JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Temperature Converter - JSF

SEW
4AHITM 2015/16

Özsoy Ahmet

Betreuer: Michael Borko
Dominik Dolezal

Version 1.0
Begonnen am 06.06.2016
Beendet am 06.06.2016

1 Einführung

1.1 Aufgabenstellung

Folge der Anleitung und erstelle einen Temperatur-Konverter und erweitere das Projekt, sodass auch Grad Kelvin unterstützt werden!
Abgabe: Protokoll (inkl. Kopf-und Fußzeile, Screenshots, Ergebnis, ...) + Code auf GitHubpushen.

2 Aufwandschätzung

Aufgabe	Geschätzter Aufwand
Temperatur Converter zum Laufen bringen JSF Aufgabe	30 min
Dokumentation	30 min
Summe	**60 min**

3 Github-Link

Link : https://github.com/aoezsoy-tgm/JSF_TEMP_CONV

4 Arbeitsdurchführung

Die Arbeit wurde in IntelliJ IDEA durchgeführt.

4.1 Ein Gradle Projekt importieren

Auf File – New – Project from Existing Sources

4.2 Erster Schritt

Dann den Pfad auswählen und auf OK drücken.

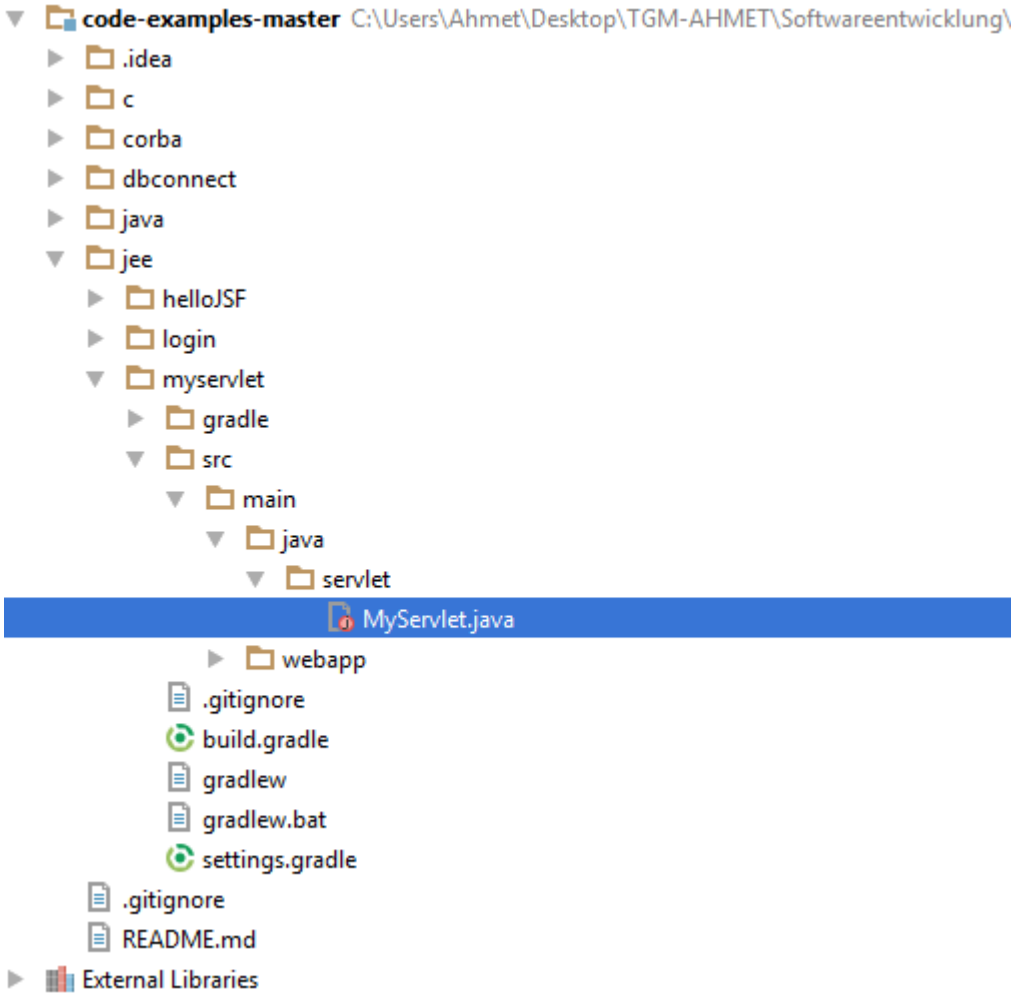
4.3 Zweiter Schritt

Von Import-Projekt wizards, wählen wir Gradle aus und drücken au NEXT.

