

Veranstaltung „Anwendungssysteme 2“

Systemaufbau

1	Systeme – Stand SoSe 2018	2
1.1	Eclipse.....	2
1.2	Anwendungsserver TomCat	2
1.3	Datenbank	2
1.4	Eigene Pfade	2
2	Schnellaufbau.....	4
2.1	Aufbau mit Programmversionen	4
2.2	Aufbau mit bereits installierten Programmen	4
3	Ausführlichere Infos, Stand 2012	5
3.1	Erster Start des Workspaces.....	5
3.2	Erster Start des Workspaces (alternativ).....	5
3.3	Spätere Starts des Workspaces.....	6
4	Tomcat Server in der Eclipse-Basiskonfiguration.....	7
4.1	Öffnen der Eclipse-Basiskonfiguration.....	7
4.2	Installierte Server angeben.....	8
4.3	Eingetragene Server überprüfen	10
5	Eclipse-Projektkonfiguration.....	13
5.1	Projekt anlegen.....	13
5.2	Bibliotheken im Java-Kontext einbinden.....	16
5.3	Bibliotheken im Web-Kontext einbinden.....	20
6	Daten in den Workspace kopieren.....	21
6.1	Java-Packages und Klassen	21
6.2	WebContent	21
6.3	Zugangsdaten für Datenbank	21
6.4	Ergebnis.....	21
7	Datenbankzugriff mit Eclipse	22
7.1	„Driver Definition“ in der Eclipse-Basiskonfiguration.....	22
7.2	Erzeugen einer „Database Connection“ im Data Source Explorer.....	24
7.3	Anmelden an der Datenbank	27
7.4	Arbeiten mit der Datenbank.....	27

1 Systeme – Stand SoSe 2018

Die hier angegebenen Systeme sind auf den Pool PCs installiert. Bei einer Installation auf einem eigenen PC oder Laptop sind diese ebenfalls zu installieren. Falls andere Komponenten oder Versionen verwendet werden, liegt es in der Verantwortung des Benutzers, ob diese Komponenten oder Versionen zusammenspielen.

1.1 Eclipse

Version: Eclipse Neon (Eclipse 4.6)

Programmverzeichnis im Pool: C:\Programme\eclipse

Workspace-Verzeichnis: Z:\Eclipse-Workspace-2018-SoSe (für spätere Semester anpassen)

1.2 Anwendungsserver TomCat

Tomcat-Version: Tomcat v8.0.xx (die genaue Version kann variieren)

Programmverzeichnis: C:\Programme\Apache\apache-tomcat-8.0.xx

1.3 Datenbank

1.3.1 Datenbanksystem und Datenbanktreiber

Version: PostgreSQL 9.5

Programmverzeichnis: C:\Programme\PostgreSQL\9.5

Verzeichnis der Java-Bibliotheken: C:\Programme\PostgreSQL

Namen der relevanten Java-Bibliotheken (JDBC-Treiber): postgresql-9.4.1212.jar

Datenbankname: BWUEBDB

1.3.2 Datenbank-Zugangsdaten

Zugangsdaten zur PostgreSQL-Datenbank im Pool sind die folgenden. Das Schema setzt sich zusammen aus Veranstaltungskürzel und Matrikelnummer:

Variable	Beispielwert
dbDrivername	org.postgresql.Driver
dbURL	jdbc:postgresql://143.93.200.243:5432/BWUEBDB
dbUserId	user1
dbPassword	pgusers
dbSchema	z.B. bw440_654321

1.4 Eigene Pfade

Bei Installation auf einem eigenen PC oder Laptop können diese Verzeichnispfade abweichen und sollten aufgeschrieben werden:

	Verzeichnispfad
Eclipse-Software	
Eclipse-Workspace	
Tomcat	
PostgreSQL-JAR-File	

DB-Driver Class	
-----------------	--

Die Zugangsdaten zur Datenbank können auf einem eigenen PC / Laptop ebenfalls abweichen und sollten aufgeschrieben werden:

Variable	Erklärung	Eigener Wert
dbDrivername	z.B. org.postgresql.Driver	
dbUserId	Id eines DB-Users auf dem Betriebssystem	
dbPassword	Passwort zur angegebenen Userid	
dbURL	z.B. jdbc:postgresql://localhost:5432/BWUEBDB	
dbSchema	Veranstaltungskürzel_Matrikelnummer, z.B. bw441_654321	

