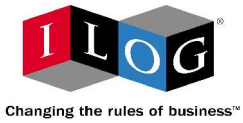


Projekt: *Gastbeitrag für JSP-develop.de*
Thema: *Implementierung einer grafischen JSF-Komponente*
Länge: *ca. 29.000*
Red.-Schluss: *-*
Autor: *Patrick Mégard, ILOG*

Implementierung einer grafischen JavaServer Faces Komponente

Die JavaServer Faces (JSF)-Technologie bringt bei der Entwicklung interaktiver Web-Applikationen im Vergleich zu anderen Technologien wie JavaServer Pagers oder Apache Struts viele Vorteile. JSF trennt die Anwendungslogik klar von der GUI-Präsentation, verbessert die Wartung von Web-Applikationen und bietet ein Framework für die Entwicklung und Wiederverwendung von UI-Komponenten.
Viele Web-Anwendungsentwickler migrieren zu JSF, um festzustellen, dass die vordefinierten JSF UI-Komponenten auf DHTML-Widgets beschränkt sind. Komplexere Anwendungen wie Überwachungs- oder Business Process Monitoring-Lösungen hingegen benötigen ausgefeilte Visualisierungskomponenten, die mit dem JSF-Framework kompatibel sind.
Die Standardisierung durch das JSF-Framework vereinfacht die Entwicklung individueller GUI-Komponenten, die Anwendungsentwickler wiederverwenden können. Anbieter von Web-Komponenten können jetzt ausgefeiltere Komponenten anbieten und Web-Anwendungsentwickler können sich sicher sein, dass sie diese leicht verwenden können. Solche JSF UI-Komponenten müssen sauber in das JSF Laufzeit-Framework integrieren und ebenso zuverlässig auch in die IDE, die JSF-Support anbietet, eingebunden werden können.
Obwohl JSF ein Standard-Framework mitbringt, gibt es einige Stolpersteine für Entwickler, die ihre erste eigene JSF-Komponenten entwickeln. Dieser Artikel beschreibt, wie eine graphische JSF-Komponente erstellt werden kann, die nur auf HTML-Basis nicht einfach zu implementieren ist. Die Charakteristiken einer grafischen JSF-Komponente erfordern nicht nur die Generierung von DHTML, sondern die zusätzliche Unterstützung für Bildgenerierung und Client-seitige Interaktionen.
Als Beispiel dient eine Chart-Komponente, die grafische Charts darstellen soll und verschiedene Client-seitige Navigations- und Interaktions-Möglichkeiten bietet. Außerdem wird gezeigt, wie die Chart-Komponente in eine JSF-fähige IDE integriert werden kann.

1. Zur Erinnerung - JavaServer Faces

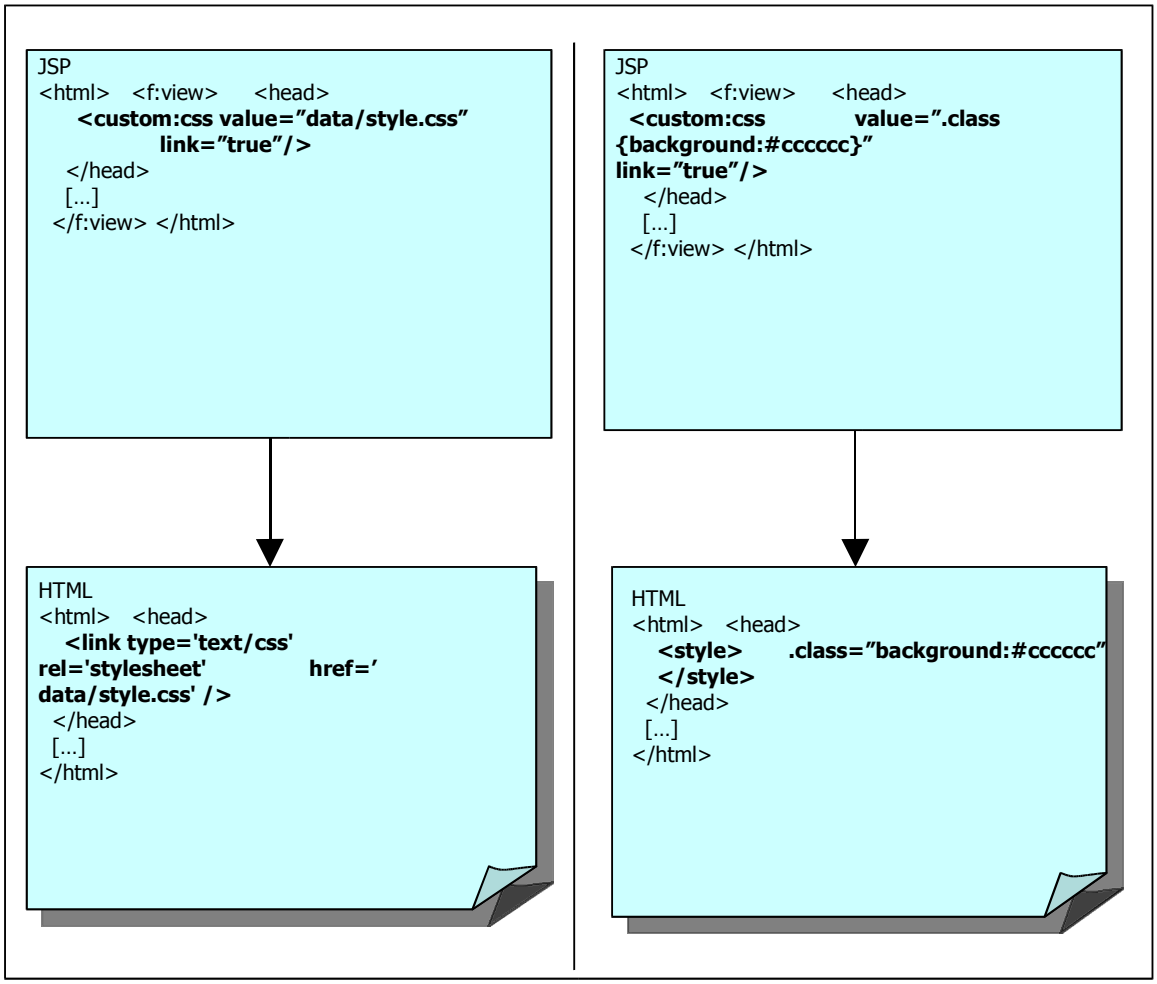
JavaServer Faces (JSF) ist ein Standard-Server-Side-Framework, das die Konstruktion des Präsentationslayers von Web-Applikationen vereinfacht. Entwickler können wiederverwendbare UI-Komponenten zusammenstellen, um Webseiten zu generieren, diese Komponenten mit Datenquellen verknüpfen und Client-seitige Ereignisse mit Server-seitigen Event-Handlern verarbeiten. Der Spezifikation folgend, können Anbieter Komponenten entwickeln, die sauber in das JSF-Laufzeit-Framework und in IDEs integrieren, so dass diese schon bei der Entwickung JSF-kompatibel sind.
JSR 127[(1)] liefert eine Referenzimplementation mit, die das JSF-Framework definiert. Diese umfasst einfache UI-Komponenten wie Eingabefelder und Buttons. In aller Regel korrespondieren diese JSF-Komponenten direkt mit den HTML-Komponenten und Tags aus der HTML 1.4 Spezifikation. Für viele Webanwendungen ist das ausreichend. Andere Anwendungen, zum Beispiel solche zur Überwachung von Prozessen, benötigen komplexere Funktionen zur Darstellung von Daten und zur Interaktion, wie zum Beispiel Charts, Diagramme oder Geoinformationen. Das Design solcher komplexeren Komponenten ist nicht trivial, da die Möglichkeiten, komplexere Grafik-Widgets direkt in HTML wiederzugeben, begrenzt sind. Das Problem lässt sich lösen, indem die Server-seitige Komponente die Bilder an den Client liefert. Das bringt jedoch neue Probleme mit sich, denn die Interaktionsfähigkeit mit einfachen HTML-Bildern ist begrenzt. Um dem Nutzer Navigation und Interaktion mit den Daten zu ermöglichen, muss zusätzlich JavaScript eingesetzt werden.