4.4 Dritter Schritt

Im Fenster des wizards, die Gradl Project Settings spezialisieren und das global Gradle Setting. Danach auf FINISH drücken.

5 Dynamische Webprojekte



Das Servlet bearbeitet die HTTP-Get bzw. HTTP-Post requests und bietet einen HTTP-Response an:

```
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request,
                HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request,
                HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        String output=
        "<html><body><h1>Welcome!</h1>";
```

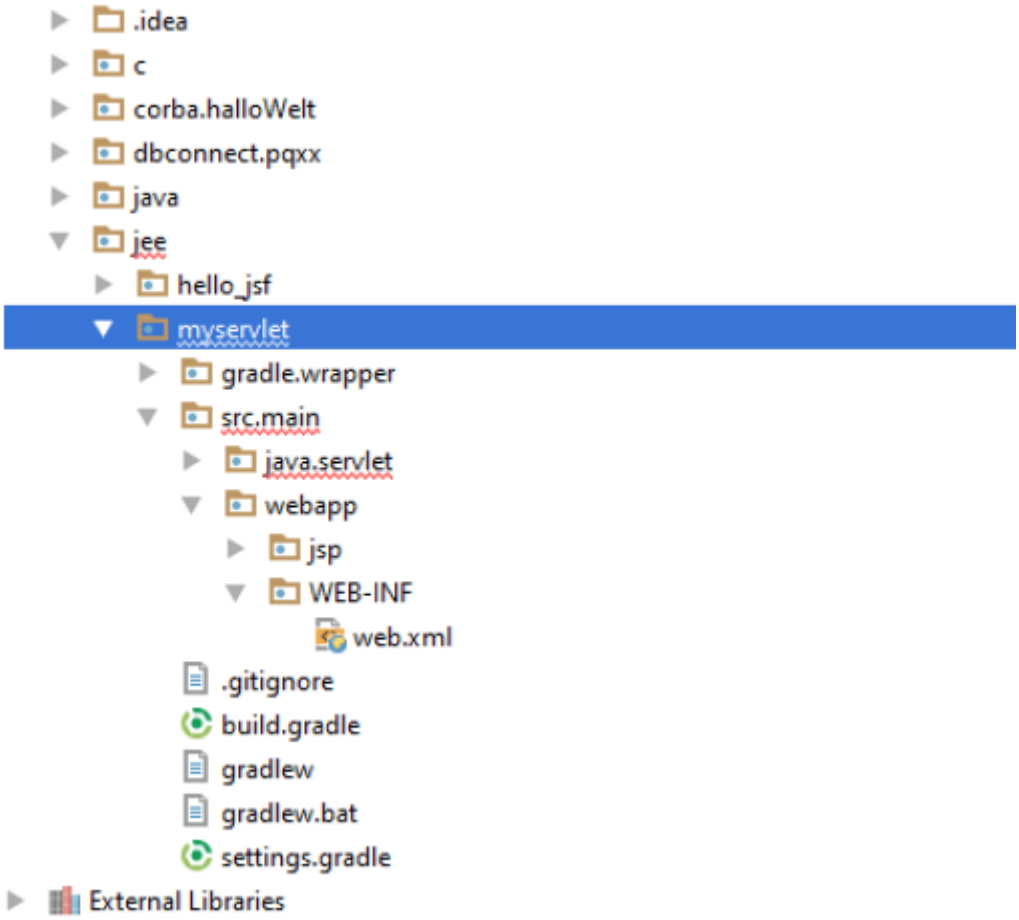
Für die Buildautomation mittels Gradle werden nun Repositories und Dependencies definiert:

```
apply plugin: 'java'
apply plugin: 'war'
apply plugin: 'jetty'
apply plugin: 'eclipse'
repositories {
jcenter()
}
dependencies {
    compile 'javax.servlet:javax.servlet-api:3.1.0
    runtime 'javax.servlet:jstl:1.2'
}

jettyRun {
    httpPort = 8080
    contextPath = 'myservlet'
}
```

Dadurch wird das Skript auf allen Plattformen kompilierbar und startbar.

