abb. 6). Nachdem Sie die Eingabefelder ausgefüllt und den Button SENDEN angeklickt haben, wird die Flugbestätigung mit Ihrer Buchungsnummer angezeigt (Abb. 7). Abbildungen 7 und 8 zeigen außerdem das Internationalisierungsfeature des Struts-Frameworks: die Anwendung wird mit einem englischsprachigen Browser ausgeführt. Das Framework entscheidet sich je nach Browser, welche *ApplicationResources_xx.properties*-Datei

Anzeige

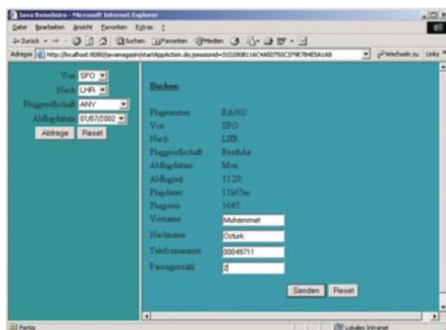

Abb. 6: Maske für die Flugbuchung

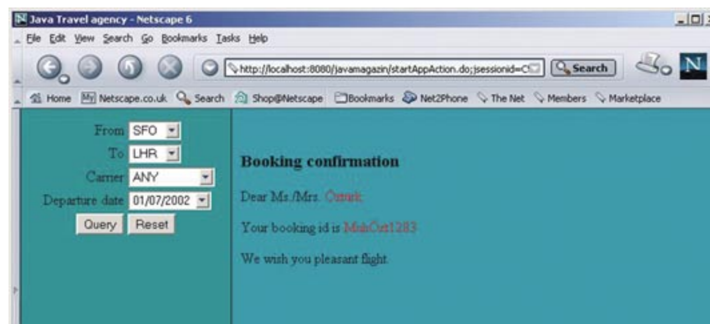

Abb. 7: Die Bestätigung der Buchung in einem englisch-sprachigenBrowser

zu verwenden ist. Man hätte die Sprache natürlich auch durch eine explizite Auswahl beim Start festlegen können.

Abbildung 8 zeigt im Übrigen einen Fehler bei einer fehlenden Eingabe. Die Überprüfung der Eingabe erfolgt in der Methode *validate()* der Klasse *FormModel* (Listing 1). Sie überprüft die Benutzereingabe und im Falle eines Fehlers wird ein Objekt der Klasse *ActionError* mit dem jeweiligen Text erzeugt. Wie im Code zu sehen ist, wird bei dem Erzeugen des Objekts *new ActionError("error.customerfistname")* der Key der Fehlermeldung in der *ApplicationResources_xx.properties*-Datei übergeben. Das Struts-Framework ersetzt den Key mit dem jeweiligem Wert aus der *ApplicationResources_xx.properties*-Datei. In Listing 2 sind die Textabschnitte der jeweiligen Property-Datei zu sehen. Sollte während der Ausführung eine Fehlersituationen auftreten, wird das erzeugte *ActionError*-Objekt in einem *ActionErrors* Containerobjekt gespeichert. Die gespeicherten Objekte werde dann durch das *<html:errors/>* Tag der Struts Tag Library auf der Seite angezeigt. In Listing 3 ist der JSP-Quellcode der Form auf der rechten Seite der Abbildung 8 dargestellt.

Fazit

Das Struts-Framework setzt das MVC-Paradigma für Web-Anwendungen durch die Benutzung von Tags in JSP auf äußerst elegante Weise um. Durch die Trennung der Komponenten erreicht man eine hohe Wiederverwendbarkeit dieser Komponenten. Außerdem werden die Arbeitsbereiche von HTML-Designern und Java-Entwicklern klar getrennt, sodass sich jeder auf sein Spezialgebiet konzentrieren kann.

