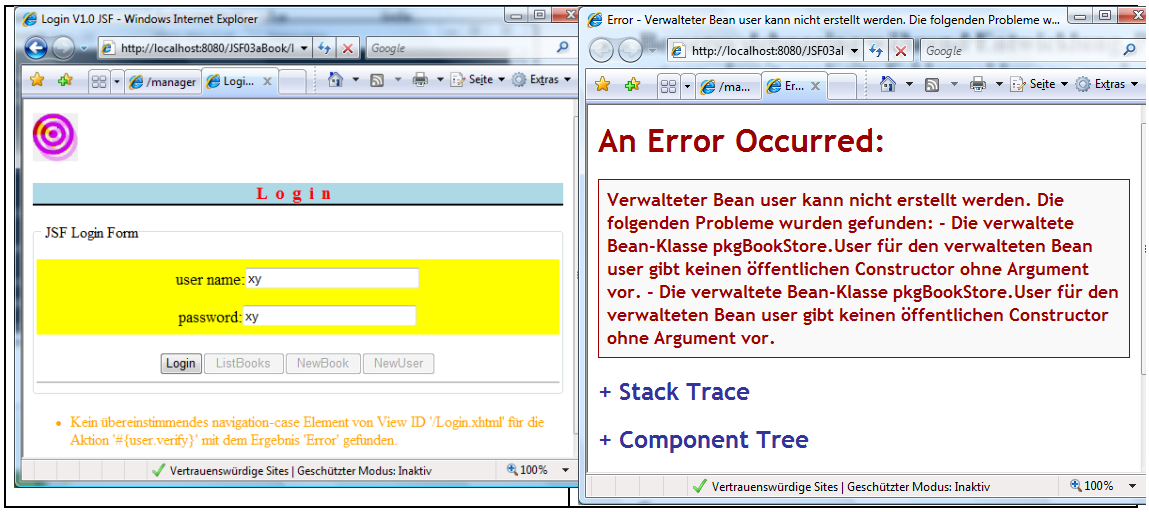
Inhalt

1. New Features in JSF 2.0

- **Besseres debugging während Entwicklung.** PROJECT_STAGE in der *web.xml* setzen, damit explizite Fehlermeldungen erscheinen. Damit ist auch ein „System.out.println("bis daher");“ zwecks Fehlersuche sinnvoll.



- **Durchgehend JSF pages.** Die Seiten werden *irgendwas.xhtml* bezeichnet, die URL ist jedoch *irgendwas.jsf* (vorausgesetzt ein url-pattern *.jsf gibt es in web.xml). Keine *@taglib*, sondern *xmlns:h="http://java.sun.com/jsf/html"* verwenden. Dann kann h:head, h:body und h:form (nicht f:view) in der Page verwendet werden.
- **Default Bean Bezeichner.** es reicht *@ManagedBean* über der Class Definition. Der Klassenbezeichner der Bean wird genommen, der erste Buchstabe wird ein Kleinbuchstabe und so heißt dann das Bean. ZB wird aus *package1.package2.MyBean* im Code *#{myBean.whatever}*. Der Bean-Scope ist standardmäßig „request“, mit Annotationen wie *@SessionScoped* wird er geändert
- **Default mapping bzgl Navigation.** Gibt es keine explizite Navigations-Regel, dann ist der Return Values des Action Controller gleichzeitig der Filename der nächsten aufzurufenden Seite. ZB die Form (file form.xhtml, URL form.jsf) beinhaltet <h:commandButton ... action="#{someBean.someMethod}"/>. Wird er Button geklickt, das Bean *someBean* existiert (zB Request Scope), Setter-Methoden passen mit *h:inputBlah* zusammen, Validation wird durchgeführt, und *someMethod()* wird ausgeführt. Wenn *someMethod()* "foo" und "bar" zurückgibt, und es gibt keine expliziten Navigation Rules in *faces-config.xml* für diese Ergebnisse, dann ruft JSF automatisch *foo.xhtml* und *bar.xhtml* auf.
- **Verwendung von #{myBean.myProperty} statt <h:outputText value="#{myBean.myProperty}"/>.** *h:outputText* wird nur benötigt, wenn *escape="false"* benötigt wird, oder wenn eine Render Property geändert werden soll
- **Ajax-Erweiterung der Application.** *xmlns:f="http://java.sun.com/jsf/core"* am Page Header hinzufügen. Innerhalb der Start/Ende Tags von *h:commandButton* noch *<f:ajax execute="@form" render="resultId"/>* angeben. Weiters *<h:outputText value="#{myBean.myProperty}" id="resultId"/>*. Das bedeutet, dass wenn der Button geklickt wird, werden alle Form Elemente zum Server geschickt und normal ausgeführt. Dann wird er Wert von *getMyProperty* berechnet und zum Server

zurückgesendet. JavaScript erhält den Wert und fügt es in die aktuelle Seite ein (wo *h:outputText element* ursprünglich gestanden ist)

