2023

# Tomcat, JSP und Java Servlets

## WHAT'S ALL ABOUT IT?

ELEUTHERIOS ATHANASSIOU

# Tomcat, JavaServerPages und Servlets

## Inhaltsverzeichnis

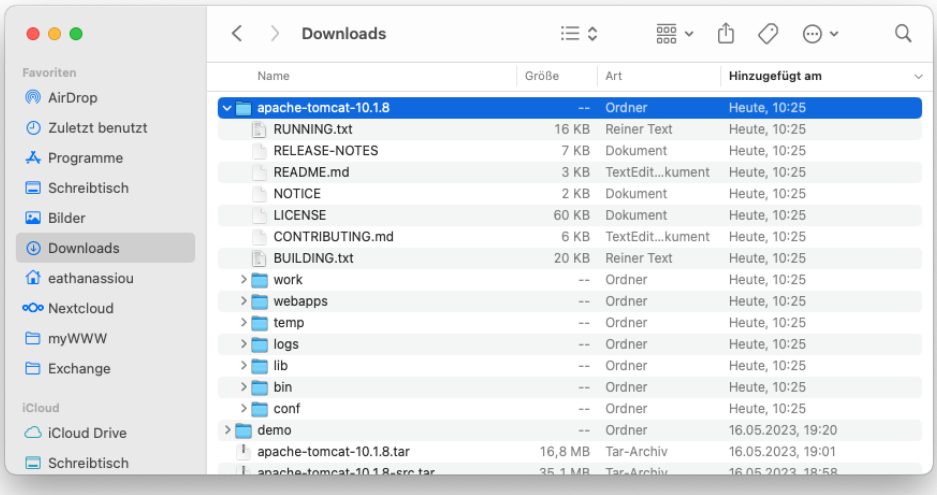
# 1. TOMCAT

## 1.1 Installation auf MacOS

1. **Download** einer Binärdistribution des Kernmoduls Apache-Tomcat-10.1.x von https://tomcat.apache.org/download-10.cgi.
   Dort habe ich die Datei tar.gz im Abschnitt „Binary Distributions Core“ ausgewählt.

2. Durch **Öffnen /Entpacken des Archivs** im Download Ordner. Es wird eine neue Ordnerstruktur in dem Download-Ordner erstellt:

   ~/Downloads/Apache-Tomcat-10.1.8




3. Starten einer Terminal-App, um diese **Ordnerstruktur** nach /usr/local zu **verschieben**:

   ```
   sudo mkdir -p /usr/local
   sudo mv ~/Downloads/apache-tomcat-10.1.8 /usr/local
   ```

4. Um es künftig einfacher zu haben diese Version durch neue zu ersetzen, wird ein **symbolischer Link** erstellt. In dem wird der versionsabhängige Teilpfad substituiert:

   ```
   sudo rm -f /Library/Tomcat
   sudo ln -s /usr/local/apache-tomcat-10.1.8 /Library/Tomcat
   ```

5. **Ändern des „Besitzers**“ der Ordnerhierarchie /Library/Tomcat:

   ```
   sudo chown -RL $USER /Library/Tomcat
   ```

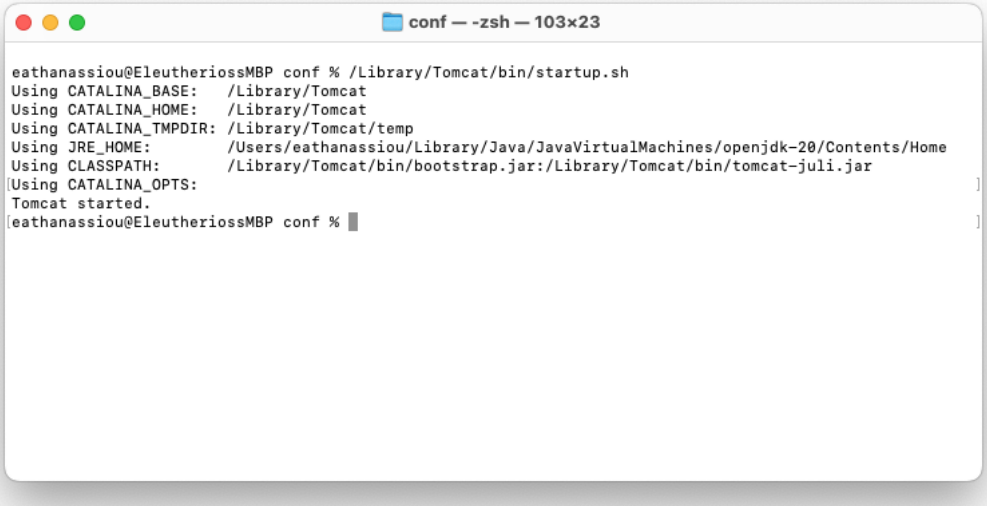
6. Alle **Scripte** in Tomcat's ./bin Ordner **ausführbar** machen:

   ```
   sudo chmod +x /Library/Tomcat/bin/*.sh
   ```

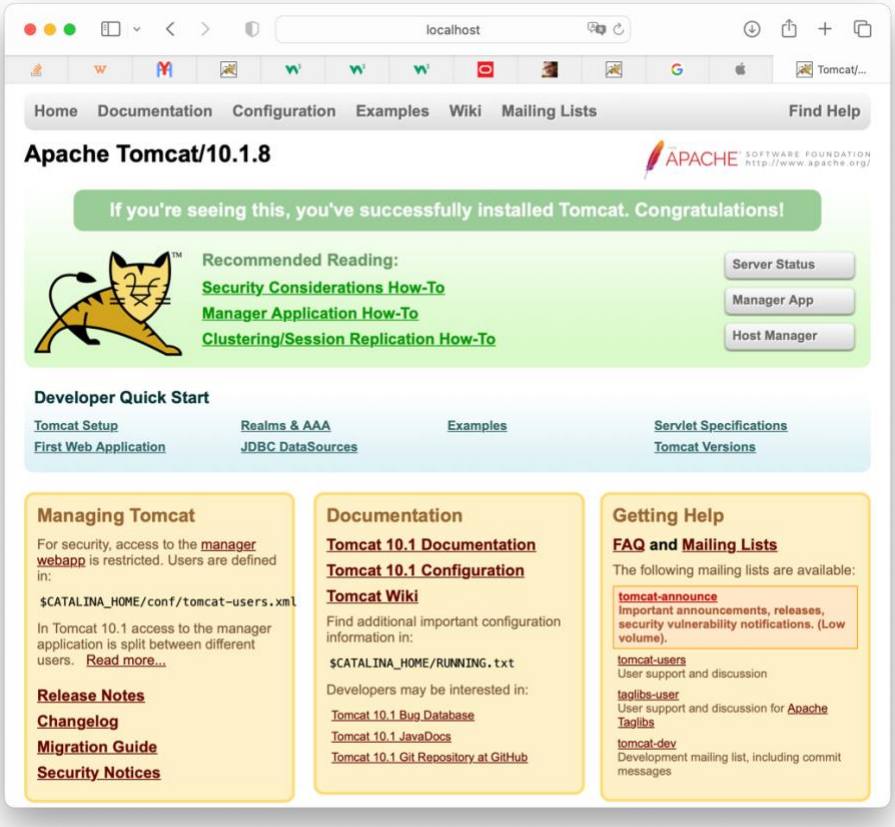
## 1.2 Starten von Tomcat 10.1.x

Tomcat wird gestartet durch das Ausführen des bereitgestellten Skript startup.sh :

```
/Library/Tomcat/bin/startup.sh
```



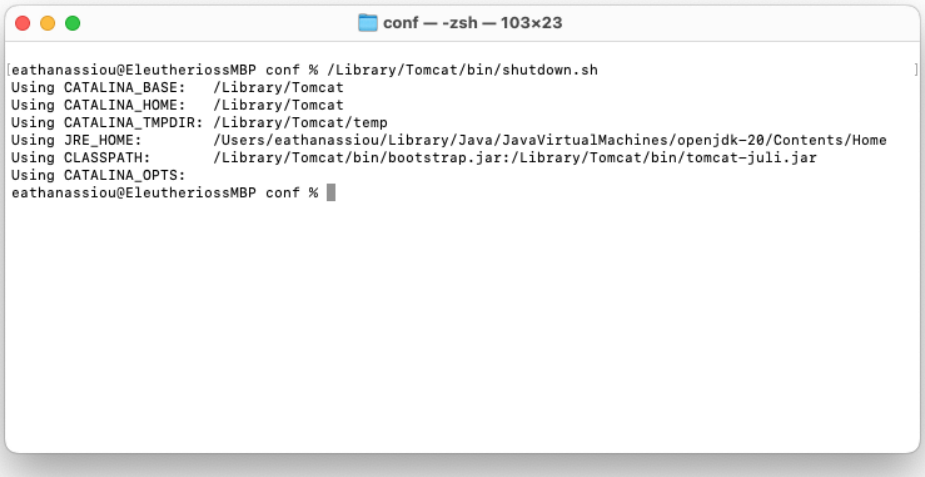

Nachdem Tomcat gestartet wurde, kann man im Webbrowser des Macs einen Blick auf die Standardseite: http://localhost:8080 werfen.




## 1.3 Stoppen von Tomcat 10.1.x

Tomcat wird gestoppt durch das Ausführen des bereitgestellten Skript shutdown.sh :

```
/Library/Tomcat/bin/shutdown.sh
```




## 1.4 Tomcat Verzeichnisse

### 1.4.1 Systemverzeichnisse

- **/bin** - Startup, shutdown, und weitere scripts.
- **/conf** - Konfigurationsdateien und zugehörige DTDs. Die wichtigste Datei hier ist server.xml. Es ist die Hauptkonfigurationsdatei für den Container.
- **/logs** - Protokolldateien sind her standardmäßig drin.

### 1.4.2 webapps

```
/Library/Tomcat/webapps
```

Das Verzeichnis /webapps ist der Ort, an dem sich Anwendungen in Tomcat befinden. Es ist der Standardort für die Deployments, das kann jedoch konfiguriert werden. Sobald es mehrere Anwendungen gibt, empfiehlt es sich weitere Subdirectories anzulegen, jeweils mit dem Namen der Anwendung.

Beispiel: /Library/Tomcat/webapps/examples

### 1.4.3 /ROOT

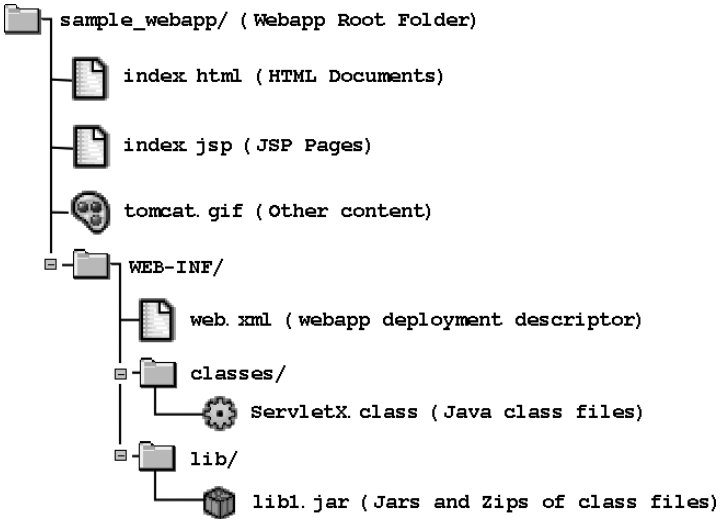
```
/Library/Tomcat/webapps/ROOT/
```

Falls der Request wie zum Beispiel: localhost:8080/index.jsp keine weiteren Verzeichnisse involviert, sucht Tomcat immer im ROOT Verzeichnis. In ROOT liegen in gewissem Sinne die namenlosen Web-Applikationen.

## 1.5 Layout einer Web-Anwendung

Tomcat setzt die Servlet- und JSP-Spezifikationen als Teil von Sun's Java EE um. Java EE wurde entwickelt, um Entwicklern das Verschieben von Anwendungen zwischen J2EE-Applikationsserver zu ermöglichen, ohne dass sie umgeschrieben werden müssen.
Dazu werden Anwendungen auf portable Weise verpackt, und zwar im **Web Application aRchive (WAR)** Dateiformat und dessen Dateistruktur.

Damit die Web-Applikation unabhängig vom verwendeten Applikationsserver ist, müssen ihre Dateien Konventionen entsprechen, v.a. dem Verzeichnislayout zum Speichern von Webseiten, Konfigurationsdateien usw. Dieses allgemeine Layout ist unten dargestellt:




Webseiten (ob statisches HTML, JSP oder andere dynamische Vorlagensprachen) können also im Stammverzeichnis einer Webanwendung oder in fast jedem beliebigen Unterverzeichnis abgelegt werden, mit Ausnahme von WEB-INF oder META-INF.

Bilder sind oft in einem /images-Unterverzeichnis, wobei dies eine Konvention ist, keine Voraussetzung.

### 1.5.1 /WEB-INF

Das /WEB-INF-Verzeichnis enthält mehrere spezifische Inhalte. Es enthält Unterverzeichnisse zum Speichern von

- kompilierten Java-Klassen /class
  - der Ort, an dem alle verwendeten Java-Klassendateien abzulegen sind, die sich nicht in einer JAR-Datei befinden, unabhängig davon, ob es sich um Servlets oder andere Klassendateien handelt.

- JAR-Dateien (Libraries) /lib
  - der Ort, an dem alle JAR-Dateien abgelegt werden, die packages von Klassen enthalten

und dem

- *„Deployment descriptor“* web.xml
  - der die Konfiguration,
  - eine Beschreibung
  - zusätzliche Anpassungen

  der Webanwendung enthält.

web.xml ist eine XML-Datei, die die zugrundeliegenden Servlets parametrisiert und konfiguriert, aus denen die Anwendung besteht, zusammen mit allen Initialisierungen von

Parametern und Sicherheitsbeschränkungen, die der Server erzwingen soll. Hier wird alles über die Anwendung definiert, was ein Server wissen muss.

Im Kapitel 3.7 geh ich etwas näher auf die Datei ein.

WEB-INF und META-INF sind automatisch vor dem Zugriff durch Client-Webbrowser geschützt, da diese Dateien (die Kontonamen und Passwörter enthalten können) für den Client uneinsehbar sind.

```
/Library/Tomcat/webapps/examples/WEB-INF
/Library/Tomcat/webapps/examples/WEB-INF/web.xml
/Library/Tomcat/webapps/examples/WEB-INF/class
/Library/Tomcat/webapps/examples/WEB-INF/lib
```

### 1.5.2 /META-INF

Im Verzeichnis /META-INF kann man eine Datei context.xml hinterlegen.