Erstellen einer einfachen JSF-Komponente

Im folgenden Abschnitt werden die nötigen Schritte beschrieben, um eine sehr einfache JSF-Komponente zu entwickeln, die CSS in eine HTML-Seite importiert. Die Beschreibung und die Code-Beispiele in den nächsten Abschnitten dienen dann als Grundlage für die Entwicklung einer aufwändigeren JSF Charting-Komponente.
Die folgenden Abbildungen zeigen, wie die zu entwickelnde Komponente verwendet wird, und wie das Ergebnis aussehen wird.

Changing the rules of business™

Projekt: *Gastbeitrag für JSP-develop.de*
Thema: *Implementierung einer grafischen JSF-Komponente*
Länge: *ca. 29.000*
Red.-Schluss: *-*
Autor: *Patrick Mégard, ILOG*

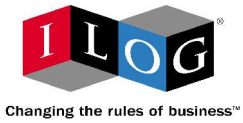


Der Vorteil einer solchen Komponente ist, dass das gesamte Aussehen der Seite mit einem anderen Wert in der Komponente durch die JSF-Aktion verändert werden kann.
Eine JSF-Komponente besteht aus mehreren Java-Klassen und Konfigurationsdateien. Um eine eigene JSF-Komponente zu generieren, sind folgende Schritte nötig:

- Schritt 1: Entwickeln einer Java-Klasse, die eine der Basis-Komponentenklassen von JSF erweitert.
- Schritt 2: Entwickeln eines Renderers für das „default render kit“ der Referenzimplementation.
- Schritt 3: Entwickeln einer Java-Klasse, die den Tag in der JSP-Seite beschreibt.
- Schritt 4: Schreiben einer Tag Library Definition Datei.
- Schritt 5: Schreiben der JSF Configuration Datei.

Die Einzelschritte werden jetzt erklärt.

Schritt 1: Die Java-Komponenten-Klasse
Die Komponenten-Klasse ist zuständig für die Verwaltung der Properties, die den Status der Komponente repräsentieren. Abhängig vom Verhalten der Komponente, wie zum Beispiel Input oder Output, muss eine passende Basis-Klasse gewählt werden.



Projekt: *Gastbeitrag für JSP-develop.de*
Thema: *Implementierung einer grafischen JSF-Komponente*
Länge: *ca. 29.000*
Red.-Schluss: *-*
Autor: *Patrick Mégard, ILOG*

```
import javax.faces.component.*;
public class CSSComponent extends UIOutput {
 private Boolean link;
 public String getFamily() {
   return "faces.CSSFamily";
 }
 public boolean isLink() {
   if (link != null)
       return link.booleanValue();
     ValueBinding vb = getValueBinding("link");
   if (vb != null) {
     Boolean bvb = (Boolean) vb.getValue(FacesContext.getCurrentInstance());
     if (bvb != null)
         return bvb.booleanValue();
   }
   return false;
 }
 public void setLink(boolean link) {
   this.link = new Boolean(link);
 }
 public Object saveState(FacesContext context) {
   return new Object[] { super.saveState(context), link };
 }
 public void restoreState(FacesContext context,
                          Object stateObj) {
   Object[] state = (Object[]) stateObj;
   super.restoreState(context, state[0]);
   link = (Boolean) state[1];
 }
}
```