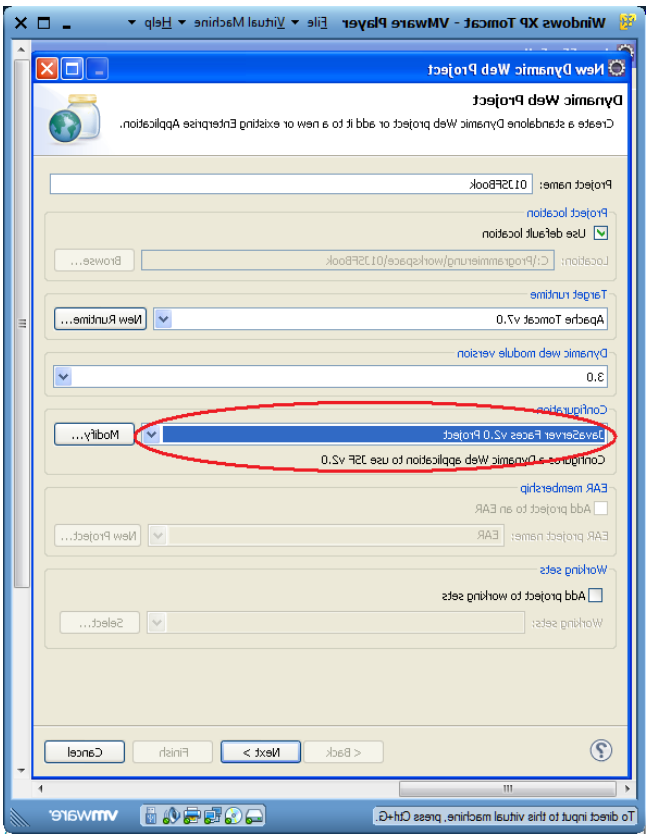
- **Einfachere Verwendung von Custom Components.** Bei JSF 1.*x* ist die Verwendung der third-party component libraries zB PrimeFaces, RichFaces, IceFaces, Tomahawk, ADF, und Web Galileo eine Herausforderung für JSF Programmierer.Jetzt geht es etwas einfacher.

2. Was wird gebraucht

Grundsätzlich

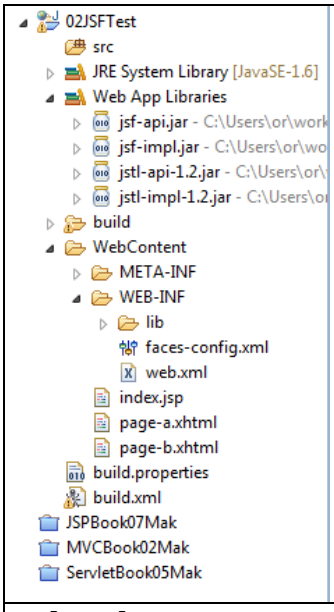
- Java SE 6
- Eclipse 3.6+
- Tomcat 6+ (mit 2x jsf.jar zu jedem Projekt/lib siehe Projektbaum unten)
- java/lib/ext... javax.faces-2.1.7.jar

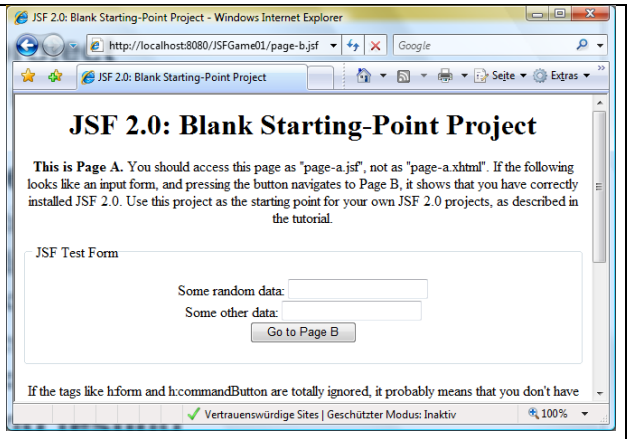
Eclipse Projekt anlegen:



nachfolgend fragt er nach User-Librarys, die kann man trotz Proteste seitens Eclipse „disable library“ auswählen;