Wenn im Zusammenhang mit Tomcat der Begriff context auftaucht, dann ist damit eine Web-Anwendung gemeint. Da ein ernsthafter Tomcat durchaus mehrere Web-Anwendungen hostet, macht es Sinn mehrere „Kontexte“ zu unterscheiden.

context.xml kann verwendet werden, um für jede Web-Anwendung Tomcat-spezifische Konfigurationsoptionen zu definieren, z. B. Zugriffsprotokolle, Datenquellen, die Konfiguration des Sitzungsmanagers und viele mehr. Manche sagen, die Datei sollte man als Entwickler nicht weiter anschauen und einfach in Ruhe lassen.

/Library/Tomcat/webapps/examples/META-INF
/Library/Tomcat/webapps/examples/META-INF/context.xml

## 1.6 *.WAR Dateien

WAR (kurz für Web Archive) ist die Extension einer Datei, die eine Verzeichnishierarchie im obigen Layout für Webanwendungen im ZIP-Format verpackt. Java-Webanwendungen werden in der Regel als WAR-Dateien für das Deployment verpackt.

Diese Dateien können über die Befehlszeile oder mit einer IDE erstellt werden.
In der Befehlszeile nutzt man das JAR-Tool von JDK, um im Top-Level des gemäß dem oben beschriebenem Layout angelegten Projektverzeichnis mit

```
jar -cvf projektname.war *
```

ein WAR Datei mit dem Name *projektname* zu erzeugen.

Die Parameter -cvf bedeuten:

- -c Datei erstellen (create),
- -v ausführliche Ausgabe zu generieren (verbose)
- -f Namen der Archive-Datei (file).
- * alle Dateien dieses Verzeichnisses (einschließlich Unterverzeichnis) verwenden

Weitere relevante Befehlsvariationen sind:

```
jar -xvf projektname.war
```

entpackt eine WAR Datei mit dem Name *projektname* .

```
jar -tf projektname.war
```

zeigt den Inhalt einer WAR Datei mit dem Name *projektname* an.

## 1.7 Deployment von Web-Anwendungen in den Tomcat

Es gibt mehrere Optionen, und je nach verwendeter IDE gibt es weitere Wege um ein Artefakt zu deployen. Hier wird auf die Tomcat Tools auf das Deployment mit intellij UE eingegangen.

### 1.7.1 Tomcat Tools

Zur Ausführung (auch während der Entwicklung) muss eine Web-Anwendung deployed werden. Dazu müssen die Anwendungsdateien im obigen Layout einer Verzeichnisstruktur abgelegt sein. Sie kann in den Tomcat mit einem der folgenden Ansätze deployed werden.:

- **Kopieren der Verzeichnishierarchie** der Anwendung in **ein Unterverzeichnis von $CATALINA_BASE/webapps/**

  Tomcat weist der Anwendung einen Kontextpfad zu, der auf dem Namen des Unterverzeichnis basiert. Dies ist der der schnellste und einfachste Ansatz während der Entwicklung. Tomcat sollte hiernach neu gestartet werden.

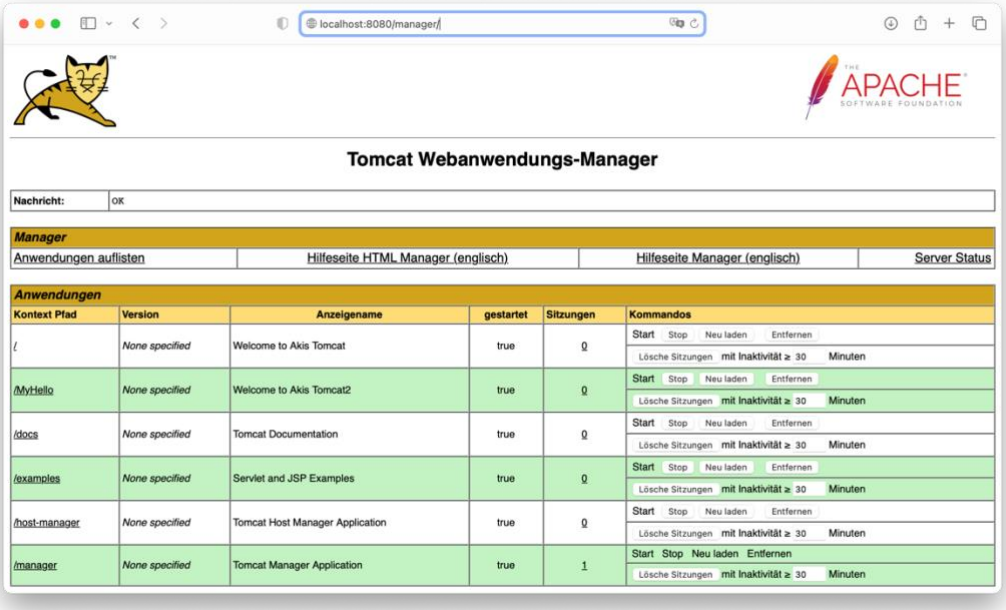
- **Erstellen einer WAR Datei** und Kopieren der WAR Datei (mit der gepackten Webanwendung) **in das Verzeichnis $CATALINA_BASE/webapps/**

  Tomcat überwacht den Inhalt des Verzeichnisses und extrahiert bei einer neuen *.war*-Datei diese in das Verzeichnis webapps/war-datei. Der Vorgang wird Hot-Deployment

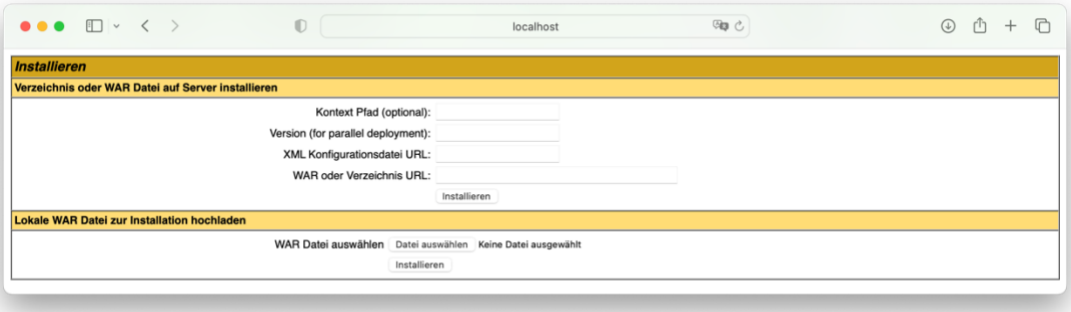
genannt.
Dieser Ansatz wird in der Regel verwendet, um eine Anwendung, die von anderen Entwicklern stammt, in eine vorhandene Tomcat-Installation zu installieren.

- **Verwenden der Tomcat-Webanwendung "Manager"**

Tomcat enthält eine Webanwendung, die im Kontextpfad /manager bereitgestellt wird. Sie zeigt zum einen die deployten Webanwendungen („/Context“) an:




und kann Web-Anwendungen auf einem laufenden Tomcat-Server deployen und löschen, ohne ihn neu zu starten. Informationen zur Manager-Webanwendung finden sich in der Tomcat Doku unter „Manager-App-Anleitung“.




Es gibt noch weitere Optionen um den Manager zu automatisieren, die sind aber hier erstmal ausgeklammert.

### 1.7.2 Deployment mit intellij UE

Der gängige Weg ist das Deployment aus einer IDE heraus. Sowohl eclipse als auch intellij (letzteres in der kostenpflichtigen UE Version) bieten das entwickeln und ausführen von JSP und Servlets-Programmen aus der IDE heraus an.

Das Ausführen aus der IDE heraus erfolgt in einer temporären Instanz des Tomcat. In der Regel kollidiert die Portnummer der lokalen Instanz und der regulären Instanz – das lässt sich konfigurieren, oder man fährt die „reguläre“ während der Entwicklung runter.

Wenn ein Deployment in die „reguläre“ Instanz ansteht, ist der stabilste Weg folgender:

1. Tomcat runterfahren mit

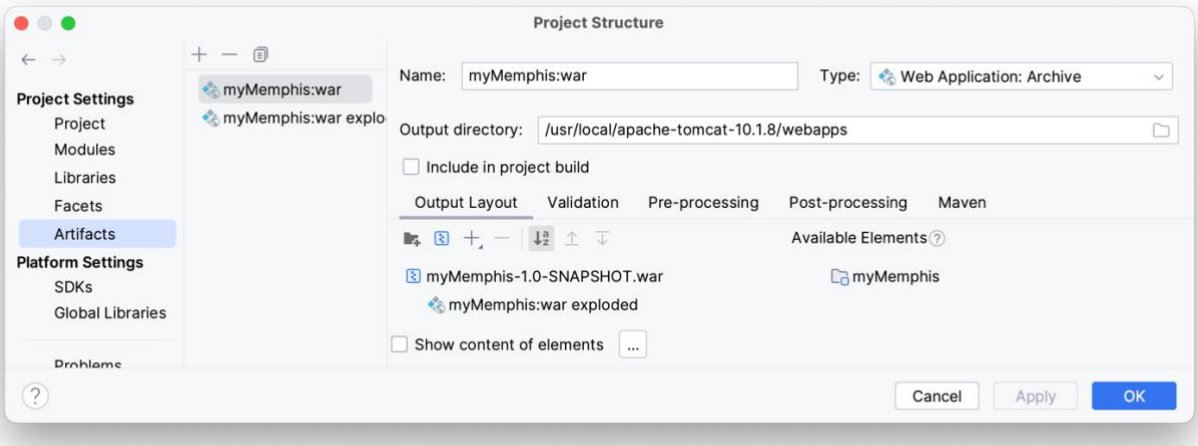
   /Library/Tomcat/bin/shutdown.sh

2. intellij IDE das *.war Artefakt bauen lassen.

   Dazu in der Menüleiste unter **Build** den Punkt **Build Artifacts** auswählen, und dann das <projekt>:war File bauen lassen. Hier ist der Projektname myMemphis:

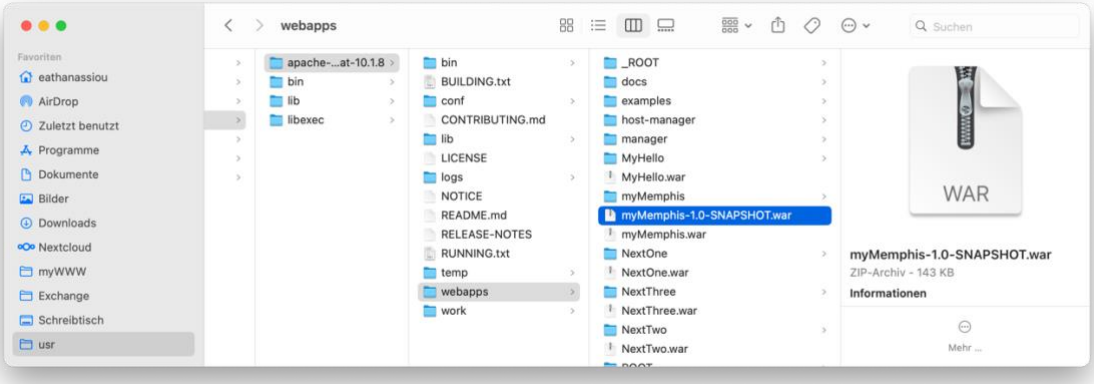

3. Das *.war File kann der IDE direkt in das webapps Verzeichnis schreiben lassen.

   Das kann man unter dem Menüpunkt **File** und dann **Project Structure** einstellen:



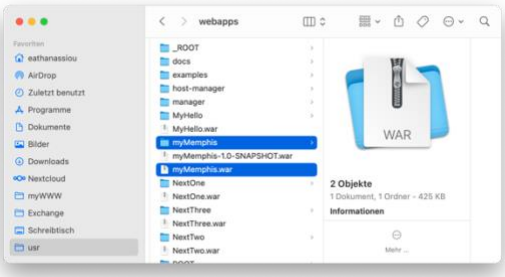
4. Um ein versehentliches Überschreiben zu verhindern wird eine Nummer und eine „Snapshot“-Nomenklatur verwendet.

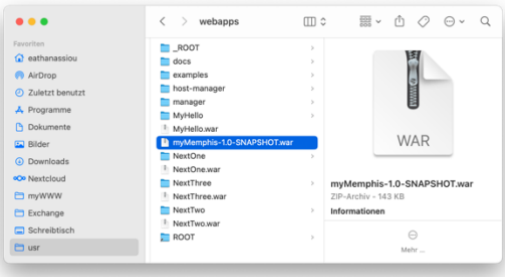
   Hier ist es myMemphis-1.0-SNAPSHOT.war:




5. Löschen des „alten“ war Files und Verzeichnis.

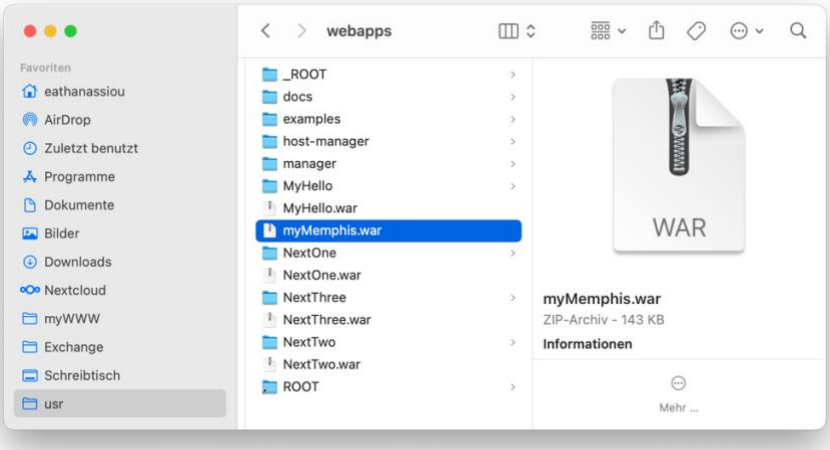
   Hier sind das die Datei myMemphis.war und das Verzeichnis myMemphis:




6. Umbenennen des „neuen“ war Files.

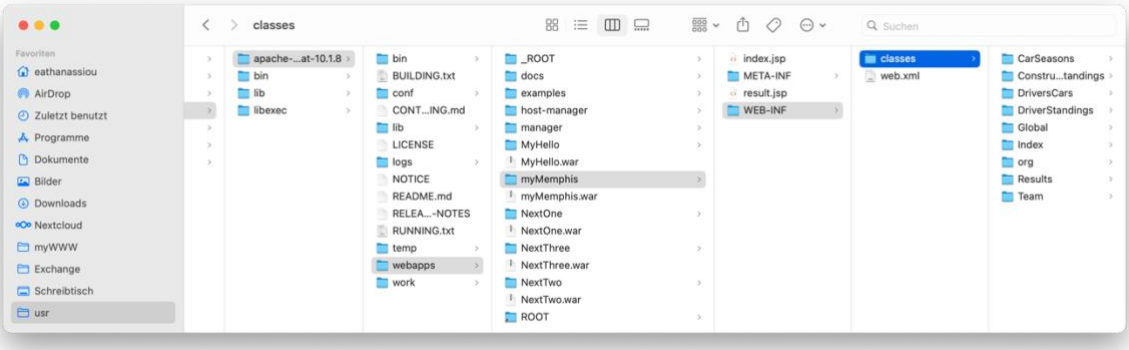
   Hier wird myMemphis-1.0-SNAPSHOT.war umbenannt in myMemphis.war:




7. Starten des Tomcats mit

   /Library/Tomcat/bin/startup.sh

8. Der frische gestartete Tomcat „entdeckt“ das neue *war File, entpackt es, und führt damit ein „Hot Deployment“ aus:




Um das ganze etwas zu vereinfachen habe ich für die Schritte 4-7 ein Shell-Skript erstellt:

```
#!/bin/sh
MY_MESSAGE=">----------  Deploying NextThree  "
echo $MY_MESSAGE       # This is a comment, too!
cd /usr/local/apache-tomcat-10.1.8/webapps
echo "--- Deleting old *.war File"
rm NextThree.war
echo "--- Removing old Directory"
rm -r NextThree
```

```
echo "--- Renaming new *.war File"
mv NextThree-1.0-SNAPSHOT.war NextThree.war
echo "--- Starting TomCat"
/Library/Tomcat/bin/startup.sh
```

Und unter ./Shell/first.sh. abgelegt, sowie ausführbar gemacht (chmod).