2 Schnellaufbau

2.1 Aufbau mit Programmversionen

Eclipse downloaden von <http://eclipse.org>

Eclipse installieren: Installationsarchiv entpacken, z.B. nach C:\Programme\eclipse

TomCat Archiv downloaden von <http://tomcat.apache.org>: TomCat Core reicht, nicht den Windows-Installer nehmen

TomCat installieren: Nur das Archiv auspacken, z.B. nach C:\Programme\apache

PostgreSQL downloaden von <https://www.postgresql.org/>

PostgreSQL installieren, z.B. nach C:\Programme\PostgreSQL

PostgreSQL-JDBC-Treiber downloaden: <https://jdbc.postgresql.org/download.html>

PostgreSQL-JDBC-Treiber bereitstellen, z.B. im Programmverzeichnis der PostgreSQL-Datenbank

2.2 Aufbau mit bereits installierten Programmen

2.2.1 Eclipse-Workspace

Eclipse Workspace anlegen: Verzeichnis anlegen, z.B. Z:\Eclipse-Workspace-2018-SoSe

Eclipse starten und Workspace auswählen.

TomCat in den Eclipse-Workspace: einbinden:

Eclipse-Menü: Windows → Preferences → Server → Runtime Environments → TomCat muss angezeigt werden, andernfalls

→ Add... → Apache/Apache Tomcat v8.0 → Tomcat installation directory: wie oben angegeben

2.2.2 Eclipse-Projekt

Eclipse-Projekt anlegen: New → Dynamic Web Project, z.B. Name „BW4S“

Tomcat Server Library in den Java Build Path einbinden:

Rechts-Klick auf das Projekt → Properties → Java Build Path → Libraries → Add Library... → Server Runtime → Apache Tomcat v8.0 (wenn Apache hier nicht angeboten wird, zurück 2.2.1).

PostgreSQL-JDBC-Treiber in den Java Build Path einbinden (Beginn wie oben):

Rechts-Klick auf das Projekt → Properties → Java Build Path → Libraries → Add External JARs... → Verzeichnisname des PostgreSQL-JDBC-Treibers (vgl. 1.3 bzw. 1.4) → bestätigen mit Apply oder ok

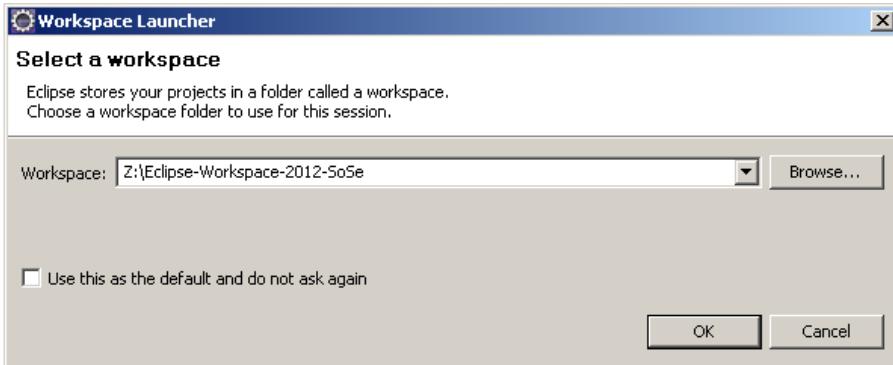
PostgreSQL-JDBC-Treiber in die Deployment Assembly einbinden:

Rechts-Klick auf das Projekt → Properties → Deployment Assembly → Add... → Java Build Path Entries → Dateiname des JDBC-Treibers (JAR-File), z.B. postgresql-9.4.1212.jar

3 Ausführlichere Infos, Stand 2012

3.1 Erster Start des Workspaces

Verzeichnis des Workspace beim Start von Eclipse angeben



Das Verzeichnis wird von Eclipse automatisch erzeugt und kann danach auch im Explorer gefunden werden.