Achtung: web.xml muss korrigiert werden: <url-pattern>*.jsf</...> ändern





web.xml

```
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

<url-pattern>...verwende JSF wenn URL mit *.jsf endet; auch möglich: *.faces

a-page.xhtml

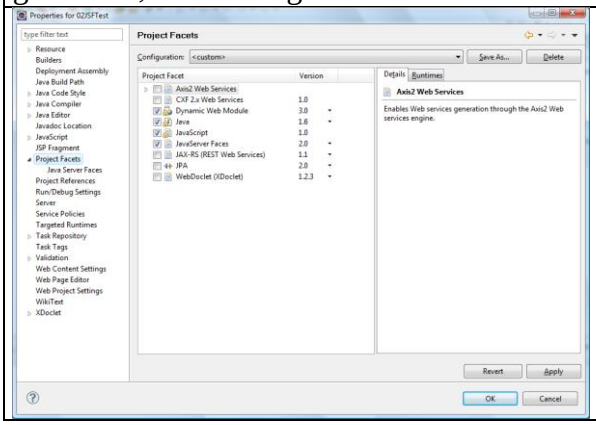
```
<h:head><title>JSF 2.0: Blank Starting-Point Project</title>
</h:head>
<h:body>
<div align="center">
<h1>JSF 2.0: Blank Starting-Point Project</h1>
<p>
<b>This is Page A.</b> You should access this page as "page-a.jsf",
not as "page-a.xhtml". If the following looks like an input form, and pressing
the button navigates to Page B, it shows that you have
correctly installed JSF 2.0. Use this project as the starting point
for your own JSF 2.0 projects, as described in the tutorial.
</p>
<fieldset>
<legend>JSF Test Form</legend>
<h:form>
  Some random data: <h:inputText/><br/>  <!-- Textfield ignored -->
  Some other data: <h:inputText/><br/>   <!-- Textfield ignored -->
  <h:commandButton value="Go to Page B"
                   action="page-b"/>     <!-- Navigates to page-b.jsf -->
</h:form>
</fieldset>
```

faces-config

steht nichts Spannendes drinnen, wird aber gebraucht, damit das ganze läuft.

index.html

<% response.sendRedirect("page-a.jsf"); %>



context.xml

```
<Context reloadable="true"
antiResourceLocking="true">
...
</Context>
```

damit jsf*.jar-Files auch wieder gelöscht werden können bei undeploy

3. XHTML – Grundlagen

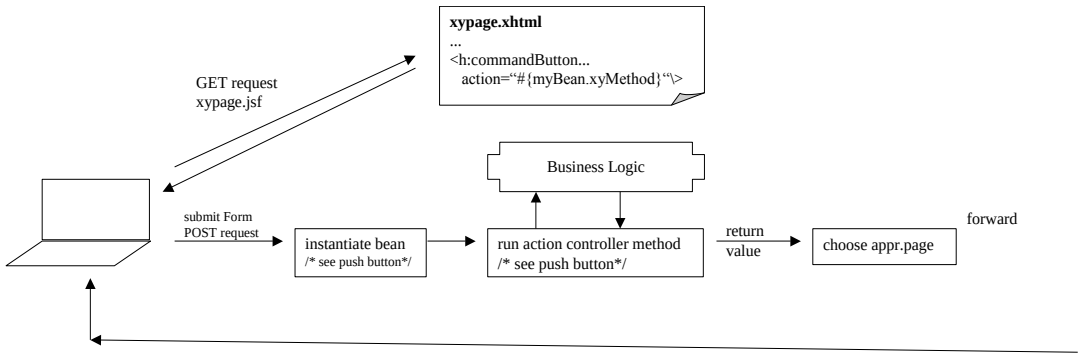
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">                /*1*/
<h:head><title>JSF 2.0: Blank Starting-Point Project</title>
```

```
</h:head>
<h:body>                                                    /*2*/
<div align="center">
<h1>JSF 2.0: Blank Starting-Point Project</h1>
<fieldset>
<legend>JSF Test Form</legend>
<h:form>
  Some random data: <h:inputText/><br/>                     /*3*/
  Some other data: <h:inputText/><br/>
  <h:commandButton value="Go to Page B"
                   action="page-b"                          /*4*/
</h:form>
</fieldset>
```

/*1*/...nicht JSP, sondern JSF sind die Standard – URL (File ist XHTML)
/*2*/...wird dann nötig, wenn im XHTML <h:outputScript> oder <f:ajax> vorkommt.
/*3*/...wird hier noch ignoriert
/*4*/...gehe zur Seite „page-b.xhtml“;
es könnte auch zB action=“#{healthBean.signup}“ stehen (ruft dann Methode auf; Bean-Bezeichner wird aus gleichnamiger Klasse abgeleitet)

```
package pkgHealth;
import javax.faces.bean.*;
@ManagedBean                                /*1*/
public class HealthPlanBean {
     public String signup() {
          if (Math.random() < 0.2) {
               return("accepted");          /*2*/
          } else {
               return("rejected");          /*2*/
          }
     }
}
/*1*/...braucht keinen Eintrag in faces-config.xml
/*2*/...gibt es keinen expliziten Navigations-Eintrag in faces-config.xml,
     wird damit „accepted.xhtml“ bzw. „rejected.xhtml“ aufgerufen.
```

