Beispielanwendung unter Einsatz der Quasar Web Client Architektur

Carsten Lucke

28. Januar 2009

Vorwort

Vor der Lektüre dieser Beschreibung zur praktischen Verwendung der Quasar Web Client Architektur (QWCA) sollten Sie die zugehörige Ausarbeitung gelesen haben oder zumindest anderweitig erworbene Kenntnisse zur Quasar Web Client Architektur und idealer Weise auch JavaServer Faces besitzen. Alle hier verwendeten Begrifflichkeiten und Konzepte sind ausführlich in der Ausarbeitung besprochen worden und deren Kenntnis wird daher vorausgesetzt.

Ablauf-Umgebung einrichten

Um die Beispielanwendung testen zu können[1], benötigt man einen Servlet-Container wie etwa den Apache Tomcat Server. Dieser kann unter der Internetadresse *http://jakarta.apache.org/tomcat/* herunter geladen werden. Dort stehen unterschiedliche Varianten für verschiedenste Betriebssysteme zur Verfügung.

Nach der Installation muss die Datei *QWCA-Demo.war* in das *webapps* Verzeichnis unterhalb des Tomcat-Installationspfades (*TOMCAT_HOME*) kopiert werden. Nach einem Neustart des Servers ist die Anwendung im Browser über die URL *http://localhost:8080/QWCA-Demo/* erreichbar.

Die Datei *QWCA-Demo.war* enthält sämtliche Java-Quellcode Dateien der Anwendung. Diese können mit einem herkömmlichen Packprogramm wie Winzip[2] extrahiert werden. Eventuell muss dazu die Dateiendung auf *zip* geändert werden.

Die Demo-Anwendung

Die in der Folge beschriebene Anwendung dient dazu, die interessanten Aspekte des Prototyps der QWCA genauer zu betrachten und die Benutzung der angebotenen Features zu demonstrieren. Die Applikation ist daher sehr einfach gehalten. Es findet weder ein Zugriff auf einen entfernten Anwendungskern statt, noch wird eine Anwendungskernfassade eingesetzt.

Die Demo-Anwendung besteht aus einem Login-Dialog und einer Reihe weiterer Dialoge, die ein Hinzufügen und Ändern von Personendaten erlauben. Der Personensuche-Dialog demonstriert eine einfache Filtermöglichkeit nach dem Personennamen. Die Anwendung demonstriert folgende Features:

- Konfiguration der Applikation
- Standard-Services der QWCA
- Dialogerstellung
- Struts-Tiles Integration
- Redirect after Post
- Behandlung veralteter Dialogzustände und von Double-Submits

Konfiguration der Applikation

Um eine Webanwendung mit der QWCA zu entwickeln, muss zuerst einmalig eine Konfiguration innerhalb des Web-Application Deployment Deskriptors (*web.xml*) und der JavaServer Faces Konfigurationsdatei (*faces-config.xml*) vorgenommen werden. Beide Dateien befinden sich im *WEB-INF* Verzeichnis der *QWCA-Demo.war* Datei. Aus Platzgründen wird auf eine Darstellung des gesamten Inhaltes dieser Dateien hier verzichtet.

Einstellungen, die in der *web.xml* vorgenommen werden:

- Name(n) der JSF-Konfigurationsdatei(en) und FacesServlet
- State-Saving Methode von JSF

[1] Download aller Dateien unter http://architektur-grafischer-bedienoberflächen.de/.

[2] http://www.winzip.de/

- Implementierungsklasse des QWCA Application Interfaces
- Implementierungsklasse des QWCA ConfigManager Interfaces
- Applikationsweite Konfigurationsdatei (enthält weitere Konfigurationsdaten für die Anwendung)
- Konfiguration von *DialogServletFilter* und *SequenceNumberServletFilter*
- Konfiguration des Tiles-Servlet für Jakarta Struts Tiles Integration

Einstellungen, die in der *faces-config.xml* vorgenommen werden:

- Integration der von der QWCA bereitgestellten JSF-Austauschkomponenten:
  - ApplicationFactory
  - ViewHandler
  - NavigationHandler
  - VariableResolver
  - PropertyResolver
  - FormRenderer

Die in der *web.xml* definierte Konfigurationsdatei (siehe Listing 1) enthält unter anderem die Definition des UnknownRequestHandler, InvalidSequenceNumberHandler, applikationsglobalem ServiceManager, SessionManager und Konfiguration der Session-Umgebung. Die Konfiguration der Session-Umgebung wird anhand der Inhalte einer weiteren Konfigurationsdatei vorgenommen (in der Demo-Anwendung *SessionConfig.xml*).