Beim ersten Start eines Workspaces erscheint zunächst das Begrüßungsfenster, das mit Klick auf „Workbench“ verlassen wird:



3.2 Erster Start des Workspaces (alternativ)

Das Verzeichnis (vgl. Fehler! Verweisquelle konnte nicht gefunden werden.) kann auch manuell im Explorer angelegt werden und kann dann beim Start von Eclipse ausgewählt werden (vgl. 3.3).

3.3 Spätere Starts des Workspaces

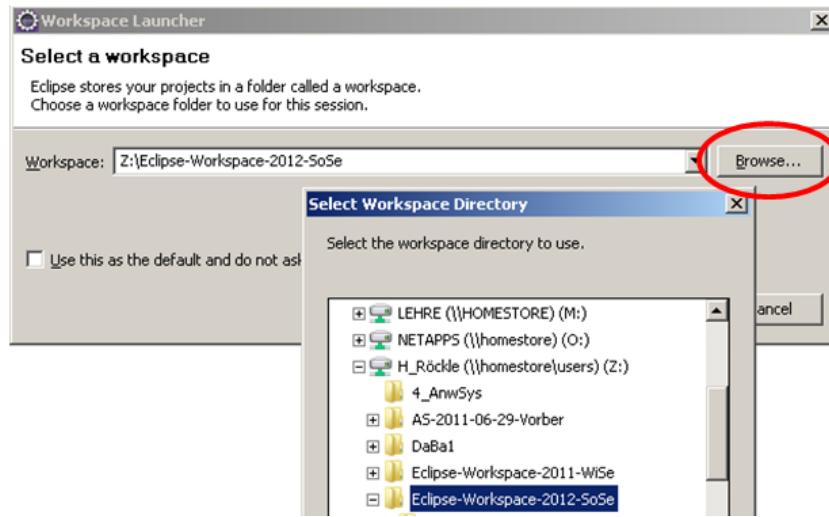
Nachdem der Workspace angelegt wurde, kann er beim Start von Eclipse ausgewählt werden:

3.3.1 Alternative 1

Eclipse schlägt das zuletzt benutzte Workspace-Verzeichnis vor (Screenshot wie in 3.1).

3.3.2 Alternative 2

Falls nicht das richtige Workspace-Verzeichnis vorgeschlagen wird, kann dieses mit „Browse...“ ausgewählt werden.



4 Tomcat Server in der Eclipse-Basiskonfiguration

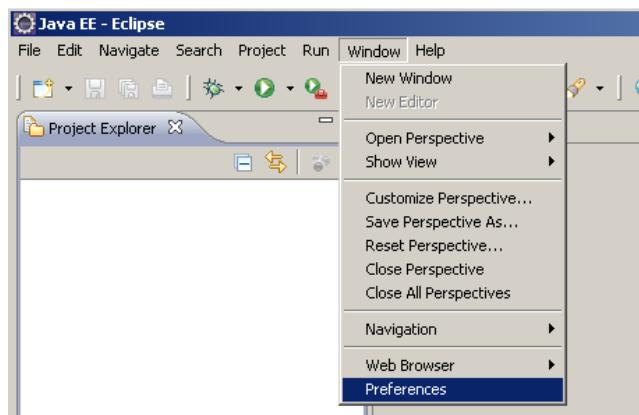
Die Eclipse-Basiskonfiguration wird im Workspace gespeichert und gilt für alle Projekte innerhalb dieses Workspace.

Für unsere Zwecke ist es wichtig, hier den installierten Tomcat-Server anzugeben.