6 Java Server Pages



Auch JSP-Files können wie Servlets verwendet werden. Wobei die Zuordnung im Web-Deskriptor festgelegt wird.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSch
3    <display-name>MyJSP</display-name>
4      <servlet>
5          <servlet-name>testjsp</servlet-name>
6          <jsp-file>/example.jsp</jsp-file>
7          <load-on-startup>0</load-on-startup>
8      </servlet>
9
10     <servlet-mapping>
11         <servlet-name>testjsp</servlet-name>
12         <url-pattern>*.jsp</url-pattern>
13         <url-pattern>*.jspf</url-pattern>
14         <url-pattern>*.jspx</url-pattern>
15         <url-pattern>*.xsp</url-pattern>
16         <url-pattern>*.JSP</url-pattern>
17         <url-pattern>*.JSPF</url-pattern>
18         <url-pattern>*.JSPX</url-pattern>
19         <url-pattern>*.XSP</url-pattern>
20     </servlet-mapping>
21 </web-app>
```

Ein <jsp-file> Tag ersetzt den <servlet-class> Tag. Zusätzlich sollte noch ein <servlet-mapping> definiert werden.

Für die Buildautomation mittels Gradle werden Repositories und Dependencies definiert.

Für die Ausführung ist nur JavaServer Pages Standard Tag Library (JSTL) notwendig.
Folgende Ausgabe kann bei einem beliebigen URL-Muster *.jspf (in diesem Fall hello.jspf) erhalten werden.

Das JSP-File enhält html-Code, JSTL-Tags und Java-Code:

```
1 <html>
2 <head>
3 <title>Java Code Snippet - Sample JSP Page</title>
4 <meta>
5 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
6 </meta>
7 </head>
8 <body>
9     <c:out value="Jetty JSP Example"></c:out>
10    <%-- This is a JSP comment --%>
11    <br />
12    Current date is: <b><%=(new java.util.Date()).toLocaleString()%></b>
13 </body>
14 </html>
```

Unabhängig davon können auch html-Dateien vorhanden sein, welche nicht von den url-pattern überlagert sind.

7 JSP und MVC

Gerade in Applikationen im Zusammenhang mit JEE ist das Design Pattern MVC besonders ausgeprägt.

Zentrales Element ist hier wieder das Servlet, welches mittels

web.xml definiert wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://jav
  <display-name>MyServlet</display-name>
  <welcome-file-list>
    <welcome-file>/index.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>servlet.MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyServlet</servlet-name>
    <url-pattern>Select.do</url-pattern>
  </servlet-mapping>
</web-app>
```

Das Servlet wird hier auf das Url-Pattern Select.do horchen.

Umso flexible wie möglich zu sein, wird häufig das Servlet HTML-Get-und HTML-Post-Requests zusammenfassen.

```
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /* (non-Javadoc)
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        this.process(request, response);
    }

     * (non-Javadoc)
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        this.process(request, response);
    }
     * process html get and post requests
    private void process(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String wish = request.getParameter("wish");
        String state = request.getParameter("state");
        RequestDispatcher view;
        if (wish.equals("one state")) {
            // get a capital
            view = request.getRequestDispatcher("index2.jsp");
```

Das Modell hält die Daten bzw. die notwendigen Auswertungen bereit:

```
public class Bundesland implements java.io.Serializable{
    /**..
    private static final long serialVersionUID = 1L;
    private static Map<String, String> HS;

    static {..
    public static String[] getAllCapitals(){
        return HS.values().toArray(new String[0]);
    }
    public static String[] getAllStates(){
        return HS.keySet().toArray(new String[0]);
    }
    private static String[] getCapital(String state){
        return new String[]{"Die Hauptstadt von "+state+" lautet "+
                HS.get(state)+"!"};
    }
    public static String[] decide(String wish, String state){
        switch(wish){
        case "all capitals": return getAllCapitals();
        case "capital": return getCapital(state);
        case "all states": return getAllStates();
        }
        return null;
    }
```

Die dynamische View wird mittels JSP erstellt.