Diese XML basierten Konfigurationsdateien verwendeten die Dependency-Injection Features des Spring-Frameworks[3]. In der Konfigurationsdatei für die Session-Umgebung (Environment oder veraltet auch Dialograhmen genannt) definiert man Komponenten wie den DialogController, DialogManager, HierarchyManager, ServiceManager oder die Session selbst. Auf den ersten Blick mag der Umfang der zu konfigurierenden Komponenten übermächtig erscheinen. Allerdings nimmt man diese Konfiguration nur einmal pro zu entwickelnder Applikation vor. Außerdem hat die Konfiguration der Komponenten auf diese Weise den großen Vorteil, dass bestehende Abhängigkeiten automatisch aufgelöst werden. Zum Beispiel erhält der DialogController eine Referenz auf die von ihm benötigte DialogManager-Komponente. Nach der Konfiguration bekommt man die Initialisierung der Komponenten quasi „geschenkt“.

```
<context-param>
  <param-name>com.sdm.quasar.client.components.
      configuration.implementation.
      DefaultConfigManager.CONFIG</param-name>
  <param-value>QWCA-ProtoConfig.xml</param-value>
</context-param>
```

Listing 1: Definition der applikationsweiten Konfigurationsdatei

Vordefinierte Services

Die Umgebungskonfiguration erlaubt die Definition von sitzungsweit gültigen Diensten, die über den ServiceManager bereitgestellt werden. Diese wären demnach direkt unter Angabe der aktuellen Instanz der `SessionId` und des gewünschten Service-Interfaces abrufbar.

Die QWCA bietet bereits einige Services, die hier definiert werden können:

- DialogControlService (QNCA + QWCA): Bietet Zugriff auf die Funktionalität des DialogControllers
- DialogLifecycleNotificationService (QNCA + QWCA): Ermöglicht Registrierung von Observern für den Lebenszyklus von Dialog-Instanzen
- DoubleSubmitChecker (QWCA): Ermöglicht Prüfung von doppelten Submits. Wird in der Regel Framework-intern verwendet.

[3] http://www.springframework.org/

- PresentationURIService (QWCA): Bietet Zugriff auf die URL der JSP-Präsentationsdatei von Dialogen.
- DialogChangeService (QWCA): Ermöglicht den Wechsel zwischen Dialogen.
- SingletonDialogService (QNCA + QWCA): Bietet Zugriff auf Singleton-Dialog Objekte.

Neben den von der Session bereitgestellten Services, erfolgt auch die Angabe einer Konfigurationsdatei, die Informationen über die einzelnen Dialoge der Anwendung enthält (siehe Listing 2).

```
<bean id="dialogconfigprovider" class="com.sdm.quasar
    .client.architecture.dialogmanager.springconfig.
    SpringDialogConfigurationProvider" autowire="
    constructor">
  <constructor-arg value="com/sdm/clucke/da/prototype
      /dialogs/DialogsSpringConfig.xml" />
</bean>
```

Listing 2: Angabe der Konfigurationsdatei für Dialoginformationen

Die Datei enthält Daten über die Dialog-Id, die implementierende Klasse, den Dialogtyp (Singleton oder Multiton) und vieles mehr. Für jeden Dialog einer Applikation muss ein Eintrag in dieser Konfigurationsdatei erfolgen.

Damit sind alle notwendigen Konfigurationsdateien besprochen und wir können mit der Betrachtung der eigentlichen Anwendung beginnen.

LoginDialog

Der Einstieg in den Ablauf der Applikation erfolgt über einen Login-Dialog. Wie in der Ausarbeitung beschrieben besteht in der QWCA jeder Dialog aus den Komponenten *Dialogkern* und *Präsentation*. Der Dialog respektive sein Dialogkern werden im Beispiel durch eine Java-Klasse repräsentiert. Die Präsentation wird in einer JSP-Datei beschrieben. Dazu verwendet man eine Mischung aus HTML und Tags, die von Taglibs bereitgestellt werden. Die Taglib-Tags erlauben die Definition von UI-Komponenten. Außerdem enthält die JSP-Datei die Data- und Action-Bindings, die mit Hilfe der JSF Expression Language (JSF-EL) definiert werden.