## 1.8 Deinstallieren von Tomcat

Die Deinstallation von Tomcat ist fast noch einfacher als die Installation.

**Achtung**: Bei der Befehlssequenz werden aber auch alle Web-Apps (*.html,*.jsp,*.jav), die im Verzeichnis webapps installiert wurden , ebenfalls entfernt werden.

```
sudo rm -f /Library/Tomcat
sudo rm -rf /usr/local/apache-tomcat-10.1.8
```

# 2. Java Server Pages (JSP)

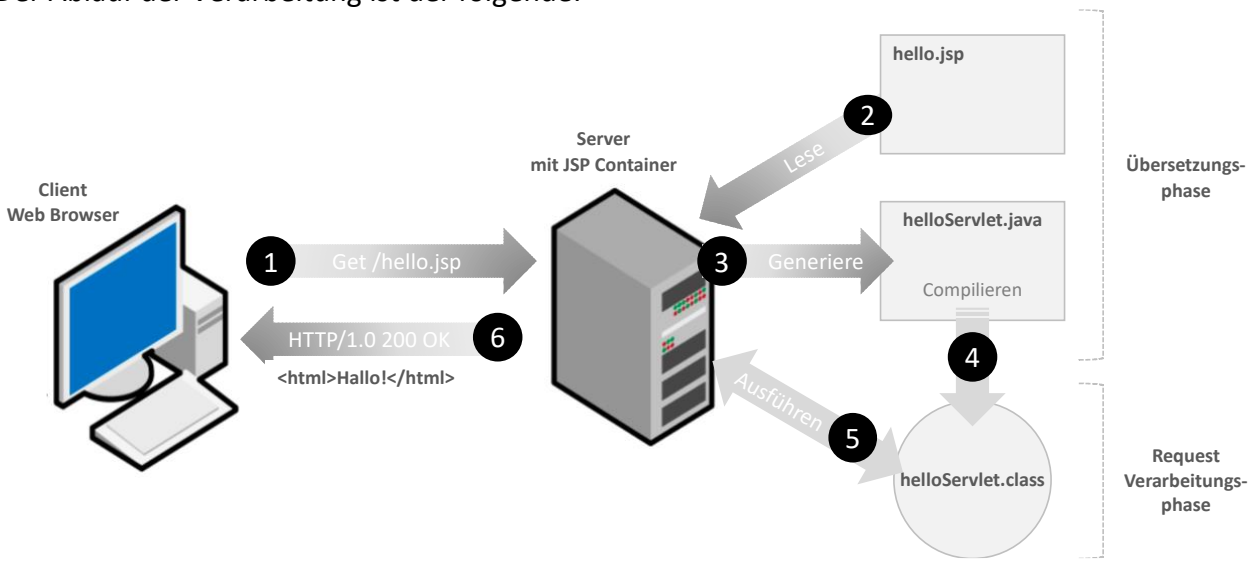
JSP ist keine Programmiersprache, sondern eine Technologie. Es ermöglicht Webseiten-autoren, nicht nur statische HTML-Seiten zu gestalten, sondern mit dynamischen Daten zu versehen.

Dazu bettet JSP Java-Code in HTML-Seiten ein, der Code wird auf vom Webserver (genauer: Application-Server) ausgeführt und das Ergebnis als HTML-Code an den Client übertragen, so dass der Client die Java-Anteile gar nicht mehr zu sehen bekommt. Im Gegensatz zu Java-Script oder JAVA-Applets wird der Programmcode nicht auf dem Client (Browser) ausgeführt, sondern auf dem Server.

## 2.1 Verarbeitung

Technisch basieren JSP-Seiten auf JAVA-Servlets, jede JSP-Seite wird in ein Java-Servlet übersetzt bevor sie vom Application-Server ausgeführt wird. Sie sind limitiert auf Browser als Clients und HTML als Datenformat.

Der Ablauf der Verarbeitung ist der folgende:




1. Wie bei einer HTML-Seite sendet der Browser eine HTTP-Anfrage an den Webserver.

2. Der Webserver erkennt, dass die HTTP-Anfrage für eine JSP-Seite bestimmt ist und leitet sie an eine JSP-Engine weiter. Die JSP-Engine lädt die JSP-Seite von der Festplatte und konvertiert sie in einen Servlet-Inhalt.

   Diese Konvertierung ist sehr einfach, da der ganze Vorlagentext in println( )-Anweisungen und alle JSP-Elemente in Java-Code konvertiert werden. Dieser Code implementiert das entsprechende dynamische Verhalten der Seite.

3. Die JSP-Engine kompiliert das Servlet in eine ausführbare Klasse und leitet die ursprüngliche Anfrage an eine Servlet-Engine weiter.

4. Der Applikations-Server, genauer die Servlet Engine, lädt die Servlet-Klasse und führt sie aus. Dabei erzeugt das Servlet eine Ausgabe im HTML-Format. Die Ausgabe wird von der Servlet-Engine als HTTP-Antwort an den Webserver gereicht.

5. Der Webserver gibt die HTTP-Antwort als statischen HTML-Inhalt an den Browser.

6. Schließlich behandelt der Webbrowser die dynamisch generierte HTML-Seite in der HTTP-Antwort genauso, als wäre es eine statische Seite.

In der Regel prüft die JSP-Engine, ob bereits ein Servlet für eine JSP-Datei vorhanden ist und ob das JSP älter als das Servlet ist. Ist dies der Fall, wird davon ausgegangen, dass sich die JSP nicht geändert hat und dass das generierte Servlet immer noch mit der JSP übereinstimmt

Im Grunde genommen ermöglicht JSP, Servlets zu schreiben, ohne sich tief in der Java-Servlet Programmierung auskennen zu müssen. Mit Ausnahme der Übersetzungsphase wird eine JSP-Seite genauso behandelt wie ein normales Servlet.

## 2.2 Einbettung von Java-Code in HTML <%

Die Einbettung von JAVA-Code in HTML geschieht durch ein < bzw. > Tag mit dem Prozentzeichen %. Zwischen den Tags <% und %> wird Java „gesprochen“.

```
<% /* Hier steht Java-Code */ %>
```

Dieser eingebetteter Java-Code wird auch Scriptlet genannt. Für das obige Scriptlet gibt es auch ein XML-Äquivalent:

```
<jsp:scriptlet>
  c/* Hier steht Java-Code */
</jsp:scriptlet>
```

Im Folgenden das einfachste und erste Beispiel für JSP:

```
<html>
  <head><title>Hello World</title></head>
  <body>
    Hello World! <br/>
    <%
```

```
        out.println("Your IP address : "+ request.getRemoteAddr());
        %>
      </body>
    </html>
```

Auf die Methode request.getRemoteAddr()gehen wir im Kapitel zu impliziten (vordefinierten) Objekten gleich ein.

## 2.3 JSP-Deklarationen <%!

Eine Deklaration deklariert eine oder mehrere Variablen oder Methoden, die später in der JSP-Datei in Java-Code verwendet werden können. Variablen oder Methoden müssen vor Verwendung deklariert, bevor Sie sie in der JSP-Datei verwenden. Die Syntax für JSP-Deklarationen verwendet das TAG <%!, also mit einem Ausrufezeichen:

Syntax:
```
<%! declaration; [ declaration; ]+ ... %>
```

In XML:
```
<jsp:declaration>
        code fragment
</jsp:declaration>
```

Beispiele:
```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

## 2.4 JSP- Expressions <%=

Eine JSP-Expression enthält einen Skriptletausdruck, der ausgewertet, in einen String konvertiert und dort eingefügt wird, an der er in der JSP-Datei angezeigt wird. Die Syntax für JSP-Expressions verwendet das Tag <%=, also mit einem Gleichheitszeichen:

Syntax:
```
<%= expression %>
```

In XML:
```
<jsp:expression>
        expression
</jsp:expression>
```

Beispiel:
```
<html>
```

```
<head><title>A Comment Test</title></head>
<body>
  <p>
  Today is <%= (new java.util.Date()).toLocaleString()%>
  </p>
</body>
</html>
```

Folgende Ausdrücke kennt JSP:

- Boolean true und false
- Integer wie in JAVA
- Floating point wie in JAVA
- String mit einfachen und doppelten Anführungszeichen
  Maskierung: " -> \", ' -> \' , \ -> \\
- Null null

Erwartungsgemäß ist das analog zu den Ausdrücken in JAVA.

## 2.5 Kommentare <%--

JSP-Kommentare markieren Text oder Anweisungen, die der JSP-Container ignorieren soll. Ein JSP-Kommentar ist nützlich, wenn ein Teil einer JSP-Seite ausblenden werden soll. Die Syntax für JSP-Kommentare verwendet das Tag <%--, also mit zwei Minuszeichen:

Syntax:

```
<%-- This is a JSP comment --%>
```

Beispiel:

```
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
<%-- This comment will not be visible in the page source --%>
</body>
</html>
```

Zur Unterscheidung:

<%-- comment --%> JSP-Kommentar, wird von der JSP-Engine ignoriert.

<!-- comment --> HTML-Kommentar, wird vom Browser ignoriert.

## 2.6 JSP-Control Flow und Operatoren

Alle JAVA Sprachkonstruktionen, –elemente und APIs können in der JSP-Programmierung verwendet werden:

- alle Verzweigungen if...else und Fallunterscheidungen (switch...case )
- Die drei grundlegende Arten von Schleifen (for, while und do ... )
- alle logischen und arithmetischen Operatoren, die von Java unterstützt werden

## 2.7 Eigene Java-Methoden in JSP

Wenn auf einer JSP-Seite eine eigenen Java-Methode benötigt wird, wird sie mit dem Deklarations-Tag <%!und %> umschlossen.

```
<%!
String mfg(String name)
{    return "Hallo " + name; }
%>
…
<% String user = (String)session.getAttribute("user"); %>
<% out.print(mfg(user)); %>, schön Dich zu sehen.
```

Die Methode kann sowohl im Header als auch im Body der JSP-Seite angelegt werden.

## 2.8 JSP-Direktiven <%@

Wird hinter dem öffnenden JSP-Tag <% ein @-Zeichen gesetzt, handelt es sich um eine sogenannte *Direktive*.

Das erste Schlüsselwort der JSP-Direktive bezeichnet die Art der Direktiven, es gibt drei Arten:

- page
- include
- taglib

### 2.8.1 page-Direktive

Die page-Direktive wird dazu benutzt, Voreinstellungen zur aktuellen JSP-Seite zu treffen. Mehrere page-Direktiven innerhalb einer JSP sind erlaubt, jedoch pro Attribut nur einmal. Page-Direktiven können an einer beliebigen Stelle stehen, gemäß der Konvention werden sie aber oben auf der JSP-Seite positioniert.

Syntax: `<%@ page attribute = "value" %>`

In XML: `<jsp:directive.page attribute = "value" />`

Die Attribute sind:

```
<%@
page language      ="java"
Extends            ="Klassenname"
Import             ="Klassen-/Paket- Liste"
Buffer             ="none          | n [KiloByte]"
autoFlush  ="true  | false" nfo="Infotext"
inThreadSave       ="true  | false"
contentType        ="MIME-Type [; charset]"
errorPage  ="URL"
isErrorPage        ="true  | false"
session            ="true  | false"
%>
```

### 2.8.2 page-Direktive für Java-Import

Java **import**-Anweisungen können nicht einfach zwischen die Standard-Prozent-Tags gesetzt werden. Das betrifft nicht nur eigene Klassen, sondern vor allem die Standardbibliotheken.

```
<%@ page import = "java.util.Enumeration" %>
<%@ page import = "mypackage.Myklasse" %>
```

Eigene Klassen müssen darüber hinaus in eigenen Packages abgelegt und importiert werden.

### 2.8.3 include-Direktive

Teile von Webseiten können in eigenen Dateien abgelegt werden. Sie bekommen per Konvention die Endung jspf. Diese Fragmente können von verschiedenen JSP-Seiten aus eingebunden werden, indem eine Include-Direktive verwendet wird:

```
<%@ include file="header.jspf" %>
```

### 2.8.4 Taglib-Direktive

In JSP können benutzerdefinierte JSP-Tags definiert werden, die wie HTML- oder XML-Tags aussehen. Eine Tag-Library ist ein Satz benutzerdefinierter Tags, die benutzerdefiniertes Verhalten implementieren.

Die taglib-Direktive deklariert, dass eine JSP-Seite eine Reihe von benutzerdefinierten Tags verwendet und definiert wo die entsprechede Tag-Library zu finden ist.

Syntax `<%@ taglib uri="uri" prefix = "prefixOfTag" >`

`uri` zeigt wo die Library zu finden ist, und mit dem `prefix` Attribut kann man deklarieren welche Markups benutzerdefinierte Tags sind – Voraussetzung ist das die selbst definierten Tags alle mit dem `prefix` anfangen.

In XML `<jsp:directive.taglib uri = "uri" prefix = "prefixOfTag" />`

Mit der Taglib-Direktive können auch und gerade die 4 JSTL-Bibliotheken (Jakarta Standard Tag Library) eingebunden werden:

- `core` iterative, konditionale, URL-spezifische und allgemeine Tags
- `xml` Tags aus dem Bereich XML und XML-Transformation
- `sql` Tags zur direkten Datenbankverarbeitung
- `i18n` Tags zur Formatierung und Internationalisierung

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml"%>
```

Die Core-Library wurde hier dem Prefix `c` und die XML-Library dem Prefix `x` zugeordnet.