Die hier beschriebene Komponente erweitert javax.faces.component.UIOutput, um eine URL anzuzeigen, die auf eine Style Sheet Datei oder den Inhalt eines Inline Style Sheets verweist. Die Komponente kann zum Wechseln von einem Style Sheet zum anderen innerhalb einer JSF-Aktion verwendet werden. Die 'link' Property spezifiziert den Typ des Werts: entweder eine URL oder der Inline Style. Die Komponente muss zudem in der Lage sein, mit Hilfe eines Objektes, das durch das JSF-Framework verarbeitet wird, ihren Status zwischen Anfragen zu speichern und wieder herzustellen. Der Status der Komponente besteht aus einer Reihe von Property-Werten, die für die Wiederherstellung des Objekts nötig sind. Das JSF-Framework ruft automatisch die saveState und restoreState Methoden auf, die in der Komponente implementiert wurden.

Schritt 2: Der Renderer

Der Renderer hat zwei Funktionen. Erstens ist er dafür zuständig, ein passendes HTML-Fragment auszugeben, das die Komponente beim Client erzeugt. Üblicherweise besteht dieses HTML-Fragment aus einigen HTML-Tags, die für das Rendern in den gebräuchlichen Webbrowsern geeignet sind. Diese Phase des JSF-Lifecycle nennt sich Encoding oder Render Response Phase. Diese Rendering-Phase kann auch dazu verwendet werden JavaScript-Code auszugeben, der der Verbesserung der Client-seitigen Interaktion dient. Die zweite Aufgabe des Renderers ist die Dekodierung der Daten, die vom Client kommen und zur Aktualisierung des Server-seitigen Komponentenstatus verwendet werden, so zum Beispiel Text, den der Nutzer in ein Textfeld eingibt. Das Standard-Renderer-Kit ist obligatorisch, doch andere können eingesetzt werden, um eine andere Client-seitige Repräsentation oder Sprache wie SVG[(2)] zu nutzen.

```
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;
public class CSSRenderer extends Renderer {
public void encodeEnd(FacesContext context,
                      UIComponent component)
  throws IOException {
    super.encodeEnd(context, component);
    if (component instanceof CSSComponent) {
      CSSComponent cssComponent  =
        (CSSComponent) component;
      String css = (String)cssComponent.getValue();
      boolean isLink = cssComponent.isLink();
      if (css != null)
        if (isLink)
          context.getResponseWriter().write("<link type='text/css' rel='stylesheet' href='" + css + "'/>");
        else
          context.getResponseWriter().write("<style>\n" + css + "\n<style/>\n");
    }
  }
}
```

Der hier eingesetzte Renderer wählt den CSS-Typ, der in der HTML-Seite ausgegeben werden soll, indem er den Property-Link der Komponente überprüft.

Schritt 3: Die Tag-Klasse

Auch hier bietet das JSF-Framework Basis-Klassen, die erweitert werden können, um den Tag zu schreiben, der mit der Komponente verbunden werden soll. Die Tag-Klasse ist verantwortlich für:

- Die in der faces-config.xml-Datei verwendete Definition des Komponenten- und Rendering-Typs der Komponente. Sie wird im nächsten Abschnitt beschrieben.
- Das Erzeugen der JSF-Komponente (die vom JSF-Framework gehandhabt wird) und deren übergebene Attribute, die im JSF-Tag zur Initialisierung der Komponente enthalten sind.

```
import javax.faces.webapp.UIComponentTag;
public class CSSTag
  extends UIComponentTag {
  private String value;
  private String link;
  public String getComponentType() {
    return "faces.CSSComponent";
  }
  public String getRendererType() {
    return "HTML.LinkOrInlineRenderer";
  }
  protected void setProperties(UIComponent component) {
    super.setProperties(component);
    Application app = getFacesContext().getApplication();
    if (value != null)
        if (isValueReference(value))
         component.setValueBinding("value",
              app.createValueBinding(value));
        else
          component.getAttributes().put("value", value);
    if (link != null)
       if (isValueReference(link))
         component.setValueBinding("link",
             app.createValueBinding(link));
        else
          component.getAttributes().put("link",
             new Boolean(link));
  }
  public String getLink() {
    return link;
  }
  public void setLink(String link) {
    this.link = link;
  }
 public String getValue() {
    return value;
  }
  public void setValue(String value) {
    this.value = value;
  }
}
```

Das Tag stellt Get- und Set-Methoden bereit, um den Link und die Attribute zu verwalten. Wenn die Komponente erzeugt wird, wird die setProperties Methode aufgerufen, um ihre Properties aus den Tag-Attributen zu initialisieren. Jedes Tag-Attribut kann entweder ein expliziter Wert sein oder über eine Bean-Property definiert werden.

Schritt 4: Die Tag Library Definition (TLD)