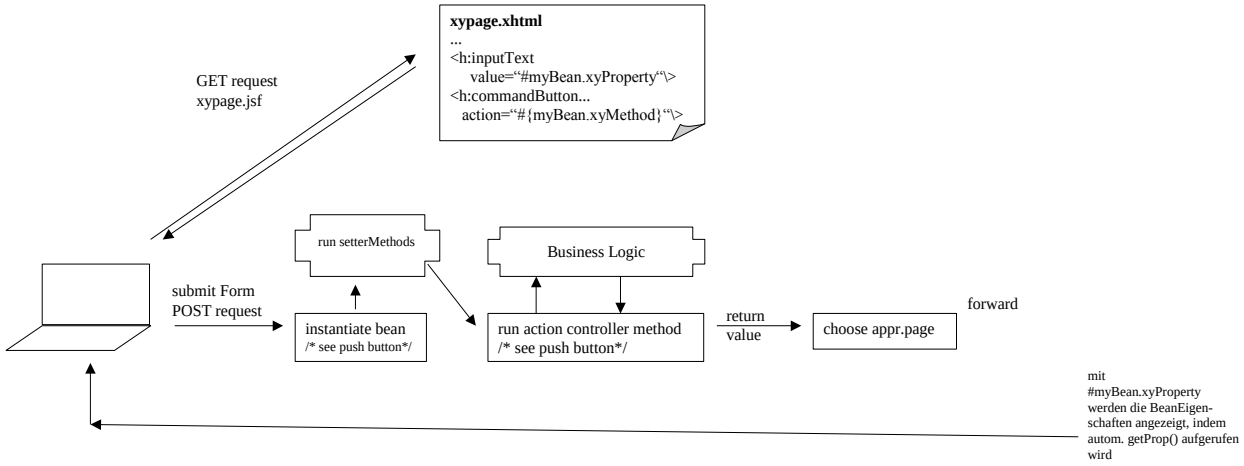
3.1. CommandButton



Managed Beans

- meist POJOs, dh kein spezielles Interface, keine JSF-spezifischen Argumente
- jede Menge getter/setter – Methoden für <h:input/outputText...>
- eine „action controller method“, welche keine Parameter und String als Rückgabe hat; Aufruf durch Button in XHTML – Form; Rückgabewert entscheidet über Weiternavigation.

3.2. InputText



Standardmäßig wird das Bean 2x instantiiert; zuerst um „inputText“ via Getter zu füllen (falls es was gibt), dann nach dem Submit, um mittels Setter den „inputText“ auszulesen.

4. Beans

Deklaration

- einfach...@ManagedBean vor der Klassendeklaration (request-scope)
- sonst...<managed-bean> in *faces-config.xml*.

Problem

Getter/Setter werden häufig aufgerufen; wenn das DBk-Zugriffe erfordert, wird Performance uU schwach.

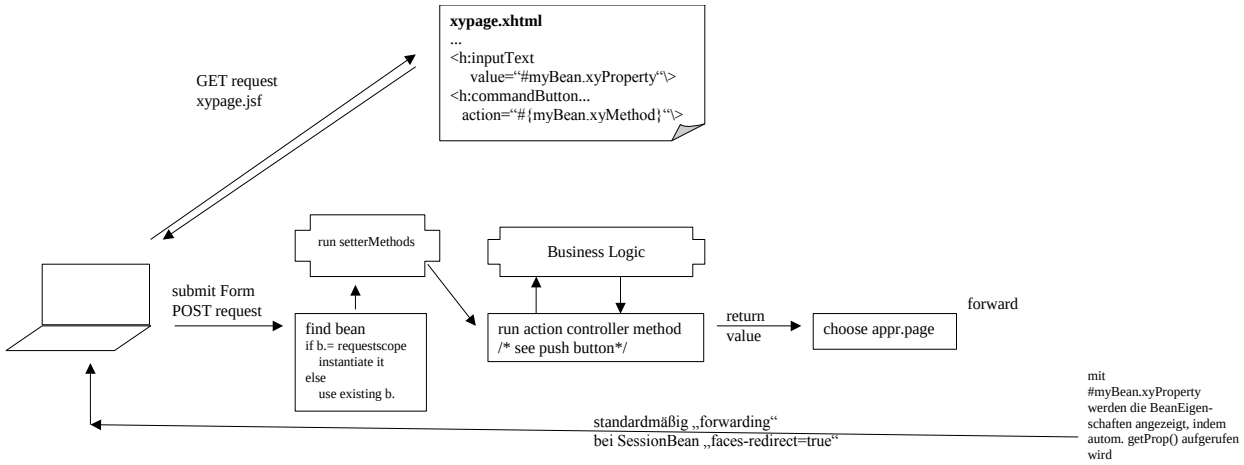
Lösung: Daten möglichst in Instanz-Variablen speichern und von dort via Getter holen.