# 3. Implizite (vordefinierte) JSP-Objekte

Die zugrunde liegende JAVA-Servlets bringen eine Reihe vordefinierter Variablen und Methoden (Implicit Objects) mit sich.

Bei diesen Objekten handelt es sich um Java-Objekte, die der JSP-Container auf jeder Seite aufgerufen werden können, ohne sie vorher explizit deklarieren zu müssen. JSP unterstützt neun implizite Objekte:

| Objekt | Beschreibung |
|---|---|
| out | Das **PrintWriter** Object zum Senden von Output an den Client (Browser). |
| session | Ein **HttpSession** Objekt. Das „Sitzungsobjekt“ wird verwendet, um **sessions** zwischen Client-Requests zu verfolgen. Hier können Daten aus einer Session gespeichert und zwischen JSP-Seiten einer Session übergeben werden. |
| application | Das **ServletContext** Objekt ist eine Darstellung der JSP-Seite während ihres gesamten Lebenszyklus. Dieses Objekt wird erstellt, wenn die JSP-Seite initialisiert wird, und wird entfernt, wenn die JSP-Seite wieder entfernt wird. |
| request | **HttpServletRequest** Objekt. Es repräsentiert den **request** eines Clients. Jedes Mal, wenn ein Client eine Seite anfordert, erstellt die JSP-Engine ein neues Objekt, um den request darzustellen. |
| response | **HttpServletResponse** Objekt. So wie der Server das **request** Objekt erstellt, erstellt er ein Objekt, das die **response** an den Client darstellt. |
| config | Das S**ervletConfig** Objekt ermöglicht dem JSP-Programmierer den Zugriff auf die Servlet- oder JSP-Engine-Initialisierungsparameter wie Pfade oder Dateispeicherorte usw.. |
| pageContext | Das **pageContext**-Objekt wird verwendet, um die gesamte JSP-Seite zu repräsentieren. Man kann auf Informationen über die Seite zuzugreifen und dabei die meisten Implementierungsdetails ausblenden. |
| page | Dies ist lediglich ein Synonym für **this** und wird zum Aufrufen der von der übersetzten Servlet-Klasse definierten Methoden verwendet. |
| Exception | Das **Exception**-Objekt ermöglicht den Zugriff auf die Exception-Daten durch eine dedizierte, anzugebende JSP. |

In den folgenden Abschnitten soll kurz auf die wichtigsten Konzepte und Methoden der impliziten Objekte eingegangen werden.

## 3.1 out

Mit **out** werden direkte Ausgaben an den Client durchgeführt. Es bietet zudem Methoden zur Verwaltung des Ausgabe-Puffers.

- **out.print()** fügt die Werte von Java-Variablen als String in die HTML-Ausgabe ein:
  <hr> <% out.print(javavar); %> <hr.

  Das Tag kann durch ein Gleichheitszeichen zu <%= javavar %> verkürzt werden.

- Mit **out.println()** wird eine Zeile mit Zeilenvorschub ausgegeben.

- **out.flush()** veranlasst eine sofortige Übertragung aller im Puffer enthaltenen Daten an den Client /Browser.

Das Objekt stammt von der Klasse javax.servlet.jsp.JspWriter ab.

## 3.2 session

Durch das Objekt session kann auf Daten der aktuelle session zugegriffen, aber eigene Werte gespeichert werden. Typischerweise werden die Daten als assoziative Attribute abgelegt, vergleichbar mit einer Map.

- **setAttribute()** bindet Objekte und Werte unter einem dedizierten Namen an die Session gebunden: session.setAttribute("kunde", kunde); Alle verwendeten JSP Seiten einer **session** können darauf dann zugreifen.

- **getAttribute()** Liefert das verbundene Objekt mit dem angegebenen Namen in der Session zurück: Kunde kunde = (Kunde)session.getAttribute("kunde");

- **getAttributeNames()** liefert eine Enumeration von Strings, welche die Namen aller Objekte enthält, die sich in der Session befinden.

- **removeAttribute(java.lang.String name)** löscht das durch den Namen spezifizierte Objekt aus der Session

- **getId()** liefert die eindeutige Session-ID als String zurück.

**session** implementiert das Interface javax.servlet.http.HttpSession .

## 3.3 application

Auch das Objekt **application** ermöglicht die Verwendung von getAttribute und setAttribute. Sie bleiben hier allerdings für die Dauer der Anwendung und nicht nur der Sitzung zugreifbar. Es bietet auch Informationen wie den verwendeten Servlet-Container die Version der Servlet-API, und beherbergt Funktionen zur Protokollierung.

- **getAttribute()** liefert die Servlet-Container-Attribute, spezifiziert durch den Namen, oder NULL, wenn dieses nicht vorhanden ist.

- **getServerInfo()** liefert Namen und Versionsnummer des Servlet-Containers.

- **log(java.lang.String msg)** schreibt die angegebene Nachricht in die Server Log-Datei.

Dieses Objekt implementiert das Interface javax.servlet.ServletContext.

## 3.4 request

Der elementare Ablauf bei allen Web-Anwendungen ist ein Wechselspiel von „requests“ eines Clients und der „response“ eines Servers. Jedes Mal, wenn ein Client eine Seite anfordert, erstellt die JSP-Engine ein neues **request** Objekt, um die Client Anfrage und all seine Daten darzustellen.

- **request.getParameter(String)** liefert den Wert eines übergebenen Parameter aus dem Client request : String str = request.getParameter ("name"); liefert den Wert eines übergebenen Parameter mit dem Namen „name“.

- **request.getParameterNames()** liefert die Paramternamen eines requests.Die Parameternamen sollten aus dem aufrufenden HTML oder JSP Seite bekannt sein, können aber durch request.getParameterNames()als Liste erfragt werden.

- **request.getParameterValues( ).** Wenn die Namen der FORM Elemente im Client unbekannt sind, oder wenn pro FORM Element mehrere Werte möglich sind (Checkboxen), können die Namen und Werte ermittelt werden.
  Der folgende Code liest alle Namen und Werte aus einem aufrufenden HTML-Formular aus, und zeigt diese an:

```
<html><body>
<%@ page import = "java.util.*" %>
<b>Parameters:</b><br>
<%
  Enumeration parameterList = request.getParameterNames();
  while(parameterList.hasMoreElements() )
```

```
 { String   sName     = parameterList.nextElement().toString();
   String[] sMultiple = request.getParameterValues( sName );
   if( 1 >= sMultiple.length )
     out.println(sName + " = "+request.getParameter(sName )+"<br>" );
   else
     for( int i=0; i<sMultiple.length; i++ )
       out.println(sName + "[" + i + "] =" + sMultiple[i]+"<br>" );
 }
%>
</body> </html>
```

- **String getMethod()** gibt den Namen der HTTP-Methode zurück, mit der der request gestellt wurde, z. B. GET, POST oder PUT.
- **getAttribute(String name)** Gibt den Wert session Attributs als Object urück oder null zurück, wenn kein Attribut des angegebenen Namens vorhanden ist.
- **Cookie[] getCookies()** gibt ein Array zurück, das alle Cookie-Objekte enthält, die der Client mit dem request gesendet hat.
- **String getRemoteAddr()** gibt die IP-Adresse (Internet Protocol) des Clients zurück, der den request gesendet hat.
- **String getRemoteHost()** gibt den vollen Namen des Client Hosts zurück, der den requestgesendet hat.
- **String getRemoteUser()** gibt den Usernamen des Benutzers zurück, der den request gesendet hat stellt, falls der Benutzer authentifiziert wurde, bzw. null, wenn nicht.

**request** instantiiert ein javax.servlet.http.HttpServletRequest Objekt.

## 3.5 response

Das response Objekt liefert die Antwort an den Client. Wenn ein Webserver auf einen HTTP-request antwortet, besteht die Antwort in der Regel aus einer Statuszeile, diversen Antwort-headern, einer Leerzeile und dem Dokument. Eine typische Antwort sieht wie folgt aus:

```
HTTP/1.1 200 OK
Content-Type: text/html
```

```
Header2: ...
HeaderN: ...

<!doctype ...>
<html>
  <head>...</head>
  <body>
    ...
  </body>
</html>
```

Die Statuszeile besteht aus der HTTP-Version (hier:  HTTP/1.1), einem Statuscode (hier:200) und einer sehr kurzen Nachricht, die dem Statuscode entspricht (im Beispiel OK). Die danach folgenden diversen Headerzeilen (Allow**,** Content-Type, Content-Language, Content-Encoding, Last-Modified, Set-Cookie, ….) können mit dem response-Objekt gesetzt werden.

- **void setCharacterEncoding(String charset)** legt die Zeichencodierung (MIME-Zeichensatz) der Antwort, die an den Client gesendet wird, z. B. auf UTF-8 fest.
- **void setContentLength(int len)** legt den HTTP Content-Length-Header fest.
- **void setContentType(String type)** legt den content-type der response an den Client fest.
- **void setDateHeader(String name, long date)** legt einen Antwortheader mit dem angegebenen Namen und dem Datumswert fest.
- **void setHeader(String name, String value)** legt einen Antwortheader mit dem angegebenen Namen und Wert fest.
- **void sendError(int sc)** sendet, unter Verwendung des angegebenen Statuscode eine Fehlerantwort an den Client und löscht den Puffer.
- **void sendRedirect()** lenkt die response an den Client auf die angegebene URL.

```
response.sendRedirect("index.html");
response.sendRedirect("main.jsp");
```

Die Seitenumleitung wird in der Regel verwendet, wenn ein Dokument an einen neuen Ort /URL verschoben wurde und die Anfrage des Client an diesen neuen Speicherort weitergeleitet werden muss.  Der einfachste Weg, einer Umleitung, ist die Verwendung der Methode sendRedirect() des Antwortobjekt:

```
public void response.sendRedirect(String location) throws IOException
```

Diese Methode sendet die response zusammen mit dem Statuscode und der neuen URL an den Browser zurück. Das gleiche erreicht man durch die Verwendung der Methoden **setStatus**() und **setHeader()** :

```
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", "http://www.newpage.com");
```

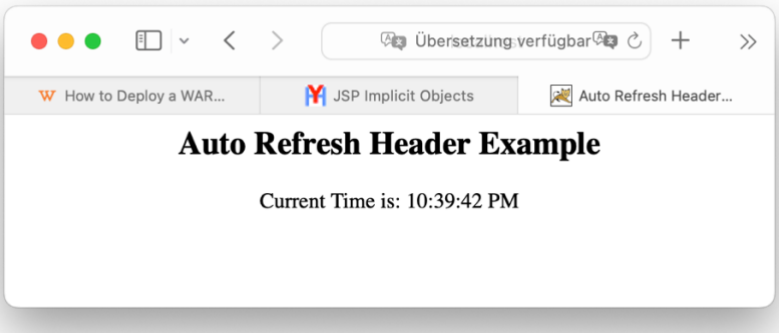
Die Status-Werte für response.SetStatus() sind:

| Name | Status Code | |
|---|---|---|
| SC_ACCEPTED | 202 | Der request wurde akzeptiert, aber nicht bearbeitet. |
| SC_BAD_GATEWAY | 202 | Der HTTP-Server hat eine ungültige response von einem Server erhalten, als er als Proxy oder Gateway fungierte. |
| SC_BAD_REQUEST | 202 | Ein vom Client gesendete request ist syntaktisch falsch. |
| SC_CONFLICT | 409 | Ein request konnte aufgrund eines Ressourcenkonfliktes nicht abgeschlossen werden. |
| SC_CONTINUE | 100 | Alles gut – der Client kann weitermachen ;-) |
| SC_CREATED | 201 | Für einen request wurde eine neue Ressource auf dem Server erstellt |
| SC_EXPECTATION_FAILED | 417 | Der Server konnte die im Expect-Anforderungsheader angegebene Erwartung nicht erfüllen. |
| SC_FORBIDDEN | 401 | Der Server verstand den request, aber die Ausführung ist untersagt. |
| SC_FOUND | 302 | Die Ressource befindet sich vorübergehend unter einem anderen URI. |
| SC_GATEWAY_TIMEOUT | 504 | Der Server hat keine rechtzeitige Antwort vom Upstreamserver erhalten, während er als Gateway oder Proxy fungierte. |
| SC_GONE | 410 | Die Ressource ist auf dem Server nicht mehr verfügbar und es ist keine Weiterleitungsadresse bekannt. |
| SC_HTTP_VERSION_NOT_SUPPORTE | 505 | Der Server unterstützt die http-Version nicht. |
| SC_INTERNAL_SERVER_ERROR | 500 | Interner Fehler. |
| SC_LENGTH_REQUIRED | 411 | Der request kann nicht ohne eine definierte Content-Length verarbeitet werden. |
| SC_MOVED_PERMANENTLY | 301 | Ressource wurde dauerhaft an einen neuen Speicherort verschoben. |
| SC_MOVED_TEMPORARILY | 302 | Eine Ressource wurde vorübergehend an einen anderen Speicherort verschoben. |
| SC_NOT_FOUND | 404 | Eine angeforderte Ressource ist nicht verfügbar. |
| SC_NOT_IMPLEMENTED | 501 | Der http-Server unterstützt die angeforderte Funktionalität nicht. |
| SC_REQUEST_TIMEOUT | 408 | Timeout im request. |
| SC_REQUEST_URI_TOO_LONG | 414 | Request-URI is t zu lang |
| SC_USE_PROXY | 305 | Auf eine angeforderte Ressource MUSS über den Proxy zugegriffen werden, der im Feld Standort angegeben ist. |

Beispiel zu implizite Objekte:

```
<%@ page import = "java.io.*,java.util.*" %>
<html>
  <head> <title>Auto Refresh Header Beispel</title> </head>
  <body>
    <center>
      <h2>Auto Refresh Header Beispiel</h2>
      <%
        response.setIntHeader("Refresh", 5)
        Calendar calendar = new GregorianCalendar();
        String am_pm;
        int hour = calendar.get(Calendar.HOUR);
        int minute = calendar.get(Calendar.MINUTE);
        int second = calendar.get(Calendar.SECOND)
        if(calendar.get(Calendar.AM_PM) == 0)
          am_pm = "AM";
        else
          am_pm = "PM";
          String CT = hour+":"+ minute +":"+ second +" "+ am_pm;
          out.println("Aktuelle Uhrzeit ist: " + CT + "\n");
      %>
    </center>
  </body></html>
```

Der Client wird sich alle 5 sec aktualisieren und die neue Zeit anzeigen.