Die TLD ist eine XML-Datei, die den Tag durch Assoziation des Namens mit der entsprechenden Java-Klasse beschreibt. Die TLD legt außerdem die zulässigen Attribute des Tags fest.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library  1.2//EN" "http://java.sun.com/dtd/web-
jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>custom</short-name>
  <uri>http://www.ilog.com/jviews/tlds/css.tld</uri>
  <description>This tag library contains a tag for a sample custom JSF Component.</description>
  <tag>
    <name>css</name>
    <tag-class>path.to.CSSTag</tag-class>
    <description>A component that displays the style inline or a link a to a css file</description>
    <attribute>
      <name>id</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
      <type>java.lang.String</type>
      <description>The id of this component.</description>
    </attribute>
    <attribute>
      <name>binding</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
      <type>java.lang.String</type>
      <description>The value binding expression linking this component to a
        property in a backing bean. If this attribute is set, the tag does not
        create the component itself but retrieves it from the bean property.
        This attribute must be a value binding.</description>
    </attribute>
    <attribute>
      <name>value</name>
      <required>true</required>
      <rtexprvalue>false</rtexprvalue>
      <type>java.lang.String</type>
      <description>The inline css text or the url to the css file to link.</description>
    </attribute>
    <attribute>
      <name>link</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
      <type>java.lang.String</type>
      <description>Whether the value is a link or the inline style.</description>
    </attribute>
  </tag>
</taglib>
```

Diese TLD definiert einen Tag namens ‘css’, der an die CSSTag Klasse gebunden ist. Sie beschreibt zudem Link und die Value Tag Attribute.

Schritt 5: Die JSF Configuration Datei

Um eine JSF-Komponente in das Framework zu integrieren, muss eine Konfigurationsdatei namens faces-config.xml bereitgestellt werden. Diese Datei verbindet Komponenten- und Renderer-Typen, die im JSP Custom Tag Handler verwendet werden, mit der entsprechenden Java-Klasse. Sie beschreibt zudem den Renderer, der mit jeder Komponente verwendet werden sollte.

Changing the rules of business™

Projekt: *Gastbeitrag für JSP-develop.de*
Thema: *Implementierung einer grafischen JSF-Komponente*
Länge: *ca. 29.000*
Red.-Schluss: *-*
Autor: *Patrick Mégard, ILOG*