8 Java Server Faces

Gerade in Applikationen im Zusammenhang mit JEE ist das Design Pattern MVC besonders ausgeprägt.

Folgendes Beispiel soll das Zusammenspiel zwischen Forms in xhtmlund JSF (PrimeFaces) und einem Model im Blickpunkt von MVC zeigen.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.
  <display-name>Temperature Convertor</display-name>

    <listener>
        <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
    </listener>

    <!-- File(s) appended to a request for a URL that is not mapped to a
        web component -->
    <welcome-file-list>
        <welcome-file>convertor.xhtml</welcome-file>
    </welcome-file-list>

    <!-- Define the JSF servlet (manages the request processing lifecycle
        forJavaServer) -->
    <servlet>
        <servlet-name>faces</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Map following files to the JSF servlet -->
    <servlet-mapping>
        <servlet-name>faces</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
</web-app>
```

Im Deployment-Deskriptor wird das Servlet inkl. Mapping festgelegt. Diese Inhalte sind für reine JSF-Projekte immer fix.

Nur das welcome-file wird hier eingetragen.

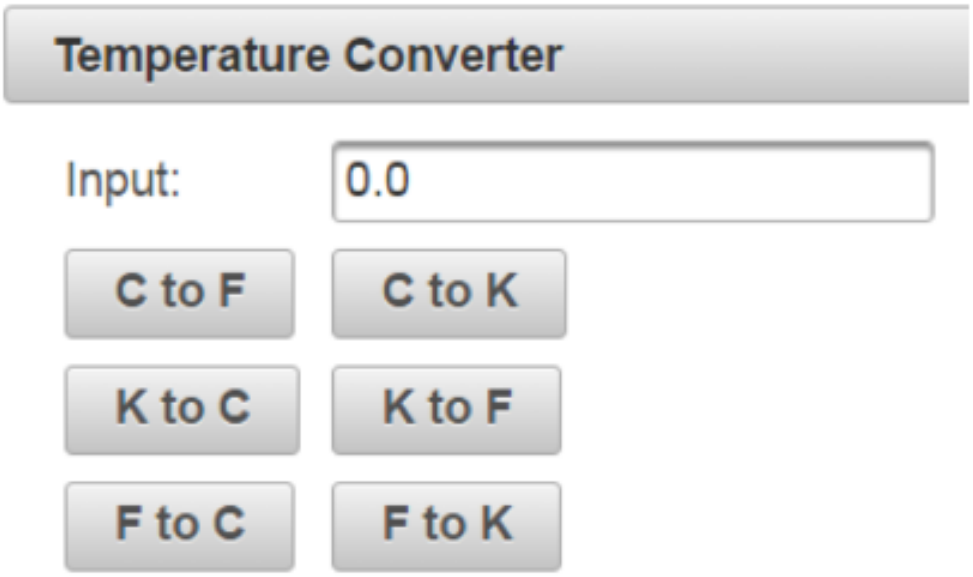
Zusätzlich muss dem faces-Servlet alle verwendeten POJO's (Beans) bekanntgegeben werden:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
    version="2.2">

    <managed-bean>
      <managed-bean-name>temperatureConvertor</managed-bean-name>
      <managed-bean-class>model.TemperatureConvertor</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
     </managed-bean>
</faces-config>
```

Hier wird der Name, die Klasse (inkl. Package) und der Scope festgelegt.

Das Model ist hingegen sehr einfach gehalten.



```
public void celsiusToFahrenheit(){
    this.initial = false;
    this.unit = "Fahrenheit";
    this.converted= (convert * 1.8) +32;
}
public void fahrenheitToCelsius(){
    this.initial = false;
    this.unit = "Celsius";
    this.converted = (convert - 32) / 1.8;
}
public void kelvinToFahrenheit(){
    this.initial = false;
    this.unit = "Fahrenheit";
    this.converted= (convert * 1.8) - 459.67;
}
public void kelvinToCelsius(){
    this.initial = false;
    this.unit = "Celsius";
    this.converted = convert - 273;
}
public void fahrenheitToKelvin(){
    this.initial = false;
    this.unit = "Kelvin";
    this.converted= (convert + 459.67) /1.8;
}
public void celsiusToKelvin(){
    this.initial = false;
    this.unit = "Kelvin";
    this.converted = convert +273;
}
```