Managed Beans

- getter/setter für Poperties
- pro Button eine action controller method
- grundsätzlich ist pro HTML-Seite ein dazugehöriges managedBean nicht schlecht

Scope

- request, session, application, view, none, custom
- Festlegung: faces-config.xml oder Annotation, zB @SessionScoped
- request...instantiiert bei jedem HTTP-Request; eigentlich 2x, weil zuerst für Getter, dann nach Benutzereingabe für Setter.
- session...Bean sollte dann Serializable sein (einige Server (speziell in verteilten Umgebungen) sichern Session-Infos auf Platte)
- application...für alle Benutzer der Applikation zugänglich; uU aufpassen wegen gleichzeitigem Zugriff (synchronisieren); weiters aufpassen, wenn Beans aus verschiedenen Packages gleich lauten (zB „Database“); es wird nur eine Instanz erstellt, auf die dann ALLE anderen Beans zugreifen; dass ist lästig, wenn nicht erwünscht; daher möglichst gleichlautende Klassenbezeichner bei „ApplicationsScope“ vermeiden.
- view...Bean instantiiert sich bei jedem Seitenaufruf; Bean sollte Serializable sein
- custom...Bean wird in einer Map gespeichert, Programmierer kann Lifecycle steuern.
- none...Beans, die durch andere (scoped) Beans erzeugt werden



Redirect

```
<h:commandButton value="ListBooks" action="ListBooks?faces-redirect=true"/>
```

5. Forms - Elemente[1]

5.1. Einfache Eingabefelder

- <h:inputText value="#{someBean.a.b.c}"/>...Getter werden alle aufgerufen (von links nach rechts, also getA().getB().getC(); Setter nur der von „c“.
- <h:selectBooleanCheckbox value="#{someBean.someProperty}"/>...Property muss Boolean sein (getSomeProperty() oder isSomeProperty())

5.2. SelectMenu

```
    user name: <h:selectOneMenu  value="#{login.name}">
         <f:selectItems value="#{login.allUsers}"/>
    </h:selectOneMenu>
**********************
    import javax.faces.model.SelectItem;
    ...
    public List<SelectItem> getAllUsers() {
         List<SelectItem> retList = new ArrayList<SelectItem>();
         try {
              for(String it:db.getAllUsers()) {
                   retList.add(new SelectItem(it));
              }
         } catch (Exception e) {
              message = "error happened: " + e.getMessage();       }
         return retList;
    }
```

5.3. Tabelle und Schleife

ListBooks.xhtml

```
         <table>
              <ui:repeat var="book" value="#{listBooks.books}">
              <tr>
                   <td>#{book.id}</td>
                   <td>#{book.title}</td>
                   <td>#{book.author}</td>
                   <td>#{book.price}</td>
                   <td>#{book}</td>
```

[1] http://www.jsftoolbox.com/documentation/help/12-TagReference/html/

```
            </tr>
            </ui:repeat>
        </table>
```

ListBooks.java

```
    public List<Book> getBooks() {
        books.clear();
        try {
            books.addAll(db.getBooks(title));
            message = "some books found";
        } catch (Exception e) {
            message = "error:" + e.getMessage();
        }
        return books;
    }
```

6. Zugriff auf Request & Response - Objekt

Sollte grundsätzlich möglichst nicht verwendet werden, weil sehr schnell komplizierter Code herauskommt. Zuerst alle JSF – Goodies nutzen.

Ausnahmesituationen

- explizite Session-Manipulation, zB Session – Abbruch, Manipulation von Cookie – Eigenschaften
- Erstellen/Abfragen eines (unmanaged) Beans in einem POJO
- erfragen von Request – Infos (zB Hostname udgl.)
- Manipulation des Response - Headers

Beispiel – Bean „Database“

Login.java

```
    ...
    private void generateBeanDb(){
        ExternalContext context =
                FacesContext.getCurrentInstance().getExternalContext();
        HttpServletRequest request =
                    (HttpServletRequest)context.getRequest();
        db = (Database) request.getSession().getAttribute("db");
        if (db == null) {
            db = new Database();
            request.getSession().setAttribute("db", db);
            message = "new database generated;";
        }
        else {
            message = "database already exists;";
        }

    }
```

