

---

# **Laborprotokoll**

## **JSF-Temperatur**

---

**SEW Übung**  
**4AHITM 2015/16**

**Kanyildiz Muhammedhizir**

**Note:**

**Betreuer: Prof. Dolezal**

**Version 1.0**

**Begonnen am 1.Juni 2016**

**Beendet am 6.Juni 2016**

## Inhaltsverzeichnis

1	Einführung .....	3
1.1	Ziele .....	3
1.2	Voraussetzungen .....	3
1.3	Aufgabenstellung .....	3
2	Gradle-projekt Import in .....	4
2.1	Schritt 1 .....	4
2.2	Schritt 2 .....	4
2.3	Schritt 3 .....	4
2.4	Schritt 4 .....	4
3	Dynamische Webprojekte .....	5
3.1	Servlets .....	5
3.2	Java Server Pages .....	7
3.3	JSP und MVC .....	9
3.4	Java Server Faces .....	12
4	Aufwandsabschätzung .....	15
5	Literaturverzeichnis .....	15

# 1 Einführung

## 1.1 Ziele

Mit dieser Aufgabe soll, das bessere auskennen und das „selbständige“ programmieren mittels Gradle erzielt werden.

## 1.2 Voraussetzungen

- Gradle Installation
- Gradle Grundkenntnisse
- Aufpassen im Unterricht

## 1.3 Aufgabenstellung

Folge der Anleitung und erstelle einen Temperatur-Konverter und erweitere das Projekt, sodass auch Grad Kelvin unterstützt werden!

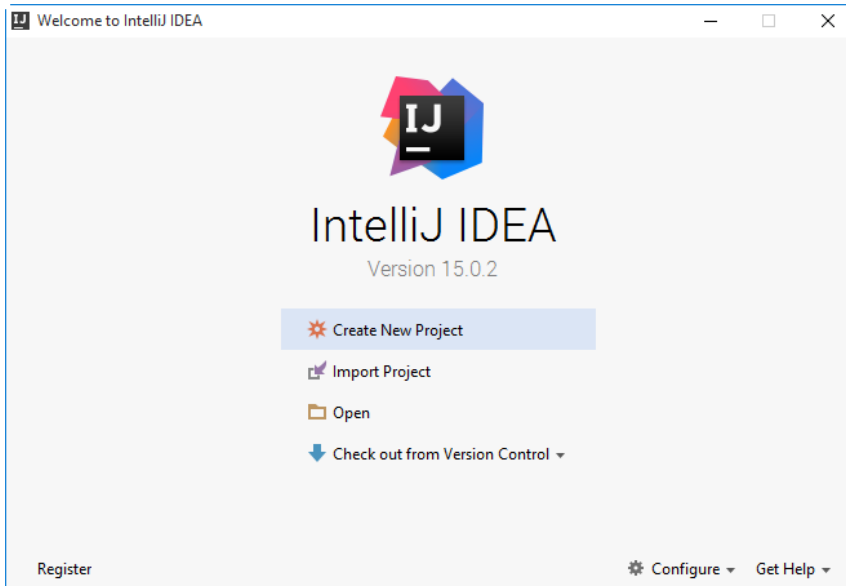
Abgabe: Protokoll (inkl. Kopf- und Fußzeile, Screenshots, Ergebnis, ...) + Code auf GitHub pushen.

## 2 Gradle-projekt Import in

### 2.1 Schritt 1

Wenn derzeit keine Projekte geöffnet sind.

Click auf Import Projekt auf dem Welcome screen



Ansonsten, klicken Sie auf File | New | Project from Existing Sources

### 2.2 Schritt 2

Dann sollte sich ein Dialog Fenster öffnen, wählen Sie den Pfad aus, indem sich Ihr Projekt befindet. Dann drücken Sie auf OK.

### 2.3 Schritt 3

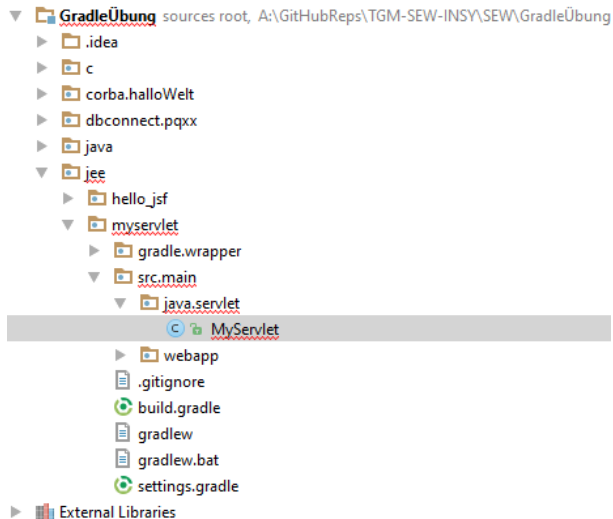
Auf dem ersten Fenster des Import Projekt wizards, wählen Sie Gradle aus und drücken auf NEXT.