## 3.6 exception

Das JSP exception-Objekt ist ein Objekt der Klasse javax.servlet.jsp.JspException. Es steht nur zur Fehlerbehandlung in speziell dafür vorgesehenen „error pages“ Pages zur Verfügung.

Mit JSP kann man für jede JSP eine „error page“ angeben. Immer wenn die Seite eine Exception auslöst, ruft der JSP-Container automatisch diese „error page“ auf.

Zum Einrichten einer Fehlerseite, verwendet man die Direktive:

```
<%@ page errorPage = "xxx.jsp" %>.
```

Ein Beispiel - die folgende JSP-Seite löst eine exception aus:

```
<%@ page errorPage = "ShowError.jsp" %>
<html>
  <head <title>Error Handling Example</title>  </head>
  <body>
    <% // Auslösen einer Eception, um die Fehlerseite aufzurufen
      int x = 1;
      if (x == 1) {
        throw new RuntimeException("Error condition!!!");
      }
    %>
  </body>
</html>
```

Nun noch die „error page“ hier ShowError.jsp. Sie wird deklariert durch die Direktive

```
<%@ page isErrorPage = "true" %>
```

Diese Direktive veranlasst den JSP-Compiler, die „exception instance“ zu generieren. ShowError.jsp benutzt die Methode exception.printStackTrace() zur Ausgabe.

```
<%@ page isErrorPage = "true" %>
<html>
  <head> <title>Show Error Page</title></head>
  <body>
    <h1>Opps...</h1>
    <p>Sorry, an error occurred.</p>
    <p>Here is the exception stack trace: </p>
    <pre><% exception.printStackTrace(response.getWriter()); %></pre>
  </body>
</html>
```

Einige der Methoden für das exception Objekt sind:

- **Throwable getCause()** liefert die Exception zurück, die in der JSP auftrat.
- **String getMessage()** liefert eine ausführliche Meldung über die aufgetretene Exception.
- **String toString()** zusätzlich den Namen der Klasse zurück.
- **void printStackTrace()** gibt das Ergebnis von toString() zusammen mit der „stack trace“ in System.err (dem Standard „error output stream“), aus.
- **StackTraceElement [] getStackTrace()** gibt ein Array zurück, das jedes Element der „error stack trace“ enthält. Am Index 0 ist das letzte Element der Aufrufliste abgelegt, und das letzte Element im Array enthält die erste Methode der Aufrufliste.

## 3.7 config

Das config-Objekt ist eine Instanziierung von javax.servlet.ServletConfig. Es enthält die Konfigurationsdetails wie den Benutzernamen, das Passwort, Pfade oder Dateispeicherorte ,

Parameternamen und ihre Werte, die in der Konfigurationsdatei WEB-/INF/web.xml festgelegt sind. Einige der Methoden des config-Objektes sind:

- String **getInitParameter(String iname)** liefert den Wert des Parameters **iname** zurück, der im <init-param> Element in der Datei WEB-INF\web.xml definiert ist, oder null.
- Enumeration **getInitParameterNames()** liefert eine Aufzählung der Namen aller initialen Parameter aus web.xml**,** oder null.
- String **getServletName()** gibt den Servlet-Namen zurück, der im <servlet-name> Element in der Datei WEB-INF\web.xml definiert ist .

## 3.8 web.xml

Es ist an der Zeit einen ersten Blick auf die Datei web.xml zu werfen. In der Datei können eine große Anzahl an Einstellungen und Initialisierungen vorgenommen werden. Der Natur von JSP folgend, gehen Sie auf die Konfiguration der zugrundeliegenden JAVA-Servlets ein.

In der Beispiel web.xml Datei weiter werden für Web-Anwendung mit einer JSP-Seite JspConfig.jsp einige Einstellungen vorgenommen.

Eine web.xml Datei umschließt mit dem <web-app> Tag eine ganze Reihe weiterer Tags.

Das Tag <servlet> sticht hervor, und hier werden (keine Überraschungen) einige XML-Kerneinstellungen zum Servlet und den dazugehörigen JSP-Seiten vorgenommen. Mit:

```
<servlet>
        <servlet-name>MyJspPage</servlet-name>
    <jsp-file>/JspConfig.jsp</jsp-file>
</servlet>
```

Wird ein Name für das Servlet definiert, zu dem ein angegebenes JSP-File gehört. Hier kann auch eine eigentwickeltes JAVA-Servlet (genauer die Klasse ) stehen, das Tag heißt <servlet-class>.

Ein Beispiel: <servlet-class>MyServlet</servlet-class>

Mit dem eingefügten Tag <init-param> werden jetzt Parameterwerte initialisiert.

```
<servlet>
        <servlet-name>MyJspPage</servlet-name>
    <jsp-file>/JspConfig.jsp</jsp-file>

        <init-param>
            <param-name>TestName</param-name>
            <param-value>DecodeJava</param-value>
     </init-param>
</servlet>
```

Hier wird ein Parameter mit dem Namen `TestName` mit dem Wert `DecodeJava` initialisiert. Diese Werte sind dann später durch das implizite config-Objekt abrufbar.

Das zweite hervorstechende Tag ist <servlet-mapping>.

```
<servlet-mapping>
        <servlet-name>MyJspPage</servlet-name>
    <url-pattern>/JspConfig.jsp</url-pattern>
</servlet-mapping>
```

<url-pattern> wird in web.xml verwendet, um ein Servlet einer URL zuzuordnen.

Wenn man <url-pattern> von /JspConfig.jsp in /MyUrl ändert, kann auf das Servlet mithilfe von /MyUrl zugegriffen werden – falls man aus Sicherheitsgründen die eigentliche URL ausblenden möchte.

| Tag | Gehört unter | Beschreibung |
|---|---|---|
| **<web-app>** | | Die gesamte Anwendung |
| **<servlet>** | <web-app> | Stellt das Servlet dar |
| **<servlet-name>** | <servlet> | Name des Servlets |
| **<servlet-class>** | <servlet> | java class des Servlets |
| **<servlet-mapping>** | <web-app> | Zuordnung von JSP-Seiten zu einem Servlet |
| **<url-pattern>** | <servlet-mapping> | pattern wird clientseitig verwendet, um das Servlet aufzurufen |

Ein weiteres Tag in web.xml ist <filter> - dazu wird es ein eigenes Kapitel geben müssen ...

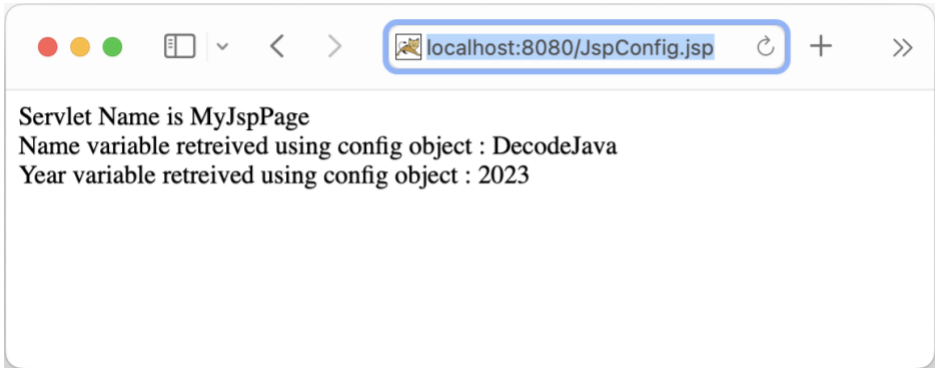
Das web.xml File sieht jetzt also so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
 https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
 version="6.0"
 metadata-complete="true">
 <display-name>Welcome to Tomcat</display-name>
 <description> Welcome to Tomcat </description>

 <servlet>
  <servlet-name>MyJspPage</servlet-name>
  <jsp-file>/JspConfig.jsp</jsp-file>
  <init-param>
   <param-name>Name</param-name>
   <param-value>DecodeJava</param-value>
  </init-param>
  <init-param>
   <param-name>Year</param-name>
   <param-value>2023</param-value>
   </init-param>
</servlet>

<servlet-mapping>
   <servlet-name>MyJspPage</servlet-name>
   <url-pattern>/JspConfig.jsp</url-pattern>
</servlet-mapping>
</web-app>
```

Die JSP-Seite JspConfig.jsp liest jetzt die <param-name> - Werte aus web.xml über das config-Objekt aus:

```
…
<%
String sname = config.getServletName();
out.println("Servlet Name is "+ sname);
%>
<br/>
<%
out.println("Name variable retreived using config object : " + config.getInitParameter("Name"));
out.println("<br/>");
out.println("Year variable retreived using config object : " + config.getInitParameter("Year"));
%>
…
```

Die Ausgabe ist wenig überraschend:




## 3.9 pageContext

Das pageContext-Objekt repräsentiert. die ganze JSP-Seite zu. Man kann auf Informationen über die JSP-Seite zuzugreifen ohne die Implementierungsdetails kennen zu müssen. Dieses Objekt speichert Verweise auf die request- und response-Objekte für jeden Request.
Die impliziten application, config, session, und out -Objekte werden durch den Zugriff auf Attribute dieses Objekts abgeleitet. Das pageContext-Objekt enthält auch Informationen zu den Direktiven der JSP-Seite, Pufferinformationen, der errorPageURL und des Seitenbereichs.

## 3.10 page

Das page-Objekt ist ein direktes Synonym für das this -Objekt. Es ist in der JSP-Programmierung nicht weiter von Bedeutung.

# 4. HTML-Formulare und JSP

In jeder Web Anwendungen müssen Informationen von einem Browser an den Webserver und schließlich an ein Backend-Programm übergeben müssen.
Ebenso überwiegend werden die Daten im Browser über HTML-Formulare erfasst. Der Browser verwendet zwei Methoden, um Informationen an den Webserver zu übergeben. Bei diesen Methoden handelt es sich um die GET-Methode und die POST-Methode.

## 4.1 GET

Die GET-Methode sendet die codierten Informationen aus dem Browser mit dem http-Request an den Server. Die Seitenanforderung und die codierten Informationen werden als key & value Paare (durch das „?“ Zeichen getrennt) übergeben:

```
http://www.test.com/hello?key1=value1&key2=value2
```

Die GET-Methode

- ist die Standardmethode zum Übergeben von Informationen vom Browser an den Webserver und erzeugt eine Zeichenfolge, die im **Feld** Location:Box des Browsers angezeigt wird.
- soll besser nicht verwendet werden, wenn ein Passwort oder andere vertrauliche Informationen an den Server weitergeben werden soll.
- hat eine Größenbeschränkung: Ein request String kann nur 1024 Zeichen lang sein.
- übergibt die Information im sog. QUERY_STRING Header. Die Daten sind über die QUERY_STRING Umgebungsvariable zugänglich, die mit getQueryString() und getParameter() des impliziten request() Objekts gelesen werden können.

## 4.2 POST

Eine zuverlässigere Methode zur Übergabe von Daten an ein Backend-Programm ist die POST-Methode.
Diese Methode verpackt die Informationen auf genau die gleiche Weise wie die GET-Methode, sendet sie aber nicht als Zeichenfolge nach einem „?“ in der URL, sondern als separate Nachricht.
JSP verarbeitet diese Art von requests mit der Methode **getParameter**() zum Lesen einfacher Parameter und der Methode **getInputStream()** zum Lesen von binären Datenstreams vom Browser.

## 4.3 GET/POST und HTML-Formulare

Die folgende URL übergibt mit der GET-Methode zwei Werte an das HelloForm-Programm.

http://localhost:8080/main.jsp?first_name=ZARA&last_name=ALI

Funktional exakt das gleiche kann man mit einem HTML-Formular bewirken.

```
<html>
  <body>
    <form action = "main.jsp" method = "GET">
      First Name: <input type = "text" name = "first_name">
      <br />
      Last Name: <input type = "text" name = "last_name" />
      <input type = "submit" value = "Submit" />
    </form>
  </body>
</html>
```

Das Drücken der „submit“-Taste wird exakt den gleichen request wie die obige URL an das JSP main.jsp übermitteln.

In dem JSP-Programm main.jsp wird mit der Methode **getParameter()** wird auf übergebenene Informationen zugegriffen.

```
<html>
  <head> <title> Read Form Data</title> </head>
  <body>

    <h1>Using GET Method to Read Form Data</h1>
    <ul>
      <li><p><b>First Name:</b>
        <%= request.getParameter("first_name")%>
      </p></li>
      <li><p><b>Last  Name:</b>
        <%= request.getParameter("last_name")%>
      </p></li>
```

```
      </ul>


    </body>
  </html>
```

Wie verhält sich das bei Verwendung der POST-Methode?

1. In einer URL kann die POST-Methode nicht verwendet werden.
2. Das HTML-Formular ändert sich nur an einer einzigen Stelle, im <form> Tag wird aus
   - method = "**GET**"

jetzt dann

   - method = "**POST**".