Wir betrachten zuerst die Java-Klasse, die den eigentlichen Dialog ausmacht. Dabei handelt es sich um die Klasse LoginDialog aus dem Package `com.sdm.clucke.da.prototype.dialogs`. Um die vom QWCA-Framework bereitgestellte Basisfunktionalität eines Dialogs zu erben, erweitert der Dialog die Klasse `com.sdm.quasar.client.architecture.dialog.jsf.AbstractSimpleDialog`. Alles, was der Entwickler dazu tun muss, ist einen entsprechenden Basisklassen-Konstruktor zu implementieren (siehe Listing 3).

```
/**
 * Constructor
 * @param diid the dialog's DIID
 * @param dialogEnvironment the dialog's environment
     (actually this is the session)
 * @param context the context-information
 */
public LoginDialog(DialogInstanceId diid,
    DialogEnvironment dialogEnvironment, Object
    context) {

  super(diid, dialogEnvironment, context);
}
```

Listing 3: Konstruktor der Klasse LoginDialog

Wie für einen Login-Dialog nicht untypisch, verfügt dieser über Attribute für einen Benutzernamen und ein Passwort sowie entsprechende Getter- und Setter-Methoden. Außerdem hat die Klasse die Methode `handleLogin()`, die bei einem Loginversuch aufgerufen werden soll, um übermittelte Logindaten zu überprüfen und abhängig vom Resultat einen Dialogwechsel vorzunehmen oder dafür zu sorgen, dass erneut der Login-Dialog angezeigt wird.

Dialogkonfiguration

Ist die Dialogklasse erstellt, kann und muss man die Dialogkonfiguration vornehmen. Der Eintrag in der Datei *DialogsSpringConfig.xml* sieht wie folgt aus (siehe

Listing 4):

```
<bean id="LoginDialog" class="com.sdm.quasar.client.
    architecture.dialog.jsf.
    DefaultJSFDialogConfiguration">
  <property name="dialogId" value="LoginDialog"/>
  <property name="className" value="com.sdm.clucke.da
      .prototype.dialogs.LoginDialog"/>
  <property name="singleton" value="true"/>
  <property name="presentationURI" value="/pages/
      LoginDialog.faces"/>
</bean>
```

Listing 4: Konfiguration des Login-Dialogs

Hier erfolgen Angaben über den Dialogtyp, die URL der Präsentation (JSP-Datei), etc.

Dialogpräsentation

Die Präsentation eines Dialogs wird wie bereits angesprochen in einer JSP-Datei definiert. In der Regel enthält eine solche JSP-Datei einen Mix aus HTML und speziellen Tags, die JSF UI-Komponenten repräsentieren. UI-Komponenten, die Werte „tragen“ können, werden außerdem durch Verwendung der JSF Expression Language mit Value-Bindings versehen. Command-Buttons und Command-Links können Action-Bindings enthalten, die Dialogaktionen auslösen. Listing 5 zeigt die JSP-Datei für den Login-Dialog.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix=
    "h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix=
    "f" %>

<html>
  <body>
  <f:view>
    <h:form>
      <table border="0">
        <tr>
          <td><h:outputText value="Username:␣"/></td>
          <td><h:inputText id="uid" size="15"
              required="true"
               value="#{DIALOG.DATA.username}" /></
                  td>
        </tr>

        <tr>
          <td><h:outputText value="Passwort:␣"/></td>
          <td><h:inputSecret id="passwd" size="15"
              required="true"
             value="#{DIALOG.DATA.password}" /></td>
        </tr>