### 2.4 Schritt 4

Auf dem nächsten Fenster des wizards, spezialisieren Sie die Gradl Project Settings und global Gradle Setting. Danach drücken Sie auf FINISH.

## 3 Dynamische Webprojekte

### 3.1 Servlets



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:

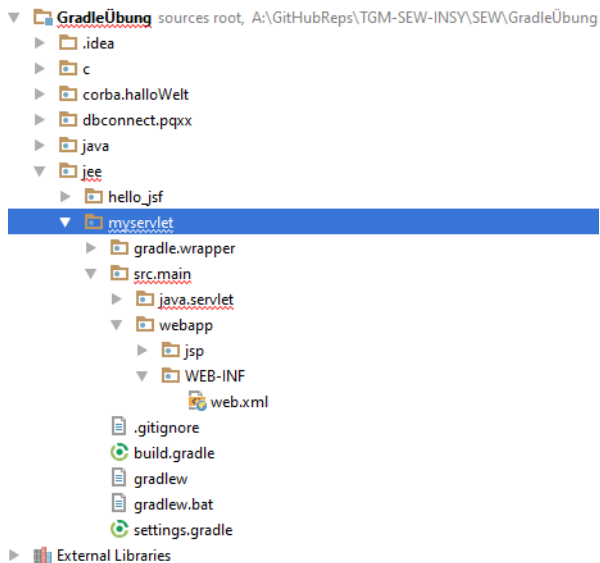
```
apply plugin: 'java'
apply plugin: 'war'
apply plugin: 'jetty'
apply plugin: 'eclipse'
repositories {
    jcenter()
}
dependencies {
    compile 'javax.servlet:javax.servlet-api:3.1.0'
    runtime 'javax.servlet:jstl:1.2'
}

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

Dadurch wird das Skript auf allen Plattformen kompilierbar und startbar. Ebenso wird bei der Konvertierung in ein IntelliJprojekt die notwendigen Libraries mit erstellt. Das web.xml kann in IntelliJ sehr einfach im Design- oder Sourcemode bearbeitet werden.

Durch die Ausführung erhält man nun eine etwas dynamischere Webseite.

## 3.2 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/XMLSchema
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>*.jspx</url-pattern>
14         <url-pattern>*.jpf</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. In diesem Fall werden eine große Zahl an Muster an das „Servlet“ weitergeleitet.

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

```
apply plugin: 'war'
apply plugin: 'jetty'
apply plugin: 'eclipse'

repositories {
    jcenter()
}

dependencies {
    runtime 'javax.servlet:jstl:1.2'
}

jettyRun {
    httpPort = 8080
    contextPath = '/jsp'
}
```

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 enthält 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.

Jetty verwendet hier ein Servlet-Service von Apache (Jasper), welches erst bei Exception (z.b. Syntaxfehlern im .jsp-File) sichtbar wird:

#### HTTP ERROR 500

Problem accessing /jsp/e.jsp. Reason:

FWC6033: Unable to compile class for JSP

FWC6197: An error occurred at line: 12 in the jsp file: /example.jsp  
FWC6199: Generated servlet error:  
Syntax error on token "=", delete this token

FWC6197: An error occurred at line: 12 in the jsp file: /example.jsp  
FWC6199: Generated servlet error:  
Syntax error, insert ";" to complete Statement

#### Caused by:

org.apache.jasper.JasperException: FWC6033: Unable to compile class for JSP

FWC6197: An error occurred at line: 12 in the jsp file: /example.jsp  
FWC6199: Generated servlet error:  
Syntax error on token "=", delete this token

FWC6197: An error occurred at line: 12 in the jsp file: /example.jsp  
FWC6199: Generated servlet error:  
Syntax error, insert ";" to complete Statement