```
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
                              "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
  <component>
    <component-type>faces.CSSComponent</component-type>
    <component-class>path.to.CSSComponent</component-class>
    <component-extension>
      <component-family>faces.CSSFamily</component-family>
      <renderer-type>HTML.LinkOrInlineRenderer</renderer-type>
    </component-extension>
  </component>
  <render-kit>
    <renderer>
      <component-family>faces.CSSFamily</component-family>
      <renderer-type> HTML.LinkOrInlineRenderer </renderer-type>
      <renderer-class>path.to.CSSRenderer</renderer-class>
    </renderer>
  </render-kit>
</faces-config>
```

Diese Datei definiert die Komponentenfamilie faces.CSSFamily. In diesem Beispiel besteht die Familie aus einer einzelnen Komponente vom Typ faces.CSSComponent. Dieser Typ ist an die CSSComponent-Klasse gebunden. Der Renderer des Typs HTML.LinkOrInlineRenderer, der durch die CSSRenderer-Klasse implementiert ist, ist mit der faces.CSSFamily Familie verbunden.
Wenn die Komponente in eine JSF-fähige IDE integriert werden soll, können zusätzliche Informationen bereitgestellt werden. Im Falle der Sun Creator IDE muss eine XML-Konfigurationsdatei mit der Bezeichnung sun-faces-config.xml erstellt werden. Die Datei beschreibt die Properties der Komponente, die in der IDE dargestellt werden sollen sowie weitere Design-Time Informationen. Mehr Informationen zu Sun Creator sind in der Produktbeschreibung auf der Sun Website zu finden. [(3)]
Nachdem nun die Schritte zur Erstellung einer einfachen JSF-Komponente klar sind, folgt jetzt ein Überlick, wie eine grafische JSF-Komponente erstellt werden kann. Weitere Informationen zum Erstellen einer JSF-Komponente finden sich im J2EE-Tutorial.

2. Erstellen einer grafischen JSF-Komponente

Die Arbeitsschritte zur Erstellung einer speziellen JSF-Grafikkomponente entsprechen der bereits erläuterten Abfolge. Als Beispiel dient hier eine Charting-Komponente, wie die ILOG JSF Charting-Komponente. Sie stellt eine visuelle Darstellung der Verteilung von Datenwerten über eine Reihe von Kategorien bereit. Das Chart kann Datensätze in einer Vielzahl von Formen abbilden, zum Beispiel Balken und Kuchendiagramme oder Bubble-Charts usw.
Die JSF-Charting-Komponente hat zwei grundsätzliche Designbeschränkungen:

1. Es existiert bereits eine Java Charting Bean-Komponente, die alle benötigten grafischen Präsentationsfunktionen enthält. Diese Komponente kann viele Charttypen anzeigen und ist vielfältig anpassbar. Idealerweise werden die Fähigkeiten dieser Bean-Komponente als Grundlage der JSF-Komponente genutzt.
2. Für gewöhnlich müssen JSF-Anwendungen eine Seite vollständig neu laden, um die Darstellung zu aktualisieren. Dieses Verhalten kann für formularbasierte Anwendungen passend sein, ist aber bei den meisten grafischen Benutzerinterfaces nicht zweckmäßig. Damit die JSF-Charting-Komponente benutzerfreundlich ist, muss sie einfache Navigations- und Interaktionsfunktionen ermöglichen, ohne dass die gesamte Seite neu geladen wird.

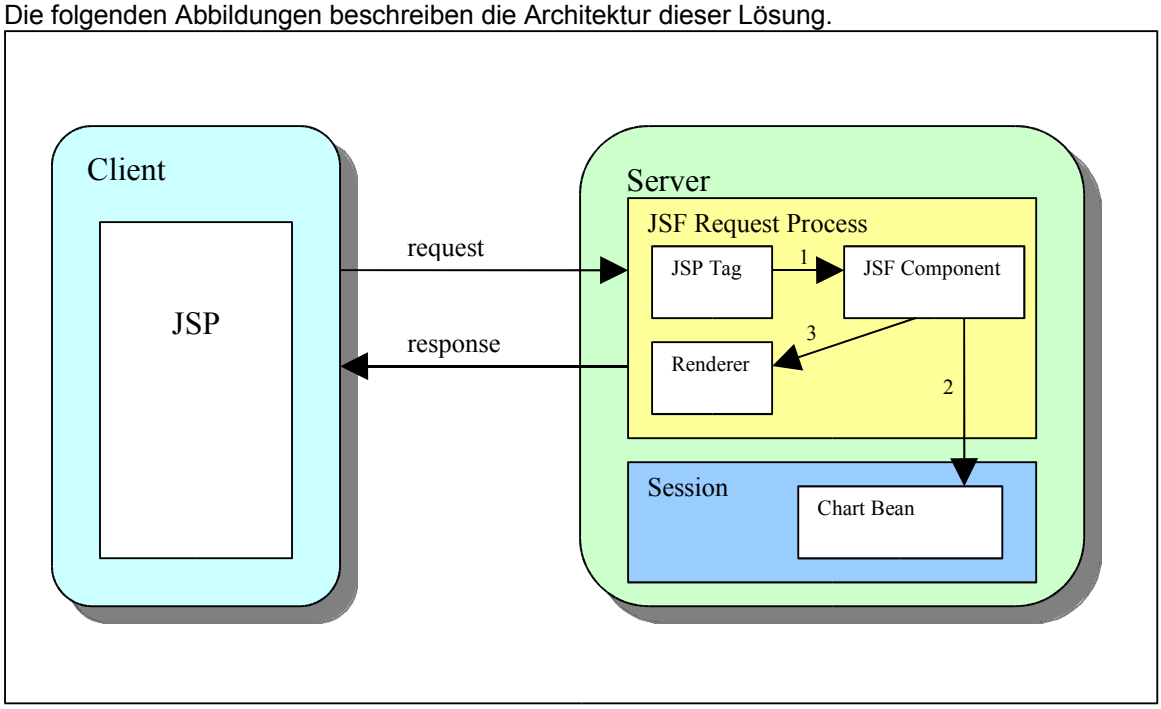
Diese Voraussetzungen werden wie folgt geschaffen:

- Die JSF-Charting-Komponente wird die Chart-Bean-Komponente managen. Zu diesem Zweck wird die Chart-Bean erstellt, die Bean modifiziert und für Server-seitige Aktionen verfügbar gemacht.
- Das Rendering der JSF-Komponente wird in zwei Phasen vorgenommen:
  - Der JSF-Renderer erzeugt einen <img> Tag und einen Satz JavaScript-Objekte (Abbildung 1).