        <tr><td colspan="2"> </td></tr>

        <tr>
          <td colspan="2"><h:commandButton value="
               Einloggen"
              styleClass="button" action="#{DIALOG.
                  CORE.handleLogin}"/></td>
        </tr>
      </table>
    </h:form>
  </f:view>
  </body>
</html>
```

Listing 5: Präsentation des Login-Dialogs

Wie zu erkennen erfolgt innerhalb der Datei das Value-Binding der Eingabefelder für Passwort und Username an die entsprechenden Attribute des aktuellen Dialogs, welcher die Klasse LoginDialog ist. Dazu wird die Pseudo-Bean `DIALOG` verwendet. Der Zugriff auf die Datenhaltung des Dialogs erfolgt mit `DIALOG.DATA`. Somit erfolgt das Binding an das passende Dialogattribut über `DIALOG.DATA.username` bzw. `DIALOG.DATA.password`.

Der Submit-Button wird ebenfalls über Verwendung der Pseudo-Bean `DIALOG` mit dem Action-Binding Ausdruck `DIALOG.CORE.handleLogin` an die zuvor betrachtete `handleLogin()` Methode der Klasse LoginDialog gebunden. Diese Methode nimmt eine Prüfung der Logindaten vor. Bei korrekten Daten erfolgt ein Dialogwechsel zum Hauptdialog der Anwendung, im Fehlerfall wird erneut der Login-Dialog angezeigt (siehe Listing 6).

```
public String handleLogin() {
  if (this.checkLoginData()) {
    try {
      this.changeDialog("PersonSearchDialog", null);
    } catch (ClientException e) {
      Assertion.fail("Dialog-Change␣failed");
    }
  }

  return "__redirect__";
}
```

Listing 6: handleLogin() Methode des Login-Dialogs

Damit ist der Login-Dialog komplett und einem Aufruf im Browser steht nicht mehr im Wege.

Erster Kontakt

Beim ersten Kontakt eines „Surfers“ mit der Anwendung erfolgt der Einstieg in die Applikation. Zu diesem Zeitpunkt sind keine Dialoginstanzinformationen vorhanden. An dieser Stelle instantiiert der DialogServletFilter den in der Konfiguration definierten UnknownRequestHandler, welcher für die Erzeugung des Einstiegsdialogs zuständig ist. Der Handler wird vom Anwendungsentwickler programmiert (siehe Listing 7) und ermöglicht den Einstieg in den Ablauf der Anwendung. In der Beispielanwendung erfolgt der Einstieg in die Applikation über den Login-Dialog.

```
package com.sdm.clucke.da.prototype.handlers;

public class DefaultUnknownRequestHandler implements
    UnknownRequestHandler {

  public DialogInstanceId handleRequestWithoutDiid(
      ServletRequest request, DialogEnvironment
      dialogEnvironment) {

    DialogControlService dialogControlService = (
        DialogControlService)dialogEnvironment.
        findServiceInHierarchy(dialogEnvironment.
        getSessionId(), DialogControlService.class);
    DialogInstanceId diid = dialogControlService.
        createIndependentDialog("LoginDialog", null)
        ;
    return diid;
  }
}
```

Listing 7: Anwendungsspezifische Implementierung des UnknownRequestHandler Interfaces

PersonSearchDialog

Nach erfolgreicher Anmeldung bewirkt der LoginDialog einen Dialogwechsel zum PersonSearchDialog. Der LoginDialog war ein einfacher Dialog, der für sich allein stand. Die Präsentation des PersonSearchDialogs hingegen wird zusammen mit anderen statischen Seitenbestandteilen dargestellt. Diese statischen Bestandteile sind eine einfache Navigation auf der linken Seite sowie ein Header und ein Footer (siehe Abbildung 1).

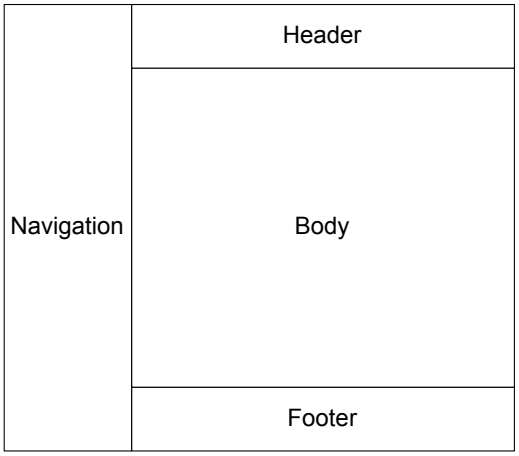

Abbildung 1: Seitenbestandteile des Tiles-Layouts

Tiles-Integration

Die dynamische Komposition einer Seite aus unterschiedlichen Bestandteilen erfolgt mit Hilfe der Struts Tiles Bibliothek. Dazu wird das Tiles-Servlet in der *web.xml* eingetragen und eine globale Tiles-Konfiguration angegeben, welche bei der Beispielanwendung in der Datei *tiles-defs.xml* vorgenommen wird (siehe Listing 8).