Natürlich macht Struts die JSP-Programmierung durch die Einführung eigener Tags etwas aufwändiger. Das wäre für einfache Web-Anwendungen nicht nötig, für komplexe Web-Anwendungen ist Struts jedoch ein geeignetes Mittel, um die Komplexität zu verwalten und zu bewältigen. Um die Arbeit mit Struts zu vereinfachen, ist der Einsatz der Struts Console 1.3 (auf der beiliegenden CD oder unter www.jamesholmes.com/struts/console zu beziehen) empfehlenswert, mit der sich JSP-Dateien auf einfache Weise in Struts HTML-Tags konvertieren lassen. Außerdem kann man mit dem Tool die *struts-config.xml*-Dateien editieren.

Sicherlich kann Struts nicht alle Wünsche erfüllen. Wie bei allen Open-Source-Tools wird das Framework in Zukunft weiter entwickelt. Ich persönlich würde kein eigenes MVC-Framework für die Web-Anwendungen entwickeln, solange ich Struts einsetzen kann. Wer sich aus irgend einem Grunde nicht für den Einsatz von Struts entscheiden will, kann dennoch einige nützliche Tipps und Anregungen für die Umsetzung eines MVC-Konzepts aus der Kombination von JSP und Servlet-Technologie für Web-Anwendungen mitnehmen.

Dipl. Ing (M.E) Muhammet Öztürk ist Projektmanager bei der Object International Software GmbH und hat mehrere Jahre Erfahrung als Softwareentwickler und Projektleiter in verschiedenen Softwareprojekten. Er ist Sun zertifizierter Developer.

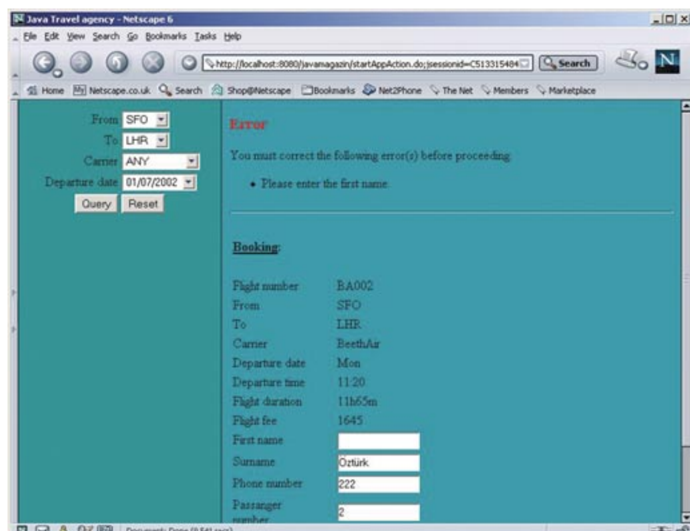

Abb. 8: Fehler bei fehlender Benutzereingabe

Literatur & Links:

[1] Jakarta-Tomcat 4.0: jakarta.apache.org/tomcat/

[2] Struts 4.0: jakarta.apache.org/builds/jakarta-struts/release/v1.0/

[3] Ant: jakarta.apache.org/jakarta-ant-1.4.1/

[4] Struts Console 1.3: www.jamesholmes.com/struts/console

[5] Design Patterns, Erich Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1997

[6] The Rational Unified Process, Philippe Kruchten, Addison-Wesley, 1998

Listing 1: Die Klasse *FormModel*

```
public final class FormModel extends ActionForm {
 private String selectedOriginName;
 private String selectedDestinationName;
 ...
  public void setSelectedOriginName(String newSelectedOriginName) {
   selectedOriginName = newSelectedOriginName;
  }
  public String getSelectedOriginName() {
   return selectedOriginName;
  }
  public void setSelectedDestinationName(String newSelectedDestinationName) {
   selectedDestinationName = newSelectedDestinationName;
  }
  public String getSelectedDestinationName() {
   return selectedDestinationName;
  }
  ...

  public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.selectedOriginName=null;
    this.selectedDestinationName=null;
  }

  public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {

   ActionErrors errors = new ActionErrors();

   if(mapping.getPath().equals(„/bookingSubmitAction“)){
    if ((customerFistName == null) || (customerFistName.length() < 1))
     errors.add(“customerFistName“, new ActionError(“error.customerfistname“));
    ...
    try{
     Integer.parseInt(bookedSeats);
    }catch(Exception e){
     errors.add(“bookedSeats“, new ActionError(“error.bookedseats“));
    }
   }
   return (errors);
  }
}
```