ListBooks.java

```
     private void generateBeanDb(){
        ExternalContext context =
                 FacesContext.getCurrentInstance().getExternalContext();
        HttpServletRequest request =
                    (HttpServletRequest)context.getRequest();
        db = (Database) request.getSession().getAttribute("db");
    }
```

7. Zugriff auf Managed Beans

Beans, die für alle zugänglich sein sollen

zB Lookup-Service, Datenbank-Methoden udgl.

```
@ManagedBean(eager=true)          /* wird gleich zu Beginn instantiiert */
```

```
@ApplicationScoped
public class Database {...}

    public class Login ...{
     @ManagedProperty(value="#{database}")
     private Database database = null;
     ...
     public void setDatabase(Database database){
              this.database = database;
     }
```

8. Session überprüfen und Form – Event

im jsf

```
<h:form id="myform">
     <f:event type="preRenderView"
          listener="#{newBook.verifySession}" />
```

im java

```
public void verifySession()  {

try {
     ExternalContext context = FacesContext.getCurrentInstance().getExternalContext();
     HttpSession session = ((HttpServletRequest)context.getRequest()).getSession();
     if (session.getAttribute("sessionId") == null ||
          ((String)session.getAttribute("sessionId")).compareTo(session.getId())!= 0){
          throw new Exception("---------- session does not fit");
     }
     message = "sessions seems to be ok";
} catch (Exception e) {
     message = " error happened: " + e.getMessage();      }
}
```

oder im java, falls eine andere Seite angesteuert werden soll

```
public void verifySession()  {
…

   if (es passt nicht){
          FacesContext fc = FacesContext.getCurrentInstance();
          ConfigurableNavigationHandler nav
             = (ConfigurableNavigationHandler)
                  fc.getApplication().getNavigationHandler();
          nav.performNavigation("access-denied"); //jsf-page
   }
```

9. Property – File

Idee

Informationen, welche sich nicht zur Laufzeit ändern (übernehmen Beans), sondern

- öfters in der Appl vorkommen
- oder sich irgendwann ändern können (zB Applikation soll in anderer Sprache erscheinen)

sollen via Konfig-Datei wartbar sein.

Property – File erstellen

- .../src/pkgBookstore/bundle/messages.properties
- Schema...key=value
- zB

```
inputname_header=Bücherverwaltung
button_listbooks=Bücherliste
```

Faces-config
Pfad realtiv zu WEB-INF/classes:

```
<resource-bundle>
        <base-name>pkgBookstore.bundle.messages</base-name>
        <var>msg</var>
</resource-bundle>
```

NICHT mehr verwenden den alten Stil: in *.xhtml (wegen Wartbarkeit)

```
<f:loadBundle basename="pkgBookstore.bundle.messages" var="msg"/>
```

XHTML

```
#{msg.button_listbooks}
```

MLS

- <f:view locale="#{facesContext.externalContext.requestLocale}">...erfragt vom Browser die entsprechende Einstellung
- man kann Benutzer am Anfang auch die gewünschte Sprache auswählen lassen

10. Data Table

Voraussetzungen

- damit editable, brauchts Methodenaufrufe **mit Parameter** beim Button
- daher dem TomCat „el-impl-2.2.0.jar“ und „el-api-2.2.0.jar“ in dessen /lib gegeben,
- und die Standard „el-api.jar“ herauslöschen (sonst versteht er die Syntax nicht)
- außerdem ist noch das web.xml zu ergänzen:

```
<context-param>
      <param-name>com.sun.faces.expressionFactory</param-name>
      <param-value>com.sun.el.ExpressionFactoryImpl</param-value>
</context-param>
```

XHTML

```
<h:dataTable value="#{newUsers.users}" var="user">
<h:column>
    <f:facet name="header">Information</f:facet>
        <h:inputText value="#{user.information}" />
</h:column>
<h:column>
    <f:facet name="header">Command</f:facet>
         <h:commandButton value="edit"
        action="#{newUsers.updateUser(user)}" />
</h:column>
</h:dataTable>
```

Java

```
public List<User> getUsers() {
//wird sehr oft hintereinander aufgerufen, vorallem BEVOR
//'updateUser(...)' aufgerufen wird; daher Steuerung mit
,hasChanged'
          if (hasChanged) {
               users.clear();
               try {
                    users.addAll(database.getUsers());
                    message += "some users found";
                    hasChanged = false;
               } catch (Exception e) {
                    message = "error:" + e.getMessage();
               }
          }
```

```
            else
                message = "same procedure as every year";
            return users;
        }
        public String updateUser(User _user) {
            try {
                database.updateUser(_user);
                hasChanged = true;
                message += "update ok";
            } catch (ExceptionBook e) {
                message = "error:" + e.getMessage();
            }
            return "NewUsers";
        }
```

Achtung
Wenn Funktion zum Füllen der Datatable einen Parameter bekommt, dann wird NICHT automatisch „get“ vor den Bezeichner gestellt, sondern der Bezeichner muss dann wie angegeben heißen:
zB:

```
<h:dataTable value="#{listFlights.getFlights(selectCity.city)}"
                        var="flight" border="3">
```

11. AJAX

11.1. Motivation

- wenn ganze Seite auf einmal zu laden ist, muss
- Benutzer derweil warten, bis letztes Byte angekommen ist.