```
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles
        Configuration//EN"
    "http://jakarta.apache.org/struts/dtds/tiles-
        config.dtd">

<tiles-definitions>
  <definition name="page.template" path="/templates/
      default-frame.jsp"/>
  <definition name="page.header" path="/tiles/header
      .html"/>
  <definition name="page.lnav" path="/tiles/lnav.jsp
      "/>
  <definition name="page.footer" path="/tiles/footer
      .html"/>
</tiles-definitions>
```

Listing 8: Globale Tiles-Konfiguration

In dieser Konfigurationsdatei erfolgt die Angabe des Rahmentemplates, welches das in Abbildung 1 gezeigte Layout umsetzt. Außerdem werden hier die statischen Bestandteile (Navigation, Header und Footer) definiert, die immer den gleichen Inhalt haben. Dadurch müssen

diese nur einmalig und nicht wiederholt in jeder einzelnen Seite definiert werden. Beim Aufruf einer Seite ist nur die Angabe des „Tiles“ notwendig, das den dynamischen Inhalt des Body-Bereiches enthält. Listing 9 zeigt das Rahmentemplate. Für die Bereiche *Navigation, Header* und *Footer* werden die in der *tiles-defs.xml* definierten Bestandteile verwendet. Der Bereich für den *Body* wird so angelegt, dass er zur Laufzeit über ein Attribut festgelegt werden kann.

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix=
    "f" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags
    -tiles-1.1" prefix="tiles"%>

<html>
<head>
  <style type="text/css">
    <!-- @import url("../css/default-styles.css"); --
        >
  </style>
</head>

<body>
<f:view>
  <table width="100%">
    <tr>
      <td colspan="2" align="center">
        <tiles:insert definition="page.header" flush=
            "false"/>
      </td>
    </tr>

    <tr>
      <td width="20%" align="left" valign="top">
        <tiles:insert definition="page.lnav" flush="
            false" />
      </td>
      <td align="left" valign="top">
        <tiles:insert attribute="body" flush="false"/
            >
      </td>
    </tr>

    <tr>
      <td colspan="2" align="center">
        <tiles:insert definition="page.footer" flush=
            "false"/>
      </td>
    </tr>
  </table>
</f:view>
</body>
</html>
```

Listing 9: Rahmentemplate des Tiles-Layouts

Beim Dialogwechsel vom LoginDialog zum PersonSearchDialog wird die Seite PersonSearchDialog.jsp (siehe Listing 10) aufgerufen. Diese enthält nur wenige Zeilen, die den Inhalt für den *Body*-Bereich festlegen.

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags
    -tiles-1.1" prefix="tiles"%>