Listing 2: Abschnitt aus den beiden *ApplicationResources.properties*-Dateien

```
//for german ApplicationResources_de.properties
errors.header=<h3><font color=“red“>Fehler</font></h3>Sie müssen die folgenden Fehler vor dem Ausführen korrigieren:<ul>
errors.footer=</ul><hr>
error.noavailableseat=<li>Es gibt max. {0} Sitztplätze zu buchen.</li>
...

//default ApplicationResources.properties
errors.header=<h3><font color=“red“>Error</font></h3>You must correct the following error(s) before proceeding:<ul>
errors.footer=</ul><hr>
error.noavailableseat=<li>There is max. {0} seats to book.</li>
...
```

Anzeige

Listing 3: Auszug aus der *Bookingjsp*-Datei

```
<%@ page language="java" contentType="text/html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<html:html locale="true">
<head>
<title><bean:message key="index.title"/></title>
<html:base/>
</head>

<html:form action="bookingSubmitAction.do" target="mainFrame">
<html:errors/>
<br>
<table border="0" cellspacing="3" cellpadding="1">
 <tr valign="top" align="left">

  <td colspan="3"><u><b><bean:message key="listflightplan.booking "/>
                                                    </b></u><b>:</b></td>
 </tr>
 <tr valign="top" align="left"><td> </td></tr>
 <tr>
  <td valign="top"><bean:message key="booking.flightnumber"/></td>
  <td width="20"> </td>
  <td colspan="3"><bean:write name="formModel" property="selectedFlight.
                                                    flightNumber"/></td>
 </tr>

 ...
  <td align="right">
   <input type="submit" value="<bean:message key='booking.submit'/>">
  </td>
  <td align="left">
   <html:reset/>
  </td>
 </tr>
</table>
...
```

Listing 4: Die Methode *BookingSubmitAction*

```
public final class BookingSubmitAction extends Action{
 public ActionForward perform(ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response)
                              throws IOException, ServletException {

  FormModel formModel =(FormModel)form;
  HttpSession httpSession = request.getSession();
                 ...
  try{
   Booking booking = new Booking(formModel.getSelectedFlight().getFlightNumber(),
   formModel.getSelectedDepartureDate(),
   formModel.getCustomerFistName(),
   formModel.getCustomerLastName(),
   formModel.getCustomerPhoneNumber(),
   Integer.parseInt(formModel.getBookedSeats()));
   Flight flight = new Flight( formModel.getSelectedFlight().getFlightNumber(),
   formModel.getSelectedDepartureDate(),
   Integer.parseInt(formModel.getBookedSeats()),
   formModel.getSelectedFlight().getMaximumSeats());
   //books the flight
   String bookingId = BOManager.getInstance(true,url).bookFlight(booking,flight);
   formModel.setBookingId(bookingId);
  }catch(BOException ex){
   ActionErrors errors = new ActionErrors();
   errors.add( ActionErrors.GLOBAL_ERROR,
   new ActionError("error.noavailableseat",ex.getMessage()));
   saveErrors( request, errors );
   //return errors;
   return( new ActionForward(mapping.getInput()));
  }
  return (mapping.findForward("success"));
 }
}
```

Listing 5: Auszug aus der *struts-config.xml*-Datei

```
<struts-config>

 <!-- ========== Form Bean Definitions ========== -->
 <form-beans>

  <form-bean    name="formModel"
   type="com.oisoft.view.form.FormModel"/>
 </form-beans>


 <!-- ========== Global Forward Definitions ========== -->
 <global-forwards>
  <forward name="main"  path="/MainFrame.jsp"/>
 </global-forwards>

 <!-- ========== Action Mapping Definitions ========== -->
 <action-mappings>
...
  <action  path="/bookingSubmitAction"
   type="com.oisoft.view.action.BookingSubmitAction"
   name="formModel"
   scope="session"
   input="/Booking.jsp">
   <forward name="success" path="/BookingOk.jsp"/>
  </action>
 </action-mappings>
</struts-config>
```