Alternativen

- Applet...kann nicht direkt mit HTML interagieren
- Flash/Flex...nicht auf allen PCs installiert; nicht für iPhone/iPad
- Silverlight...MS-Produkt; Problem:
- JavaFX...

11.2. Überblick

- kleine HTML-Teile werden nachgeladen;
- restliches Formular bleibt (zB Google – Sucheingabe aktualisiert laufend ComboBox)

Vorteil von JSF 2.0

- Client: JSF – Elemente wie „h:outputText“, „h:inputText“, „h:selectOneMenu“
- ohne JavaScript
- Server: Ajax – calls kennen ManagedBeans, dh
- ohne Servlet - Programmierung

11.3. Einfaches Beispiel

```
<h:commandButton value="Update Time" action="nothing">
       <f:ajax render="timeResult"/>
</h:commandButton>
<h:outputText value="#{dateBean.time}" id="timeResult"/>
```

- Wenn auf den Button geklickt wird
- geh zum Server und
- starte die action – Methode, dann
- führe den value – Ausdruck jenes JSF – Elementes mit dem id-Ausdruck
- und ersetzte das JSF-Element im DOM mit dem so berechneten Ausdruck

Anmerkung:

- wenn der value – Ausdruck jedesmal unabhängig zu berechnen ist,
- kann auf die action – Methode verzichtet werden.

11.4. Allgemeine Syntax

```
<h:commandButton … action="…">
       <f:ajax render="id1 id2" execute="id3 id4"
          event="blah" onevent="javaScriptHandler"/>
</h:commandButton>
```

Attribute

- render...jene Elemente, welche neu zu laden sind; meist „h:outputText“; jedenfalls müssen die im gleichen „h:form“ sein.
- execute...jene Elemente, die zwecks Ausführung zum Server gesendet werden; meist „h:inputText“, „h:selectOneMenu“, bei denen die Setter aufgerufen werden
- event...das DOM – Event, auf welches reagiert werden soll; zB „blur“, „keyup“
- onevent...JavaScript-Methode, wenn Event eintritt

Interner Ablauf beim Codieren

- sobald f:ajax verwendet wird,
- wird automatisch ein <script>-Tag im <h:head> eingefügt.
- daher wär nicht schlecht, wenn es <h:head> geben täte.

Syntax – Methode:

```
Ajax: permitted
 public void myActionControllerMethod() { … }

Ajax: preferred
 public String myActionControllerMethod() {
       …
 return(null); // In non-Ajax apps, means to redisplay form
 }
```

Sonstiges

- wenn JavaScript vom Browser deaktiviert,
- werden die Ajax-Buttons zu normale Buttons, dh
- Ablauf derselbe: zuerst action-Methode, dann value-Ausdruck, jedoch
- ganze Seite wird neu geladen (also wie gehabt, ohne Ajax)
- Da JavaScript nicht einheitliche Reaktionen bei den unterschiedlichsten Browsern hervorruft,
- Applikation unbedingt auf unterschiedlichen Browser testen.

11.5. Execute – Attribute

Syntax

```
<f:ajax render="…" execute="…" … />
```

Erklärung:

- @this...default; nur das Element, welches via <f:ajax> markiert ist
- @form...alle Elemente, welche vom <f:form> eingeschlossen sind; Input-Felder brauchen dann keine ids
- @none...keine Elemente werden gesendet
- @all...alle JSF – UI-Elemente werden gesendet (zB wenn es mehrere <f:form> gibt

Beispiel-1 – JSF

```
<h:form>
     <fieldset>
     <legend>Random Number (with execute="someId")</legend>
```

```
Range:
     <h:inputText value="#{numberGenerator.range}"
                  id="rangeField"/><br/>
     <h:commandButton value="Show Number"
                  action="#{numberGenerator.randomize}">
                  <f:ajax execute="rangeField"
                           render="numField3"/>
     </h:commandButton><br/>
<h2>
     <h:outputText value="#{numberGenerator.number}"
                  id="numField3"/>
</h2>
</fieldset>
</h:form>
```

Java

```
@ManagedBean
public class NumberGenerator {
     private double number = Math.random();
     private double range = 1.0;
     public double getRange() {
          return(range);
     }
     public void setRange(double range) {
          this.range = range;
     }
     public double getNumber() {
          return(number);
     }
     public void randomize() {
          number = range * Math.random();
          return(null);
     }
}
```

Ablauf:

- zuerst setRange()
- dann randomize()
- dann getNumber().