```
at org.apache.jasper.compiler.DefaultErrorHandler.javaError(DefaultErrorHandler.java:123)
at org.apache.jasper.compiler.ErrorDispatcher.javaError(ErrorDispatcher.java:296)
at org.apache.jasper.compiler.Compiler.generateClass(Compiler.java:376)
at org.apache.jasper.compiler.Compiler.compile(Compiler.java:437)
at org.apache.jasper.JspCompilationContext.compile(JspCompilationContext.java:608)
at org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:360)
at org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:466)
at org.apache.jasper.servlet.JspServlet.service(JspServlet.java:380)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:820)
at org.mortbay.jetty.servlet.ServletHolder.handle(ServletHolder.java:511)
```



### 3.3 JSP und MVC

Gerade in Applikationen im Zusammenhang mit JEE ist das Design Pattern MVC besonders ausgeprägt. Folgendes Beispiel soll das Zusammenspiel zwischen statischen Forms, einem Servlet, mehrerer JSP's und einem Model im Blickpunkt von MVC zeigen.

## Wähle eine Thema aus

Lerne Dein Land kennen

Thema:

alle Hauptstädte

alle Bundesländer

eine Hauptstadt

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 nun wieder mittels JSP erstellt.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@page import="java.util.*"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; char
    <title>Die L&ouml;sung!</title>
  </head>
  <body>
    <h1 align="center">Politische Bildung JSP</h1>

    <!-- Skriptlet -->
    <h2><%= request.getParameter("wish") %></h2>

    <!-- And now with JSTL!! -->
    <c:forEach items="${result}" var="item">
      ${item}<br/>
    </c:forEach>
    <p>

    <a href="/bl">let's start again...</a>

  </body>
</html>
```

### 3.4 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 XHTML und 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.

Achtung: Für Jetty muss unbedingt der ConfigurationListener als Listener bekannt gemacht werden!

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.

Nun folgt der View-Part (GUI) als XHTML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1" [
  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core" xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:p="http://primefaces.org/ui">

    <h:head>
      <title>Temperature Convertor</title>
    </h:head>

    <h:body>

      <h:form id="form">
        <p:panel header="Temperature Convertor">

          <!-- <p:growl id="growl" life="2000" /> -->
          <p:panelGrid columns="2">
            <h:outputText value="Temp:" />
            <p:inputText value="#{temperatureConvertor.convert}" />
          </p:panelGrid>
          <br />
          <p:commandButton value=" C->F " ajax="false"
            action="#{temperatureConvertor.celsiusToFahrenheit}" />
          <p:commandButton value=" F->C " ajax="false"
            action="#{temperatureConvertor.fahrenheitToCelsius}" />
          <p:commandButton value="Reset" ajax="false"
            action="#{temperatureConvertor.reset}" />

        </p:panel>
      </h:form>
      <p:panel id="result" header="Result"
        rendered="#{not temperatureConvertor.initial}">
        <h:outputLabel value="#{temperatureConvertor.unit}" />
        <h:outputLabel value="#{temperatureConvertor.converted}" />
      </p:panel>
    </h:body>
  </html>
```

Mittels Expression Language (EL) wird auf die Attribute und Methoden der ManagedBean (temperatureConvertor) zugegriffen.

Die Instanziierung erfolgt entsprechend dem vereinbarten Scope vom faces-Servlet. Da es sich um ein POJO handelt werden je nach Notwendigkeit die setter bzw. getter-Methoden aufgerufen.

Ausnahme sind die actions im commandButton, welche direkt als Methodennamen interpretiert werden!

Das Attribut rendered erwartet den Typ boolean welcher wieder mittels getter-Methode ausgelesen wird. Hier erfolgt zusätzlich noch eine Negation.

Somit kann das Result-Panel ein- bzw. und ausgeschaltet werden.

Das Model ist hingegen sehr einfach gehalten:

```
public class TemperatureConvertor implements Serializable {
    private static final long serialVersionUID = 1L;
    private double convert;
    private double converted;
    private boolean initial;
    private String unit;

    public double getConvert() {
        return convert;
    }

    public void setConvert(double convert) {
        this.convert = convert;
    }

    public double getConverted() {
        return converted;
    }

    public String getUnit() {
        return unit;
    }

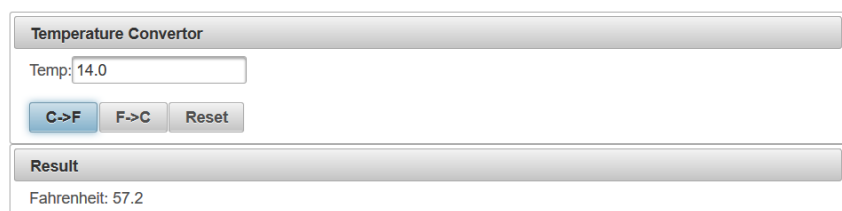
    public boolean getInitial() {
        return initial;
    }

    public void init(){
        initial = true;
        converted = 0;
        convert = 0;
        unit="";
    }
    public String reset() {
        init();
        return "reset";
    }

    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;
    }
}
```



## 4 Aufwandsabschätzung

Arbeit	Stundenabschätzung
JSF Aufgabe	2:30 h – 3:00 h
Dokumentation	0:15 h – 0:30 h

## 5 Literaturverzeichnis

GITHUB LINK
<a href="https://github.com/mkanyildiz01/TGM-SEW-INSY.git">https://github.com/mkanyildiz01/TGM-SEW-INSY.git</a>