- Der Client fragt ein Bild vom Server ab. Der Aufruf wird von einem Servlet übernommen, das die Chart-Bean abruft und mittels der vom Chart bereit gestellten Methoden ein Bild erzeugt (Abbildung 2).

- Jede weitere Benutzerinteraktion (Zoomen, Panning, Wechsel eines Stylesheets, usw.), die nur das Chart-Bild verändert, wird mit einer inkrementellen Aktualisierung des Chart-Bildes umgesetzt. Nur wenn die Client-seitige Aktion mehr als ein Update des Chart-Bildes erfordert, wird wieder die gesamte Seite abgeschickt (Abbildung 3).

Die folgenden Abbildungen beschreiben die Architektur dieser Lösung.



1. Create and initialize properties.
2. Create and customize the chart and make it available for server-side actions.
3. Render the chart.

Abbildung 1: Anfrage an das JSF-Framework

Changing the rules of business™

Projekt: *Gastbeitrag für JSP-develop.de*
Thema: *Implementierung einer grafischen JSF-Komponente*
Länge: *ca. 29.000*
Red.-Schluss: *-*
Autor: *Patrick Mégard, ILOG*

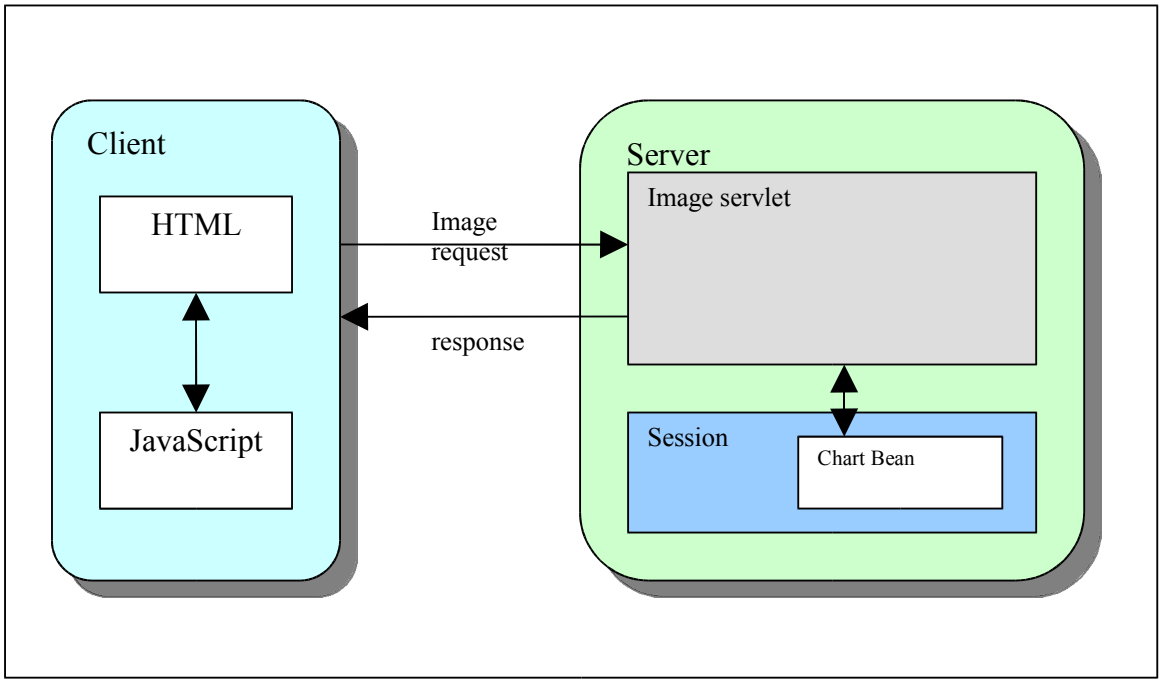


Abbildung 2: Anfrage an das Servlet zur Anfrage eines Bildes.

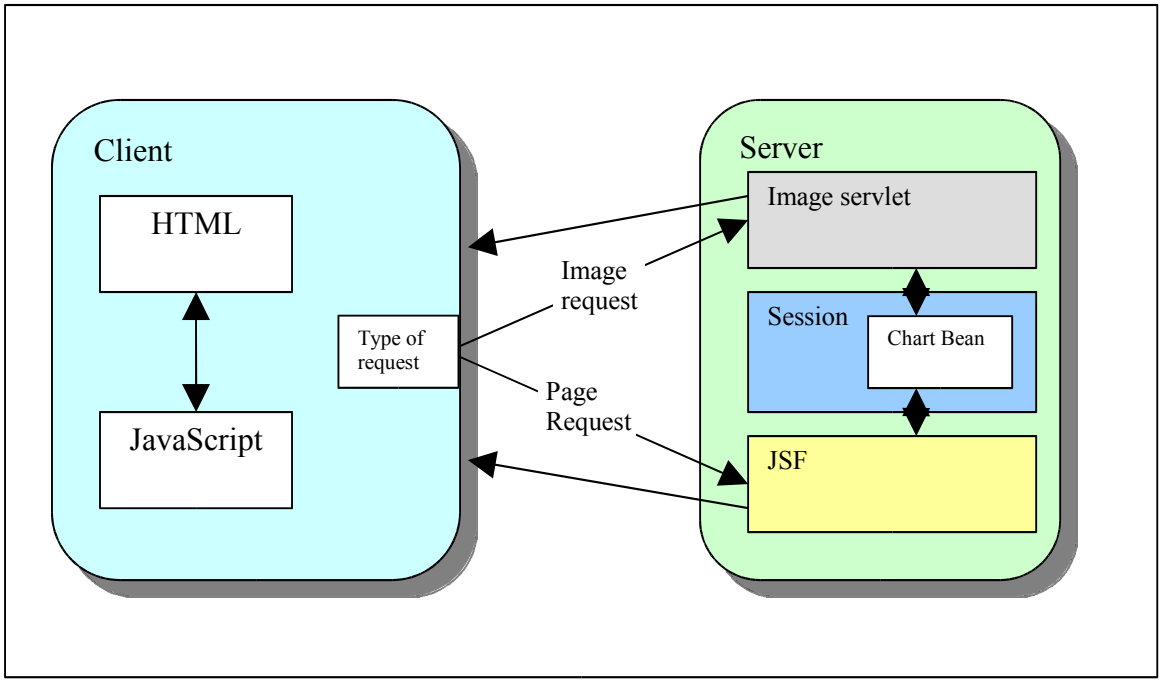
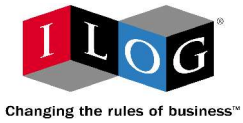


Abbildung 3: Entweder eine Seiten- oder eine Bild-Anfrage.

Die JSF-Charting-Komponente kommt mit einem Satz zusätzlicher JSF-Komponenten:



Projekt: *Gastbeitrag für JSP-develop.de*
Thema: *Implementierung einer grafischen JSF-Komponente*
Länge: *ca. 29.000*
Red.-Schluss: *-*
Autor: *Patrick Mégard, ILOG*

- Überblick: Der Überblick stellt die Gesamtansicht des Charts und ein Rechteck zur Repräsentation der eigentlichen Chart-Ansicht dar. Diese Komponente ermöglicht dem Nutzer außerdem das Bewegen innerhalb des sichtbaren Bereichs.
- Legende: Die Legenden-Komponente zeigt Informationen über die abgebildeten Daten an. Die Legende kann auch im Chart selbst abgebildet werden, abhängig vom Abbildungsstil der dargestellten Daten.
- Interaktionen: Ebenfalls mitgeliefert werden Client-seitige Interaktionen, die zum Beispiel für das Zoomen oder Panning innerhalb des Chart-Bildes genutzt werden. Diese Interaktionen können als Client-seitige Interaktionen begriffen werden, das heißt, dass eine Interaktion mit dem Chart keine Aktualisierung der gesamten Seite auslöst, wie es bei normalen JSF-Interaktionen der Fall wäre.

Um die Chart-Komponente zu rendern reicht es aus, das ChartView-Tag zu verwenden:

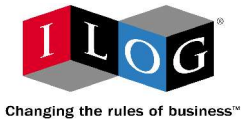
```
<jvcf:chartView id="c" style="width:500px;height:300px" ... />
```

Die Daten werden als Bild auf der HMTL-Seite dargestellt. Das Bild wird durch ein Servlet generiert, das auf einen http-Request antwortet. Der Request enthält verschiedene Parameter, die das Bild spezifizieren, wie eine Imagemap zu erzeugen ist, die eine inline Legende spezifizieren und so weiter. Das resultierende Bild wird dann in die Client-seitige DOM eingefügt und der einzige Teil, der aktualisiert wird, ist das Bild selbst.
Im Folgenden werden einige Unterschiede zwischen einer einfachen angepassten JSF-Komponente und der ausgefeilteren Charting-Komponente dargestellt.

Komponente
Die JSF-Chartkomponenten-Klasse ist einer Standardkomponente sehr ähnlich. Sie enthält zusätzlich eine Chart-Property, die den Zugang zur Chart-Bean freigibt, die wiederum für die Erzeugung des in der HMTL-Seite dargestellten Bildes zuständig ist. Die JSF-Komponente kann diese Chart-Bean durch Value Binding oder in der laufenden Session lokal aufrufen. Wenn die JSF-Chartingkomponente das zentrale Element einer Applikation ist, können optionale JSF-Komponenten, wie etwa ein Überblick oder eine Legende mit dem Haupt-Chart verbunden werden, um zusätzliche Informationen darzustellen.