```
<html>
  <body>

    <form action = "main.jsp" method = "POST">
      First Name: <input type = "text" name = "first_name">
      <br />
      Last Name: <input type = "text" name = "last_name" />
      <input type = "submit" value = "Submit" />
    </form>

  </body>
</html>
```

3. Das JSP-Programm main.jsp funktioniert unverändert (!) da keine Binärdaten an das JSP-Programm übergeben werden.

## 4.4 Lesen von HTML-Formulardaten mit JSP

JSP verarbeitet das Parsen von Formulardaten automatisch mit den folgenden Methoden, abhängig von der Situation:

- **getParameter()** − liefert den Wert eines namentlich bekannten Formularparameters.
- **getParameterValues()** − wird verwendet wenn der Parametername mehrfach verwendet wird oder mehrere Werte zurück liefert (z. Bsp. Checkbox)
- **getParameterNames()** – liefert die vollständige Liste aller Parameter des aktuellen requests.
- **getInputStream()** – zum Auslesen eines um den binären Datenstreams vom Client.

## 4.5 Eine erste JSP- Anwendung

Eine kleine Web-Anwendung soll als Beispiel dienen:

- newuser.jsp: Eingabemasken für u.a. User und Passwort. Das Passwort wird zwei Mal eingegeben. Sollte eine Nachricht für diese Sitzung bestehen, wird sie angezeigt.
- register.jsp: Die Eingabefelder werden geprüft. Fehlt die Benutzerkennung oder sind die beiden Passworteingaben unterschiedlich, wird newuser.jsp aufgerufen und eine Fehlernachricht als Session Attribut übergeben. Andernfalls werden die Eingaben für die Session gespeichert und main.jsp aufgerufen.
- main.jsp: Diese begrüßt den Benutzer und zeigt alle Angaben an.


Alle drei haben einen HTML-Rahmen, der erstmal keine JSP-Spezifika enthält, und der bei Verwendung von CSS noch breiter ausfällt. Im Kern ist der Rahmen aber simpel:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Insert title here</title>
</head>
<body>
            … JSP TEIL …
```

```
</body>
</html>
```

Zwischen die body-Tags wird der JSP-Teil eingebettet.

### 4.5.1 hello.jsp: Dateneingabe

In dieser JSP-Seite werden in HTML Forms zu Demozwecken eine Reihe von Eingaben gemacht, unter anderem ein Username und eine zweifache Passworteingabe.

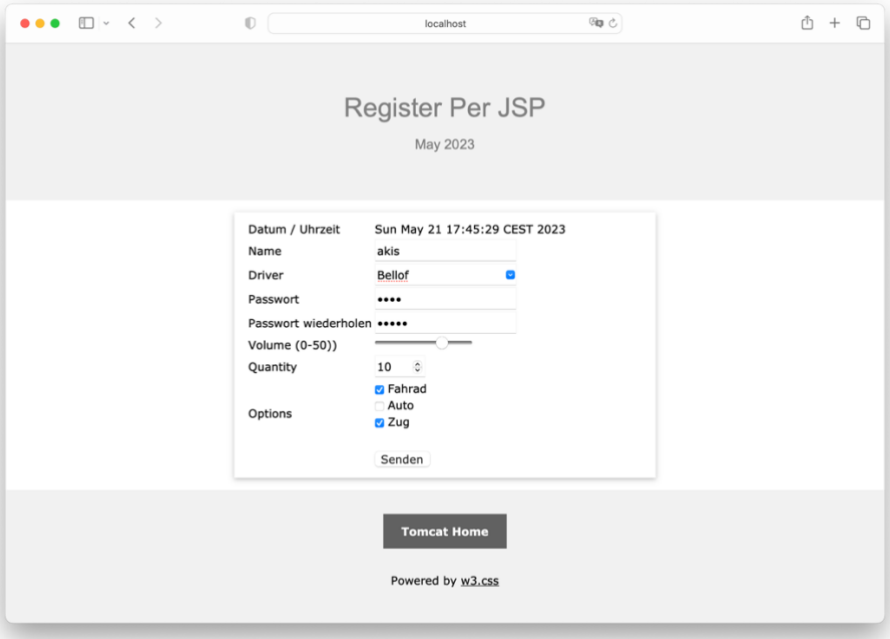
```
<form action="showdata.jsp" method="post">
  <table>
    <tr><td>Datum / Uhrzeit</td>
      <td><%= date %></td>
    </tr>
    <tr><td>Name</td>
      <td><input type="text" name="user"></td>
    </tr>
    <tr>
   <td>Driver</td>
      <td><input type="search" list="Driver" name="driver">
        <datalist id="Driver">
          <option value="Arnoux">
          <option value="Bellof">
          <option value="Cheever">
          <option value="Danner">
          <option value="Fisichella">
          <option value="Glock">
          <option value="Mansell">
          <option value="Stewart">
          <option value="Verstappen">
        </datalist>
      <td>
    </tr>
    <tr><td>Passwort</td>
```

```
        <td><input type="password" name="pw1"></td>
    </tr>
    <tr><td>Passwort wiederholen</td>
        <td><input type="password" name="pw2"></td>
    </tr>
    <tr><td>Volume (0-50)</td>
        <td><input type="range" name="vol" min="0" max="50"></td>
    </tr>
    <tr><td>Quantity</td>
        <td><input type="number" name="quantity" min="0" max="100" step="10" value="30">
    </tr>
    <tr><td>Options</td>
        <td><input type="checkbox" id="vehicle1" name="veh1" value="Bike">
            <label for="vehicle1"> Fahrad</label><br>
            <input type="checkbox" id="vehicle2" name="veh2" value="Car">
            <label for="vehicle2"> Auto</label><br>
            <input type="checkbox" id="vehicle3" name="veh3" value="Train">
            <label for="vehicle3"> Zug</label><br><br>
        </td>
    </tr>
    <tr><td></td>
        <td><input type="submit"></td>
    </tr>
  </table>
```

Das Formular für die Registrierung sendet die Daten per POST an register.jsp.
Im letzten Schritt wird geprüft, ob jemand in der Session eine Nachricht hinterlegt hat. Sie wird angezeigt und zurückgesetzt. Diese wird im Fehlerfall von register.jsp hinterlegt.

```
<%      String msg = (String)session.getAttribute("msg");
        if (msg!=null && !msg.isEmpty()) {
        %>      <hr> <center><% out.print(msg); %> </center><hr>          <%
        }
        session.setAttribute("msg", "");
%>
```

Mit etwas CSS-Zucker sieht die HTML-Seite so aus:




### 4.5.2 register.jsp: Auswertung der Registrierung

Die Datei register.jsp hat die Aufgabe, die Anmeldung zu prüfen.

- Mit request.getParameter() holt sie sich die Eingabeinhalte.
- Sie prüft, ob der Benutzername leer ist oder ob die Passwörter nicht übereinstimmen. In diesem Fall wird die entsprechende Meldung in der Session hinterlegt und der Client zu **newuser.jsp** zurückgeschickt.
- Ist alles gut, wird der eingegeben Benutzername in der Variablen user in der Sitzung hinterlegt und **main.jsp** aufgerufen.

Die Seite benötigt keine Darstellung. Aus diesem Grund könnte man sie vielleicht sogar besser als Servlet implementieren.

```
%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
```

```
<body>
   <%
   String name    = request.getParameter("user");
   String liste   = request.getParameter("driver");
   String pwd1    = request.getParameter("pw1");
   String pwd2    = request.getParameter("pw2");
   String vol     = request.getParameter("vol");
   if (name.isEmpty()) {
      session.setAttribute("msg", "Kein Benutzer" );
      response.sendRedirect("hello.jsp");
    }
   else
      if (!pwd1.equals(pwd2)) {
         session.setAttribute("msg", "Passworteingabe nicht gleich ");
         response.sendRedirect("hello.jsp");
      }
      else {
         session.setAttribute("user", name);
         session.setAttribute("liste",liste);
         session.setAttribute("pass1",pwd1);
         session.setAttribute("pass2",pwd2);
         session.setAttribute("volume",vol);
         response.sendRedirect("main.jsp");
      }
   %>
</body>
</html>
```

### 4.5.3 main.jsp: Die Hauptseite

Die Hauptseite holt nun aus der Session alle Parameter:

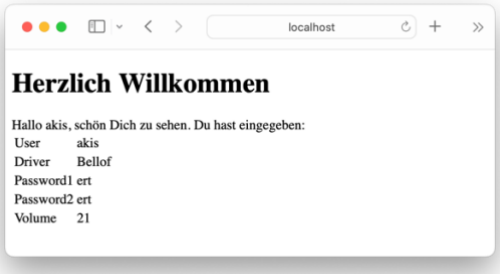
```
<% String user     = (String)session.getAttribute("user"); %>
<% String liste    = (String)session.getAttribute("liste"); %>
<% String pw1      = (String)session.getAttribute("pass1"); %>
```

```
<% String pw2      = (String)session.getAttribute("pass2"); %>
<% String vol      = (String)session.getAttribute("volume"); %>
```

und stellt die Werte mit einer persönlichen Ansprache dar.

```
Hallo <% out.print(user); %>,
schön Dich zu sehen. Du hast
eingegeben:

<table>

 <tr>
    <td>User</td>
    <td><% out.print(user); %>  </td>
 </tr>
 <tr>
    <td>Driver</td>
    <td><% out.print(liste); %>  </td>
 </tr>
 <tr>
    <td>Password1</td>
    <td><% out.print(pw1); %> </td>
 </tr>
 <tr>
    <td>Password2</td>
    <td><% out.print(pw2); %> </td>
 </tr>
 <tr>
   <td>Volume</td>
   <td><% out.print(volume d); %></td>
 </tr>

</table>
```

Im Browser sieht das (ohne CSS-Zucker) recht einfach aus:




Alternativ zu response.jsp, noch showdata.jsp, eine JSP die alle Parameter mit Namen und Werten aus dem Request ausliest und ausgibt.

```
<html>
```

```
<body>
<%@ page import = "java.util.*" %>
<b>Parameters:</b><br>
<%
 Enumeration parameterList = request.getParameterNames();
 while(parameterList.hasMoreElements() )
 {
  String   sName     = parameterList.nextElement().toString();
  String[] sMultiple = request.getParameterValues( sName );
  if(1 >= sMultiple.length )
   out.println( sName + " = " + request.getParameter( sName ) + "<br>" );
  else
   for( int i=0; i<sMultiple.length; i++ )
    out.println( sName + "[" + i + "] = " + sMultiple[i] + "<br>" );
 }
%>
</body>
</html>
```

- Die Methode getParameterNames() vom impliziten JSP-Objekt request wird hier verwendet um alle verfügbaren Formularparameter aus zu lesen.
- Diese Methode liefert eine Enumeration, die die Parameternamen enthält.
- Die Enumeration wird mit der Methode hasMoreElements()durchlaufen, und die Methode nextElement() ruft Parameternamen ab.

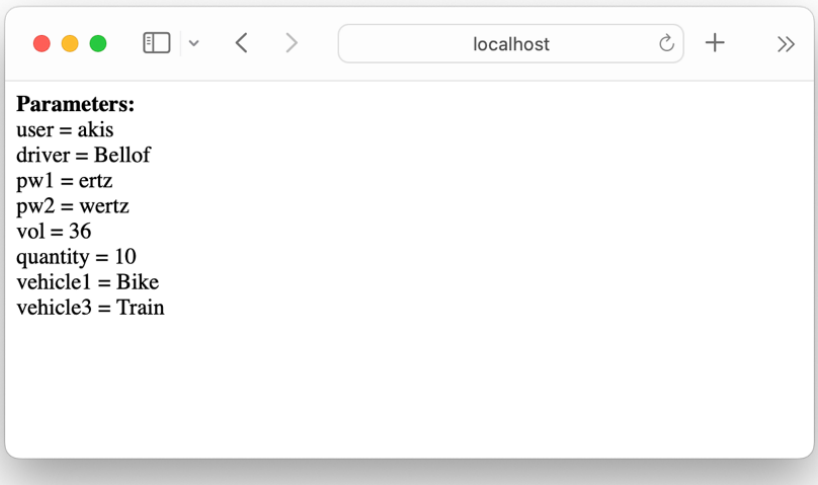
Wenn man in hello.jsp den „action“ Wert für das Formular von

```
<form action="register.jsp" method="post">
```

zu

```
<form action="showdata.jsp" method="post">
```

ändert, gibt es folgende Ausgabe (Beispiel … ohne jeden CSS-Zucker):




Es gibt also mehr als ausreichend, sehr simple und robuste Wege, Parameter aus einem Formular an eine JSP-Seite oder ein Servlet zu übergeben.

# 5. JSP-Actions

Ein weiteres Element von JSP sind JSP-Actions, ein paar davon sind hier schon verwendet worden. Nicht alle verügbaren werden in diesem Abschnitt näher betrachtet.

- JSP-Aktionen sind im Grunde vordefinierte Funktionen.
- Sie verwenden XML-Syntax, und steuern das Verhalten der Servlet-Engine
- Sie können eine Datei dynamisch einfügen, JavaBeans-Komponenten einbinden, den Benutzer auf eine andere Seite weiterleiten oder HTML für ein Java-Plugin generieren.

Es gibt nur eine Syntax für das Action-Element, da es nur in XML-Standard verfügbar ist:

```
<jsp:action_name attribute = "value" />
```

## 5.1 Übersicht

In der folgenden Tabelle sind die verfügbaren JSP-Aktionen aufgeführt:

| Syntax & Zweck | | |
|---|---|---|
| 1 | jsp:include | Bindet eine Datei zum Zeitpunkt des requests der Seite ein. |
| 2 | jsp:useBean<br>jsp:setProperty<br>jsp:getProperty | Siehe Kapitel 5. |
| 5 | jsp:forward | Leitet den request auf eine neue Seite weiter. |
| 6 | jsp:plugin | Generiert browserspezifischen Code, der Tags für ein Java-Plug-in erstellt.Wird hier nicht näher betrachtet. |
| 7 | jsp:element<br>jsp:body<br>jsp:attribute | Dynamisches Definieren von XML-Elementen (Text und Attribute). Werden hier nicht näher betrachtet. |
| 10 | jsp:text | Ist in erster Linie für JSP-Dokumente (JSP-Seiten in XML-Syntax) gedacht. Weder JSP-Aktionselemente noch Skripting-Elemente sind zulässig.<br>Da XML-Dokumente und EL-Ausdrücke nicht der explizite Fokus dieses Dokumentes sind, wird die Aktion nicht weiter betrachtet. |

## 5.2 Attribute

Zwei Attribute sind allen JSP-Aktionen gemein, das Id Attribut, und das Scope Attribut.

**Id** Das id-Attribut identifiziert eine Action-Element eindeutig und ermöglicht den Verweis auf die Aktion innerhalb der JSP-Seite. Wenn die Action eine Instanz eines Objekts erstellt, kann der id-Wert verwendet werden, um über das implizite PageContext-Objekt darauf zu verweisen.

**Scope** Das scope-Attribut legt den Geltungsbereich fest, und hat vier mögliche Werte:

- page,
- request,
- session oder
- application.