Beispiel – 2

```
<h:form>
<fieldset>
     <legend>Bank Customer Lookup </legend>
     Customer ID:
     <h:inputText value="#{bankingBeanAjax.customerId}"/><br/>
     Password:
     <h:inputSecret value="#{bankingBeanAjax.password}"/><br/>
     <h:commandButton value="Show Current Balance"
                  action="#{bankingBeanAjax.showBalance}">
                  <f:ajax execute="@form" render="ajaxMessage1"/>
     </h:commandButton>
     <br/>
     <h2><h:outputText value="#{bankingBeanAjax.message}"
                  id="ajaxMessage1"/></h2>
</fieldset>
</h:form>
```

11.6. Events

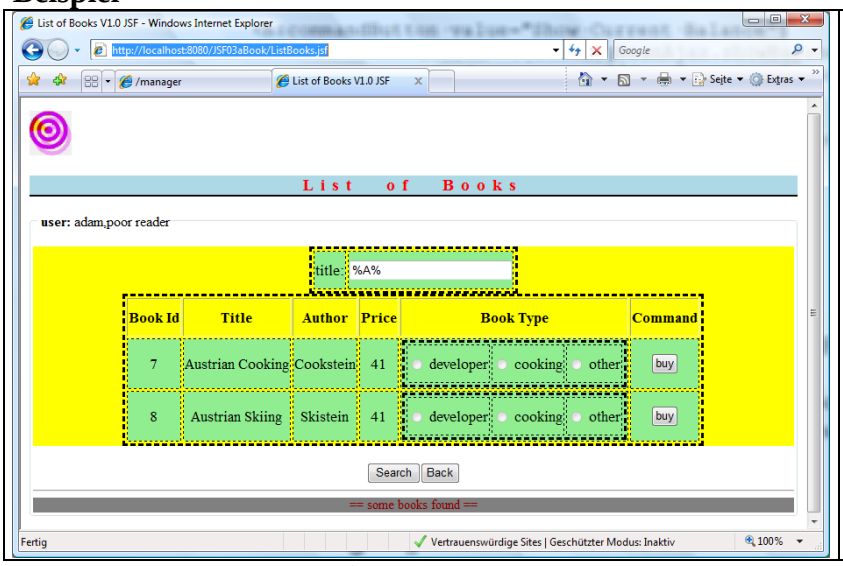
Default Event
action

- h:commandButton, h:commandLink...Button geklickt, bekommt Focus, ShortCut usf.

valueChange

- h:inputText, h:inputSecret, h:inputTextarea, radioButton, checkBox, selectOneMenu...
- ist ein JSF-Event (daher value**C**hange) und daher halbwegs browsereinheitlich

Beispiel



sobald im Title Buchstaben eingetippt werden, läuft im Hintergrund die Suche nach den passenden Büchern.

```
<td>title:</td>
<td><h:inputText  size="25" value="#{listBooks.title}">
     <f:ajax event="keyup" render="bookList"/>
</h:inputText>
</td>
</tr>
</table>
<h:dataTable value="#{listBooks.books}" var="book" border="3"
     id="bookList">
<h:column>
     <f:facet name="header">Book Id</f:facet>
          <h:outputText value="#{book.id}" />
</h:column>
```

12. Sonstiges

12.1. Zugriff auf Collections und Arrays

page.xhtml

```
<li>#{purchases.mediumItems[0]}</li>
<li>#{purchases.mediumItems[1]}</li>
<li>#{purchases.mediumItems[2]}</li>
<li>Low: #{purchases.valuableItems["low"]}</li>
<li>Medium: #{purchases.valuableItems["medium"]}</li>
<li>High: #{purchases.valuableItems["high"]}</li>
```

Purchases.java

```
public String[] getCheapItems() {
  return(cheapItems);
}
public List<String> getMediumItems() {
  return(mediumItems);
}
```

```
public Map<String,String> getValuableItems() {
 return(valuableItems);
}
```

12.2. Vordefinierte Variablen

Grundsätzlich möglichst vermeiden, da mit MVC dann nichts mehr am Hut;
Datenmanipulationen und Ablaufsteuerung sollen im Java-Code erfolgen;

```
param. Request params.
• E.g. #{param.custID}
– cookie. Cookie object (not cookie value).
• E.g. #{cookie.userCookie.value} or #{cookie["userCookie"].value}
– request, session
• #{request.queryString}, #{session.id}
```