```
<jvcf:chartZoomInteractor id="chartZoomInteractor"
                          XZoomAllowed="true"
                          YZoomAllowed="true" />
<jvcf:chartView id="chartView"
                chart="#{myBean.chart}"
                servlet="demo.ImageMapServlet"
                interactorId="chartZoomInteractor"
                width="500"
                height="300"
                styleSheets="/data/line.css"
                waitingImage="data/images/wait.gif"
                imageFormat="PNG"  />
<jvcf:chartOverview id="chartOverview"
                    style="height:100;width:150px"
                    viewId="chartView"
                    lineWidth="3"
                    lineColor="red" />

<jvcf:chartLegend id="legendView"
                  viewId="chartView"
                  width="400"
                  height="180"
                  layout="vertical"
                  waitingImage="data/images/wait.gif" />
```



Projekt: *Gastbeitrag für JSP-develop.de*
Thema: *Implementierung einer grafischen JSF-Komponente*
Länge: *ca. 29.000*
Red.-Schluss: *-*
Autor: *Patrick Mégard, ILOG*

Renderer
Der Renderer ist das komplexeste Element dieser JSF-Implementation. Wie zuvor erwähnt, erzeugt der Renderer keinen einfachen HTML-Code sondern DHTML, das aus HTML (dem <img>-Tag) und JavaScript-Proxies besteht.

Proxies
Ein Proxy ist eine Instanz einer JavaScript-Klasse, die für das Management der Darstellung des Komponenten-Bildes auf dem Client zuständig ist. Dieses Objekt ist die Client-Repräsentation der Server-seitigen Java-Komponenten-Klasse: es hat die gleichen Properties. Jede Komponente auf der Seite, das Chart und seine Begleiter, haben eine Proxy-Instanz.
Beim Rendern von JavaScript empfiehlt es sich, die facesContext.getExternalContext(). encodeNamespace(name) Methode auf jede JavaScript-Variable anzuwenden. Sie erleichter die künftige Integration der Komponente in eine Portlet-Umgebung nach JSR168 [4] Standard.

Script-Abhängigkeiten
Um einen Proxy auf dem Client zu instantiieren, müssen JavaScript Support-Libraries auf die Seite importiert werden. Um den Client so klein wie möglich zu halten, werden die JavaScript-Libraries auf Grundlage der zu unterstützenden Proxy-Klassen modularisiert. Deshalb braucht jede Proxy-Klasse eine andere, möglicherweise überlappende Zusammenstellung von Libraries. Ein schwieriger Teil des Chart-Rendering ist die Phase, die diese Script-Libraries ausgibt. Der Renderer jeder Komponente legt fest, welche Library sie benötigt und welche der zuvor ausgegebenen Libraries bekannt sein müssen, um bei der Ausgabe der Library-Referenzen Duplikate zu vermeiden. Dies wird durch einen Script-Manager erreicht, der nur während des Renderns der Seite existiert. Jedes Mal, wenn der Renderer den Satz Library-Imports erstellen will, gibt er die Liste an den Script-Manager, der die bereits bestehenden Libraries ausfiltert.

Scripting und Status-Synchronisation
Der Sinn und Zweck der Client-seitigen Proxies ist es, Scripting zu ermöglichen und dabei unnötige Seiten-Aktualisierungen zu vermeiden. Ist das Chart einmal gerendert, können die Proxies auf Client-Seite dynamisch Interaktoren installieren oder die Image-Map anzeigen oder verbergen. Die Proxy-Objekte sind auch für reguläre JSF-Komponenten verfügbar, die JavaScript-Mouse-Eventhandling unterstützen.

```
<jvcf:chartView id="chartView" .. />
<h:selectBooleanCheckbox id="genImageMap"                    onclick="chartView.setGenerateImageMap
(this.checked ? true : false, true);" />
```

Die Schwierigkeit mit der lokalen Modifikation des Client-seitigen Proxies der Komponente ist, dass ihr Status nicht mehr mit der Java-Komponente auf dem Server synchronisiert wird. Um das zu umgehen, verwenden die Proxies einen versteckten Input-Tag (<INPUT TYPE=”HIDDEN”>), der den neuen Status auf dem Client speichert. Wenn eine Standard-JSF-Aktion ausgeführt und die Seite abgeschickt wird, wird dieser versteckte Status vom Renderer decodiert, so dass Client und Server synchronisiert werden. Dieses Verhalten erfordert ein spezielles Decoding-Verhalten in der Renderer-Klasse. Die Standard Dekodiermethode wird erweitert, um den Status zu decodieren, der vom Client kommt und um den Server-seitigen Komponenten-Status zu aktualisieren.

Abhängigkeiten der Komponente
Die Verbindung zwischen dem Chart und der zugehörigen Komponente wird durch die Id der Referenzen oder Binding erreicht. Um ein flexibles Seitendesign zu ermöglichen, kann eine Komponente referenziert werden, bevor sie dann tatsächlich gerendert wird. Wenn also während des Renderns eine Komponenten-Property auf eine andere, noch nicht gerenderte Komponente verweist, wird die Ausgabe des JavaScript-Codes, das diese Abhängigkeit auf diesem auflöst verzögert, bis die referenzierte Komponente auch gerendert ist. Diese Aufgabe wird von einem Dependency-Manager übernommen.
Als Beispiel dient hier ein typischer Fall eines Überblicks, der auf ein Chart referenziert.

Beispiel