## 5.3 <jsp:include>

Mit der include-Aktion werden Dateien in die zu generierende Seite eingefügt:

```
<jsp:include page = "relative URL" flush = "true" />
```

Im Gegensatz zur include-Direktive, die die Datei zum Zeitpunkt der Übersetzung der JSP Seite in ein Servlet einfügt, , fügt diese Aktion die Datei zu dem Zeitpunkt ein, an dem die Seite angefordert wird. Attribute sind:

- **page** die relative URL der Seite, die eingebunden werden soll.
- **flush** das boolesche Attribut bestimmt, ob der Puffer der eingeschlossenen Ressource geleert wird, bevor sie eingebunen wird.

Denken wir uns eine **date.jsp** Datei:

```
<p>Today's date: <%= (new java.util.Date()).toLocaleString()%></p>
```

Dann kann sie in einer weiteren JPS-Datei wie folgt inkludiert werden:

```
<html>
  <head><title>The include Action Example</title </head>
  <body>
    <center>
      <h2>The include action Example</h2>
```

```
        <jsp:include page = "date.jsp" flush = "true" />
      </center>
    </body>
</html>
```

Die Dateien müssen beide in dem Verzeichnis der Web-Anwendung liegen.

## 5.4 <jsp:forward>

Die forward-Aktion beendet die Ausgabe der aktuellen Seite und leitet die Anforderung an eine andere Ressource weiter. Die Syntax ist:

```
<jsp:forward page = "Relative URL" />
```

**page** Sollte aus einer **relativen** URL einer anderen Ressource bestehen, z. B. einer statischen Seite, einer anderen JSP-Seite oder einem Java-Servlet.

```
<jsp:forward page="/srv1"/>

<jsp:forward page="home.jsp"/>

<jsp:forward page="index.html"/>
```

Die **relative** URL sieht aus wie ein Pfad – sie darf aber keinen Protokollnamen, keine Portnummer oder keinen Domänennamen enthalten. Die URL kann absolut oder relativ zur aktuellen JSP-Datei sein. Wenn er absolut ist (beginnend mit einem /), wird der Pfad vom Application-Server aufgelöst.

# 6. JavaBeans und JSP

In JSP-Seiten können JavaBeans sehr effizient, ohne großen syntaktischen Zucker verwendet werden. Für den Fall, dass das Konzept von JavaBeans nicht bekannt ist, folgt im nächsten Abschnitt ein kurzer Abriss des Konzeptes, bevor es mit der Verwendung von JavaBeans in JSP weiter geht.

## 6.1 JavaBeans

### 6.1.1 Prinzip

Ausgangspunkt für JavaBeans waren technische Fragen, die sich um zwei Punkte drehten:

- Wann ist eine JAVA-Klasse effizient und uneingeschränkt wieder verwendbar?
- Wie sieht ein Standard für Wiederverwendbarkeit aus?

Die Überlegungen führten zu einem leichtgewichtigen Standard/ Konvention und einer API.

Eine "Bean“ ist eine Klasse, die ein oder mehrere Objekte in einer **standardisierten** Art zur einfachen Wiederverwendbarkeit kapselt. JavaBeans müssen als Teil des Standards:

- serialisierbar sein,
- über einen Null-Argument-Konstruktor verfügen,
- und den Zugriff auf „Properties“ (Attribute der Klasse) mithilfe von Getter- und Setter-Methoden ermöglichen.

Die Standardisierung ermöglicht es, „Beans“ als generische Softwarekomponenten zu behandeln und sie ohne Erstkonfiguration einzubinden ohne dass interne Implementierungsdetails bekannt sind.

Neben der Konvention gibt es, mit dem Java package java.beans, eine Reihe von Klassen und Schnittstellen um“ Beans“ wiederzuverwenden, ersetzen und verbinden zu können.

JavaBeans können sichtbare Objekte, wie beispielsweise AWT-Komponenten, oder unsichtbare Objekte, wie Stacks, Listen, Warteschlangen sein.

### 6.1.2 JavaBeans Konventionen

Kurz ein weitergehender Blick auf den „Standard“. Um als JavaBean zu funktionieren, muss eine Klasse ein paar Konventionen bezüglich der Benennung, des Aufbaus und des Verhaltens seiner Methoden einhalten:

- Es stellt einen standardmäßigen Konstruktor **ohne** Argumente bereit.
- Es muss das serializable Interface implementieren.
- Es kann eine Reihe von **Properties** (Attribute) haben,
    - Das Property kann von einem beliebigen Java-Datentyp sein.
    - Das Property kann lesend, schreibend, schreibgeschützt oder schreibgeschützt sein.
- Alle Properties einer Bean betreiben *information hiding:* Der Zugriff ist nur über die „getter“ und „setter“ Methoden möglich.

    - Für eine Eigenschaft foo heißt die lesende Operation (Getter) getFoo**.**
      Bei booleschen Variablen ist auch isFoo möglich, was vorgezogen wird.
    - Die schreibende Operation (Setter) heißt setFoo.
    - Indizierte Eigenschaften besitzen jeweils zwei Getter und Setter: Einen für die Gesamtheit, einen für einen bestimmten Index.
    - Eine schreibgeschützte Eigenschaft besitzt keinen (öffentlichen) Setter

Ein Beispiel für eine solche Klasse ist:

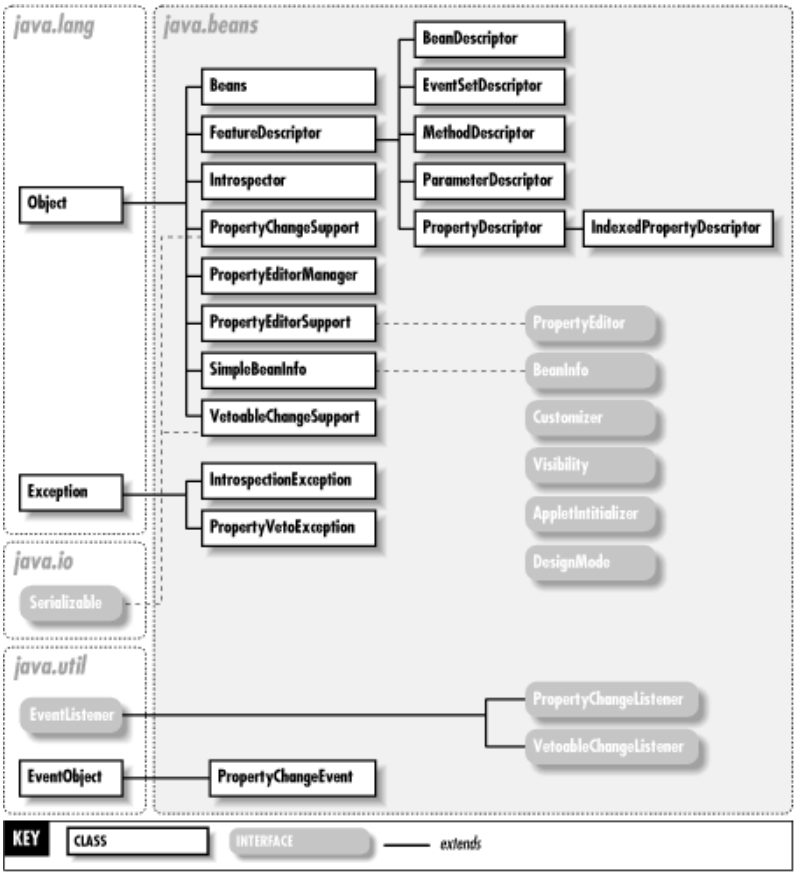
```
public class StudentsBean implements java.io.Serializable {

  private String firstName = null;

  private String lastName = null;

  private int age = 0;


  public StudentsBean() { }


  public String      getFirstName(){ return firstName; }


  public String      getLastName(){ return lastName;  }


  public int                  getAge()                { return age;     }


    public void setFirstName(String firstName) {
```

```
                this.firstName = firstName;
    }
    public void         setLastName(String lastName){
      this.lastName = lastName;
    }
    public void setAge(Integer age){
      this.age          = age;
    }
  }
```

Diese Klasse könnte man als „Bean“ wiederverwenden. Um das wirklich sinnvoll machen zu könne braucht es aber schon noch ein paar Werkzeuge, die werden im Java SDK in einem package java.beans angeboten.

### 6.1.3 JavaBeans API

java.beans enthält Klassen und Schnittstellen, die sich auf JavaBeans-Komponenten beziehen. Die meisten Klassen und Schnittstellen sind für Anwendungen gedacht, die Beans manipulieren, und nicht für die Beans selbst. Für die JSP-Programmierung ist das nicht wirklich relevant, aber es gehört hier dazu.
Das Klassen- und Interface-Model des packages sieht wie folgt aus:




Um für andere Anwendungen verwendbar zu sein, sollten eine JavaBean auskunftsfähig sein – das wird *Introspektion* genannt. Introspektion ermöglich einer anderen Anwendung, z. B. einem Design-Tool, ermöglicht, Informationen über eine Komponente zu erhalten.
Das package java.beans stellt solche Methoden zur Verfügung (interface BeanInfo), um Informationen zu der Bean abzulegen und aufzurufen.

## 6.2 JSP und JavaBeans: <jsp:useBean>

<jsp:useBean> deklariert eine JavaBean für die Verwendung in einer JSP. Damit wird die Bean zu einer Skripting-Variablen, auf die sowohl von Scripting-Elementen als auch von anderen benutzerdefinierten Tags zugegriffen werden kann, die in der JSP verwendet werden.
Die vollständige Syntax für das useBean-Tag lautet wie folgt:

```
<jsp:useBean id = "Name der Bean" scope = "Bereich der Bean" typeSpec/>
```

- Das scope Attribut kann eine page, ein request, eine session oder eine application sein.

- Das id Attributkann einen beliebiger Wert haben, solange es sich ein eindeutiger bean-Namen in derselben JSP ist.

Beispiel:

```
<html>
  <head>title>useBean Beispiel</title> </head>
  <%@ page import="java.util.Date" %>
  <body>
    <jsp:useBean id = "date" class = "java.util.Date" />
    <p>Datum/Zeit ist <%= date %>
  </body>
</html>
```

Die Klasse java.util.Date bietet Konstruktoren und Methoden für den Umgang mit Datum und Uhrzeit in Java. java.util.Date implementiert u.a. das Interface serializable und erfüllt alle Konventionen um als JavaBean genutzt werden zu können.

### 6.2.1 Zugriff auf JavaBeans-Properties

<jsp:getProperty/> wird verwendet, um auf get-Methoden der bean zuzugreifen.
<jsp:setProperty/> wird verwendet um auf set-Methoden zuzugreifen.

Die vollständige Syntax der JSP-Aktionen:

```
<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">
  <jsp:setProperty name ="bean's id" property="name" value= "value"/>
  <jsp:getProperty name ="bean's id" property="name"/>
</jsp:useBean>
```

- Name ist die id einer JavaBean, die zuvor durch die useBean-Aktion deklariert wurde.
- property ist der Name der get- / et-Methode, die aufgerufen werden soll.

Das folgende Beispiel zeigt wie auf die Attribute/Properties einer JavaBean zugegriffen wird:

```
<html>
  <head><title>get und set properties Beispiel</title> </head>
  <body>
    <jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">
      <jsp:setProperty name="students" property="firstName" value="Zara"/>
      <jsp:setProperty name="students" property="lastName" value ="Ali"/>
```

```
        <jsp:setProperty name="students" property ="age" value="10"/>
    </jsp:useBean>
    <p> Vorname: <jsp:getProperty name="students" property= firstName"/> </p>
    <p> Nachname: <jsp:getProperty name="students" property="lastName"/></p>
            <p> Alter:  <jsp:getProperty name = "students" property = "age"/></p>
  </body>
</html>
```

Achtung: StudentsBean.class muss im CLASSPATH verfügbar sein.

# 7. Expression Language ${}

*Expression language* (EL) wurde in JSP 2.0 eingeführt. Der Hauptzweck besteht darin, den Zugriff auf Daten aus JavaBeans und impliziten Objekten zu vereinfachen.

EL wird hauptsächlich verwendet, um Java-Code in der JSP zu „sparen“. Um auf einen Request-Parameter zuzugreifen, schreibt man in JAVA:

```
<% uname=request.getParameter("uname");%>
```

In EL ist das einfacher:

```
${param.uname}
```

## 7.1 Basis Syntax

Die Syntax von EL ist:

```
${expression}
```

Alles, was in den geschweiften Klammern steht ist, wird zur Laufzeit evaluiert und an den Ausgabestream gesendet.

## 7.2 Operatoren

### 7.2.1 Arithmetisch / logische Operatoren

EL enthält arithmetische und logische Operatoren und bietet so die Möglichkeit beliebige Ausdrücke (*Expressions*) an den response-Stream zu geben.

Beispiel:

```
<html>
<head>
 <title>Expression language example1</title>
</head>
<body>
${1<2}
${1+2+3}
</body>
</html>
```

Erzeugt die Ausgabe: True 6

Man kann damit also den Wert eines Ausdruckes schnell und einfach in den response-Stream einfügen. JSPs Expression Language (EL) unterstützt die allermeisten arithmetischen und logischen Operatoren aus Java.

| | | | |
|---|---|---|---|
| + | - | * | / oder div |
| % oder mod | == oder eq | **!= or ne** | **< or lt** |
| **> or gt** | **> or gt** | **<= or le** | **>= or ge** |
| **&& or and** | **|| or or** | **|| or or** | **Empty (==Null)** |

Um Ausdrücke zu gruppieren werden Klammern verwendet – nicht wirklich überraschend …

${(1 + 2) * 3} ist gleich 9, aber ${1 + (2 * 3)} ist gleich 7.

### 7.2.2 Zugriffs- Operatoren

Die gebräuchlichsten Operatoren in JSP EL sind **.** und **[]**. Mit diesen beiden Operatoren wird auf Attribute von Java Beans und integrierten JSP-Objekten zugegriffen.

**Syntax „.“ (Access-Operator**):

```
${leftvariable.rightvariable}
```

Beispiel:

```
${param.uname}
${pageContext.session}
```

**Syntax „[]“ (Collection Access-Operator):**

```
${variable[index]}
```

Beispiel Arrays:

```
<%
String[] s={“CoreJava”,”Spring”, “Hibernate”,”WebServices”}
pageContext.setAttribute(“s”,s);
%>
```

```
${s[0]}          ist   CoreJava
${s[1]}          ist   Spring
${s[2]}          ist   Hibernate
${s[3]}          ist   WebServices
${s[100]}  ist   Empty
```

Beispiel List:

```
<%
Java.uil.ArrayList al=new java.util.ArrayList();
al.add(“student1”);
al.add(“student2”);
al.add(“student3”);
pageContext.setAttribute(‘list”,1,2);
%>
```

```
${list[0]}     ist   student1
${list[“1”]}  ist   student2
${list[100]} ist   blank space
```

### 7.2.3 Deaktivieren von expressions

Um die Auswertung von EL-Ausdrücken zu unterdrücken, wird das Attribut **isELIgnored** der page-direktive verwendet:

```
<%@ page isELIgnored = "true|false" %>
```

Die gültigen Werte dieses Attributs sind true und false. Falls der Wert true ist, werden EL-Ausdrücke ignoriert, wenn sie in statischem Text oder Tag-Attributen angezeigt werden. Wenn der Wert false ist, werden EL-Ausdrücke vom Container ausgewertet.