<tiles:insert definition="page.template">
  <tiles:put name="body" value="/tiles/
      PersonSearchDialogTile.jsp" type="page"/>
</tiles:insert>
```

Listing 10: Für PersonSearchDialog aufzurufende Datei

Die eigentliche Präsentation des PersonSearchDialogs befindet sich in der Datei *PersonSearchDialogTile.jsp* im Verzeichnis *tiles*. Diese Datei enthält wiederum einen Mix aus HTML und JSF-Tags in Verbindung mit der JSF-EL zum Data- und Action-Binding.

Weitere Dialoge

Neben den bereits erwähnten Dialogen existieren noch weitere, wie der AddPersonDialog und EditPersonDialog. Ganz allgemein existiert für jeden Dialog eine Java-Klasse, die den AbstractSimpleDialog der QWCA erweitert, ein Eintrag in der Dialogkonfigurationsdatei und eine JSP-Datei für die Präsentation. Bei Tiles-basierten Layouts gibt es daneben noch eine Datei, die den Aufruf eines Dialogs behandelt und dynamische Tiles-Bestandteile festlegt.

Die JSP-Dateien für die Präsentation und die Java-Klassen der Dialoge werden genauso programmiert, als würde man „JSF pur“ entwickeln. Zusätzlich stehen die von der QWCA bereitgestellten Extra-Features wie Pseudo-Beans und die Tiles-Integration zur Verfügung.

Besondere Problemstellungen

Redirect after Post

Eine wichtige Anforderung an die QWCA ist, dass nach POST-Requests ein Redirect erfolgt. Die Gründe dafür wurden in der Ausarbeitung hinreichend beschrieben.

Requestergebnisse ohne Dialogwechsel werden von JSF in der Regel ohne Redirect geliefert. Wird der String "__redirect__" als Outcome geliefert, führt der PRGNavigationHandler einen Redirect durch.

Dialogwechsel erfordern immer einen Redirect. Der Programmierer muss sich hierbei nicht explizit um dessen Initiierung kümmern. Stattdessen wird durch den Dialogwechsel-Service implizit auch ein Redirect durchgeführt. Der Dialogwechsel erfolgt entweder über die Dialog-Methode `changeDialog()` oder über den DialogChangeService.

Double-Submit Problem

Zur Behandlung des Double-Submit Problems wird der SequenceNumberServletFilter in der *web.xml* registriert. Dieser überprüft bei POST-Requests, ob die mitgelieferte Sequenznummer des Dialogs aktuell ist. Im Fehlerfall wird der konfigurierte InvalidSequenceNumberHandler instantiiert und führt eine anwendungsspezifische Fehlerbehandlung durch. Wie der UnknownRequestHandler, muss in der Regel auch dieser Handler speziell für eine Anwendung programmiert werden. In der Demo-Anwendung kommt die Klasse DefaultInvalidSequenceNumberHandler (siehe Listing 11) zum Einsatz. Diese bewirkt die Anzeige eines Fehlerdialogs, welcher einen Link anbietet, der zur aktuellen Instanz des angeforderten Dialogs führt.

```
package com.sdm.clucke.da.prototype.handlers;

public class DefaultInvalidSequenceNumberHandler
    implements InvalidSequenceNumberHandler {

  public DialogInstanceId
      handleRequestWithInvalidSequenceNumber(
      ServletRequest request, DialogEnvironment
      dialogEnvironment, DialogInstanceId errorDIID)
       {

    DialogControlService dialogControlService = (
        DialogControlService) dialogEnvironment.
        findServiceInHierarchy(dialogEnvironment.
        getSessionId(), DialogControlService.class);
    PresentationURIService uriService = (
        PresentationURIService) dialogEnvironment.
        getServiceDirect(dialogEnvironment.
        getSessionId(), PresentationURIService.class
        );
    FacesContext fc = FacesContext.getCurrentInstance
        ();
    Assertion.checkNotNull(fc, "FacesContext must not
        be null");
    ExternalContext ec = fc.getExternalContext();

    String linkURL = uriService.getPresentationURI(
        errorDIID) + "?" + TaggingFormRenderer.
        TAG_REQUEST_PARAMETER_NAME + "=" + Util.
        diid2URLString(errorDIID);
    linkURL = ec.encodeActionURL(ec.
        getRequestContextPath() + linkURL);

    MessageDialogContext msgDlgCtxt = new
        MessageDialogContext("Fehler", "Ein Request
        auf einen veralteten Dialogzustand wurde
        abgefangen. Nutzen Sie die unten stehende
        URL, um die aktuelle Dialoginstanz
        anzuzeigen.", "Zur aktuellen Instanz",
        linkURL);
    DialogInstanceId diid = dialogControlService.
        createIndependentDialog("MessageDialog",
        msgDlgCtxt);

    return diid;
      }
}
```

Listing 11: InvalidSequenceNumberHandler der Demo-Anwendung

Fehler, die auftreten, wenn Dialoge in mehreren unterschiedlichen Fenstern gleichzeitig bearbeitet werden, können damit sauber abgefangen und exakt den Anforderungen der jeweiligen Anwendung behandelt werden.

Zusammenfassung

Die QWCA erlaubt eine zügige und saubere Entwicklung dialogbasierter Web-Clients. Dies wird nicht zuletzt durch den Einsatz von JavaServer Faces als Komponente für das GUI-Frontend erreicht. Voraussetzung für die Arbeit mit der QWCA ist natürlich eine gute Kenntnis von JSF.

Die QWCA macht JSF dialogfähig!