```
<jvcf:overview viewId="chart" [...] />
<jvcf:chartView id="chart" [....] />
```

Es gibt zwei Fälle:

- Die referenzierte Chart-Komponente ist bereits gerendert, also gibt es kein Problem:

Beispiel

```
JSP:
<jvcf:chartView id="chart" [....] />
<jvcf:overview viewId="chart" id="overview" [...] />

render:
[...]
var chart = new IlvChartViewProxy ( .. );
[...]

var overview= new IlvFacesOverviewProxy ( .. );
overview.setView(chart);
[...]
```

- Die referenzierte Chart-Komponente ist nicht vor der abhängigen Überblicks-Komponente gerendert. In diesem Fall wird beim Dependency-Manager ein component-creation listener registriert. Wenn die referenzierte Chart-Komponente dann letztlich gerendert ist, signalisiert dessen Renderer an den Dependency-Manager, dass sie fertig erstellt wurde. Zu diesem Zeitpunkt wird der Code, der zur Auflösung der Abhängigkeit benötigt wird, ausgegeben:

Beispiel

```
JSP:
<jvf:overview viewId="chart" id="overview" [...] />
<jvdf:chartView id="chart" [....] />

render:
[...]
var overview = new IlvFacesOverviewProxy ( .. );
[...]

var chart = new IlvChartViewProxy ( .. );
overview.setView(chart);
[...]
```

3. Zukünftige IDE-Integration vereinfachen

Ein Ziel der Entwicklung von JSF-Komponenten ist ihre Wiederverwendbarkeit in jeder JSF-kompatiblen IDE. Dennoch reicht JSF-Konformität allein nicht immer aus, damit eine solche Design-Time Integration auch wirklich funktioniert. Die folgenden einfachen Ideen sollte man im Hinterkopf haben, wenn man eine JSF-Komponente entwickelt. Dann funktioniert die IDE-Integration in Zukunft leichter.
Zuallererst sollte die JSF-Komponente grundlegendes HTML-Rendering ermöglichen. Zur Design-Time können JSF IDEs keine dynamischen grafischen Komponenten rendern, die Live-Daten- oder Application Server-Verbindungen benötigen. Deshalb sollten Komponenten, die komplexes oder nicht-

konventionelles Rendering (also nicht HTML) verwenden, Beans.isDesignTime() benutzen, um festzustellen, ob sie eine einfach HTML-Repräsentation oder echtes Komponenten-Rendering bereit stellen sollen.
Ein zweites Design-Time-Problem ist die Poistionierung und Dimensionierung der Komponente. Sun Creator[5] verwendet das 'style'-Attribut, während IBM WSAD[6] mit 'height' und 'width' Properties arbeitet. Eine Komponente, die in der Größe veränderbar sein soll, sollte beide Arten der Größendefinition verarbeiten können.
Um schließlich mit IDEs zu integrieren, muss die Komponente zusätzlich Informationen enthalten, die in der bisherigen JSF-Spezifikation nicht enthalten sind. Leider benötigt jede IDE spezielles Handling zur Integration der Komponente: Sun Creator verlangt ein XML-File, IBM WASD ein Eclipse-Plugin, und so weiter. Ein Hauptziel der nächsten JavaServer Faces JSR (x2.0) wird die Spezifikation dieses zusätzlichen Metadatenformats sein.

4. Fazit

In diesem Artikel wurde dargestellt, wie eine einfach JSF-Komponente geschrieben wird. Das Framework von JSF übernimmt dabei einen Großteil der Arbeit, im wesentlichen die Zusammenarbeit und das Management von verschiedenen Renderern, und so weiter.
Das Basiskonzept wurde dann erweitert, um eine aufwändigere grafische JSF-Komponente zu designen, die komplexe Datensätze darstellen kann, inkrementellen Refresh bietet, Client-seitige Interaktion ermöglicht und sich mit Begleitkomponenten abstimmt. Dafür waren einige Erweiterungen der Basis-JSF-Komponentenarchitektur nötig. Das Konzept der inkrementellen Aktualisierung wäre sicher eine gute Verbesserung künftiger Versionen des JSF-Framework. Das würde die teilweise Aktualisierung von den Teilen der Website erlauben, die aktualisiert werden müssen, und ein vollständiges Neuladen der Daten vermeiden.
Zu guter Letzt wurde aufgezeigt, dass die JSF-Spezifikation nicht immer ausreicht, um eine Komponente vollständig in eine JSF-IDE zu integrieren und dass ein neues JSR diese Probleme bald beheben wird.
Trotz dieser Hürden beschleunigt das JSF-Framework die Entwicklung von Web-Komponenten enorm und erleichtert das Mischen von Komponenten aus verschiedenen Quellen, um vollständige komplexe Web-Anwendungen zu entwickeln.

5. Referenzen

1. JSR 127: JavaServer Faces
   http://jcp.org/en/jsr/detail?id=127
2. Developing Advanced Graphics Components Using JavaServer Faces Technology
   http://www.javaone04.com/session-html/TS-1936.html
3. Creating Custom UI Components
   http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSFCustom.html
4. JSR 168: Portlet Specification
   http://jcp.org/en/jsr/detail?id=168
5. Sun Java Studio Creator http://developers.sun.com/prodtech/javatools/jscreator/index.jsp
6. Rational Application Developer for WebSphere Software
   http://www-306.ibm.com/software/awdtools/developer/application/index.html

6. Zum Weiterlesen

Eine vollständigere Beschreibung der Entwicklung einer JSF-Applikation ist zu finden unter http://java.sun.com/developer/technicalArticles/GUI/JavaServerFaces.