## 7.3 Verwendung in Java Beans

Um in einem JSP Action Tag einem Attribut einen Wert zuzuweisen, werden in der Regel Strings verwendet:

```
<jsp:setProperty name = "box" property = "perimeter" value = "100"/>
```

Mit JSP EL können **expressions** für jeden dieser Attributwerte angeben werden. Die obige Syntax des **<jsp:setProperty>**-Tag kann mit **expressions** wie folgt umgesetzt werden:

```
<jsp:setProperty name = "box" property = "perimeter"
  value = "${2*box.width+2*box.height}"/>
```

Wenn der JSP-Compiler **${}** erkennt, generiert er Code zum Auswerten des Ausdrucks und ersetzt den Wert des Audruckes .

JSP EL-Ausdrücke werden auch in Templates für ein Tag verwenden. Das Tag **<jsp:text>** fügt einfach seinen Inhalt in den Hauptteil einer JSP ein.

```
<jsp:text>
  <h1>Hello JSP!</h1>
</jsp:text>
```

Im Hauptteil von a **<jsp:text>** tag kann auch die **${}** Syntax für Attribute verwendet werden:

```
<jsp:text>
```

```
    Box Perimeter is: ${2*box.width + 2*box.height}
</jsp:text>
```

## 7.4 Zugriff auf implizite Objekte mit EL

Der smarteste Einsatz von EL ist der vereinfachte Zugriff auf implizite Objekte und deren Attribute.

| Nr. | Implizites Objekt | |
|---|---|---|
| 1 | **pageScope** | Variablen des page Objektes |
| 2 | **requestScope** | Variablen des request Objektes |
| 3 | **sessionScope** | Variables des session Objektes |
| 4 | **applicationScope** | Variablen des application Objektes |
| 5 | **Param** | Request Parameter als Strings |
| 6 | **paramValues** | Request Parameter als String Collections |
| 7 | **Header** | HTTP request headers als Strings |
| 8 | **headerValues** | HTTP request headers als String Collections |
| 9 | **initParam** | Context-Initialisierungsparameter |
| 10 | **Cookie** | Cookie Werte |
| 11 | **pageContext** | Das JSP PageContext Object für die aktuelle Seite |

### 7.4.1 pageContext-Objekt

Das pageContext-Objekt ermöglicht den Zugriff auf das pageContext-JSP-Objekt. Über das pageContext-Objekt können man auf das request Objekt zugreifen.

Um z. B. auf die eingehende Query für eine Request zuzugreifen, kann die folgende expression verwendet werden:

```
${pageContext.request.queryString}
```

### 7.4.2 Scope Objekte

Die Objekte

- **pageScope,**
- **requestScope,**
- **sessionScope**
- und **applicationScope**

bieten Zugriff auf ihre Variablen / Attribute.

Beispiel:

```
<%
request.getSession().setAttribute("CarsHTML", Cars);
%>
<a href="/F1-CarsByDriver/${sessionScope.CarsHTML}.html"> … </a>
```

setzt den Wert der session Variablen CarsHTML in den Link ein. Dies kann in der gleichen JSP Datei, oder in einer JSP in der gleichen session genutzt werden. Im übrigen funktioniert das sogar mit ${CarsHTML} – solange der Ausdruck eindeutig ist.

### 7.4.3 param- und paramValues-Objekte

Die param- und paramValues-Objekte geben normalerweise Zugriff auf die Parameterwerte über die **Methoden**

```
request.getParameter und request.getParameterValues
```

Um mit EL auf einen Parameter mit dem Namen order zuzugreifen, verwendet man den **Ausdruck**

```
${param.order} oder ${param["order"]}.
```

**Beispiel:**

In index.jsp gibt der Benutzer, einen Namen und eine Nummer ein. In display.jsp werden die eingegebenen Details mit EL aus dem param-Objekt abgerufen.

index.jsp:

```
…
<form action="display.jsp">
Student Name: <input type="text" name="stuname" /><br>
Student RollNum:<input type="text" name="rollno" /><br>
<input type="submit" value="Submit Details!!"/>
</form>
..
```

display.jsp:

```
<html>
<head>
<title>Display Page</title>
</head>
<body>
 Student name is ${param.stuname } <br>
 Student Roll No is ${param.rollno}
</body>
</html>
```

**Beispiel:**

In form.html werden mehrere Optionen ausgewählt. In first.jsp werden die einge-gebenen Optionen mit EL aus dem **paramValues**-Objekt abgerufen.

form.html:

```
…
<body>
   <form method="get" action="first.jsp">
    <br>
```

```
        <br> Name <input type="text" name="uname" /><br>
        <br> Food Items <select size="3" multiple="true" name="food">
         <option value="Orange">Orange</option>
         <option value="Apple">Apple</option>
         <option value="Grapes">Grapes</option>
        </select> <input type="submit" value="Display" />
       </form>
    </body>
    …
```

first.jsp

```
    ..
    <body>
        ..
      <br> User Name.....${param.uname}
      <br> Foot Items...
        <br> ${paramValues.food[0]}
        <br> ${paramValues.food[1]}
        <br> ${paramValues.food[2]}
        ..
    </body>
    ..
```

### 7.4.4 header und headerValues Objekte

Die **header** und **headerValue**-Objekte liefern Zugriff auf die Headerwerte, die auch über die request.getHeader-Methode und die request.getHeaders-Methode verfügbar sind.

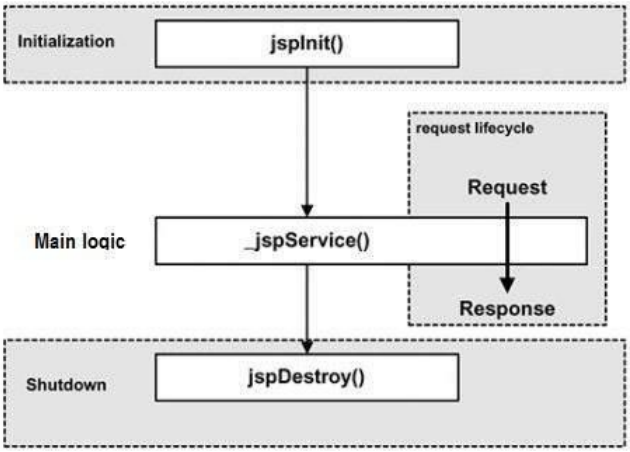
Um beispielsweise auf einen Header mit dem Namen user-agent zuzugreifen, verwendet man den Ausdruck ${header.user-agent} oder ${header["user-agent"]}.

Das **header**-Objekt gibt einzelne Strings zurück, während das **headerValues**-Objekt ein array von Strings zurückgibt.

# 8. Lebenszyklus einer JSP

Der Low-Code-Charakter von JSP wird ersichtlich bei ihrem Lebenszyklus. Der JSP-Lebenszyklus ist der eines Servlets - mit dem einem zusätzlichen Schritt, eine JSP in ein Servlet zu übersetzen:

- Compilation
- Initialization
- Execute
- Execute




## 8.1 Compilation

Wenn ein Browser eine JSP anfragt, prüft die JSP-Engine zunächst, ob die Seite kompiliert werden muss. Wenn die Seite noch nie kompiliert wurde oder wenn die JSP seit der letzten Kompilierung geändert wurde, kompiliert die JSP-Engine die Seite.

Das Kompilieren umfasst drei Schritte:

- Parsen der JSP.
- Umwandeln der JSP in ein Servlet.
- Kompilieren des Servlets.

Wie gesagt – JSP sind eine „Makrosammlung“ auf Servlets, sie verhalten sich zueinander ähnlich wie Latex und Tex ….

## 8.2 Initialization

Wenn ein Container (Application-Server) eine JSP lädt, ruft er die Methode jspInit() auf, bevor er den request bedient. Wenn eine JSP-spezifische Initialisierungen durchgeführt müssen, kann die jspInit()-Methode überschrieben werde.

```
public void jspInit(){
  Initialisierungscode...
}
```

Initialisierung werden eigentlich nur einmal ausgeführt. Analog zur Servlet-Init-Methode ist dies der Platz um Datenbankverbindungen zu initialisieren, Dateien zu öffnen, und „Look Up Tables“ zu erstellen.

## 8.3 Execution

Diese Phase des JSP-Lebenszyklus beinhaltet alle Interaktionen mit requests, bis die JSP zerstört wird. Immer wenn ein Browser eine JSP anfordert und die Seite geladen und initialisiert wurde, ruft die JSP-Engine die Methode **_jspService()** in der JSP auf.

Die Methode _jspService() verwendet **HttpServletRequest** und **HttpServletResponse** als Parameter wie folgt:

```
void _jspService(HttpServletRequest-Anforderung, HttpServletResponse-Antwort) {
  Service-Handling-Code...
}
```

Die Methode _jspService() einer JSP wird auf einen request hin aufgerufen. Diese Methode generiert die Antwort für diese request verantwortlich, und diese Methode ist auch für das Generieren von reponses für alle sieben HTTP-Methoden (GET, POST, DELETE usw.) verantwortlich.

## 8.4 Cleanup

Die letzte Phase des JSP-Lebenszyklus entfernt die JSP aus einem Container. Die Methode **jspDestroy()** ist das JSP-Äquivalent der destroy-Methode für Servlets.

**jspDestroy** kann überschrieben werden, falls individuelle Aufräumarbeiten nötig sind, z. B. das Freigeben von Datenbankverbindungen oder das Schließen geöffneter Dateien.

```
public void jspDestroy() {
  Der Bereinigungscode befindet sich hier.
}
```

# 9. Servlets

Im Verlauf dieses Dokumentes sind wir immer wieder und in den letzten Abschnitten immer näher an das Arbeiten mit und Entwickeln von Servlets gekommen.

**Java™-Servlets** sind Java-Klassen, die im Kontext einer Webanwendung auf HTTP-requests antworten. Sie erhalten vom Webserver einen request, bearbeiten diesen und senden das Ergebnis (response) wieder an den Webserver. Java™-Servlets sind also Java-Programme, die auf einem Java-Applicationserver (hier im Dokument Tomcat) ausgeführt werden und dadurch die Funktionalität des Web-Servers erweitern.

Dabei ist die Servlet-Technologie nicht auf das HTTP-Protokoll beschränkt. In der Regel wird sie so verwendet, aber „Servlet“ ist eine allgemeines JAVA-Interface und HttpServlet ist eine Erweiterung dieser Schnittstelle, die HTTP-spezifische Funktionen hinzufügt, wie z. B. doGet und doPost usw. Schließlich dient die Servlet-Technologie als Basis für weitere Web-technologien wie JSP, Spring MVC und andere.

## 9.1 Die HttpServlet Klasse

Die HttpServlet Klasse erweitert die generische Servlet Klasse und implementiert ein sog. serializable interface. Sie stellt http spezifische Methoden wie doGet, doPost, doHead, doTrace bereit.

## 9.2 Lifecycle

Den Lifecycle eines Servlets haben wir bereits im Kapitel x.x kennengelernt.

## 9.3 Methoden von HttpServlet

In der HttpServlet Klasse gibt es eine Reihe von Methoden:

public void service(ServletRequest req,ServletResponse res)

Sendet die Anforderung an die protected void service()- Methode , indem das Request- und Response objekt in den HTTP-Typ konvertiert wird.

protected void service(HttpServletRequest req, HttpServletResponse res)

empfängt den request von der  publuc Service-Methode und sendet den Request an die doXXX()-Methode in Abhängigkeit vom eingehenden HTTP-Anforderungstyp.

protected void doGet(HttpServletRequest req, HttpServletResponse res)

verarbeitet den GET- Request. Er wird vom Webcontainer aufgerufen.

protected void doPost(HttpServletRequest req, HttpServletResponse res)

verarbeitet den POST- Request. Er wird vom Webcontainer aufgerufen.

protected void doHead(HttpServletRequest req, HttpServletResponse res)

behandelt den HEAD- Request. Er wird vom Webcontainer aufgerufen.

protected void doOptions(HttpServletRequest rq, HttpServletResponse rs)

behandelt den OPTIONS- Request. Er wird vom Webcontainer aufgerufen.

protected void doPut(HttpServletRequest req, HttpServletResponse res)

behandelt den HEAD- Request. Er wird vom Webcontainer aufgerufen.

protected void doTrace(HttpServletRequest req, HttpServletResponse res)

behandelt den TRACE- Request. Er wird vom Webcontainer aufgerufen.

protected void doDelete(HttpServletRequest req, HttpServletResponse res)

behandelt den DELETE-Request. Er wird vom Webcontainer aufgerufen.

protected long getLastModified(HttpServletRequest req)

gibt die Uhrzeit zurück, an der HttpServletRequest zuletzt seit dem 1.1.1970 0:00 GMT geändert wurde.

## 9.4 Ausgaben

## 9.5 Beispiel

```
import jakarta.servlet.http.HttpServlet;
```

```
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

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

 protected void doGet(HttpServletRequest request, HttpServletResponse response)
 {   String forename  =  request.getParameter("Driver2forename");
     String Driver    =  request.getParameter("Driver2");
     String Comm      =  request.getParameter("Comm");

     String Query     =  Tools.GetSQLQuery(Comm,forename,Driver,"","");
     String SQLRes    =  Tools.getFromDB(Query);

     PrintWriter out  =  response.getWriter();
     out.print(SQLRes);
 }
}
```

## 9.6 Ablauf

1. Der Server prüft, ob das Servlet das erste Mal angefordert wird. Falls **ja,** führt der Container folgende Tasks aus:

   - lädt die Servlet Klasse,
   - instantiiert die Servlet Klasse,
   - ruft die init() Methode mit einem ServletConfig Objekt auf

2. Ruft die public service() Methode mit den request und response Objekten auf.

3. Der Webcontainer ruft die destroy-Methode auf, wenn das Servlet entfernt werden muss, z. B. beim Beenden des Servers oder beim „Un-Deployment“ des Projekts.