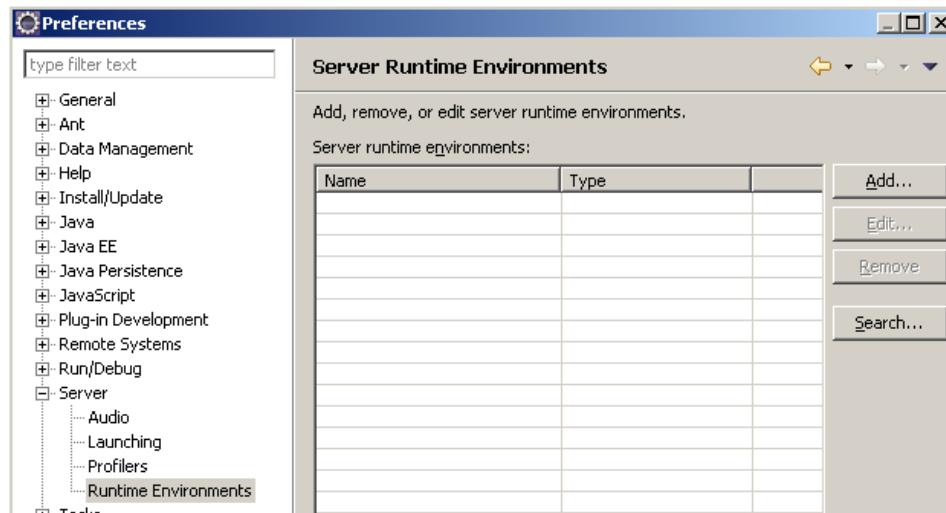
Voraussetzung: Es muss ein Tomcat Server installiert sein, vgl. 1.2 bzw. 1.4.

4.1 Öffnen der Eclipse-Basiskonfiguration

Die Eclipse-Basiskonfiguration ist zu erreichen durch Window → Preferences



Es öffnet sich das folgende Fenster:



Es muss angegeben werden, welche Anwendungsserver auf dem System installiert sind. In unserem Fall vgl. 1.2 bzw. 1.4.

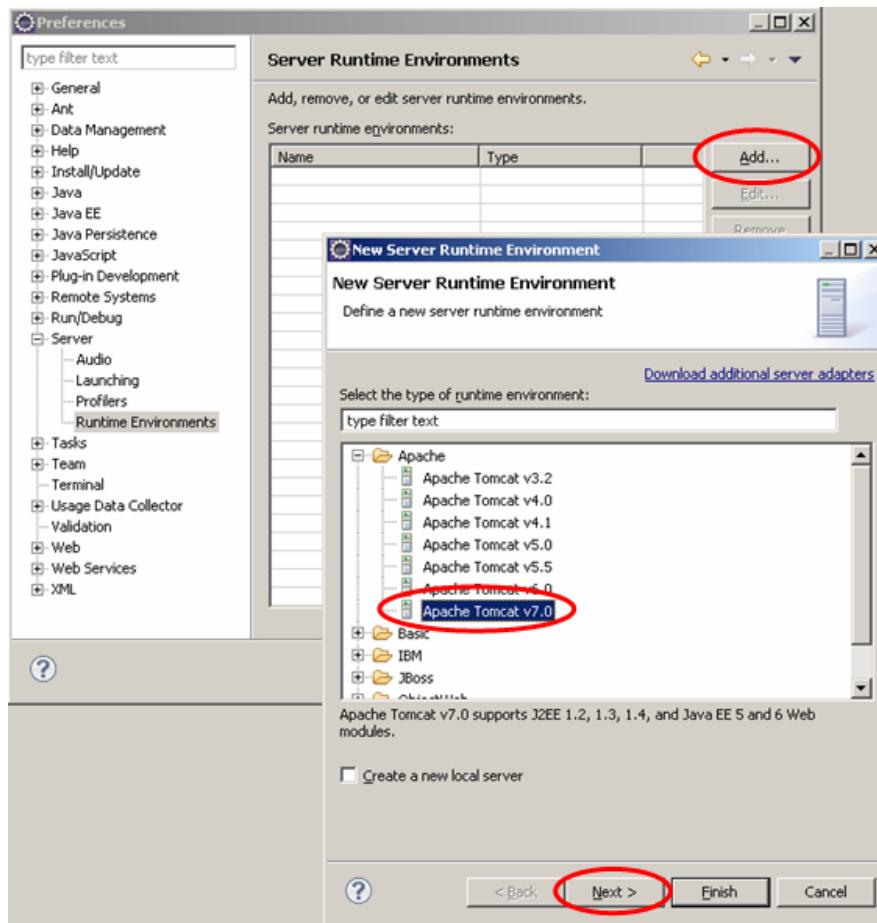
Im Fenster „Preferences“ ist dazu auszuwählen Server → Runtime Environments

Im Fall eines neu angelegten Workspaces, dürfte hier noch kein Server angegeben sein. Es muss deshalb entsprechend 4.2 ein installierter Server angegeben werden.

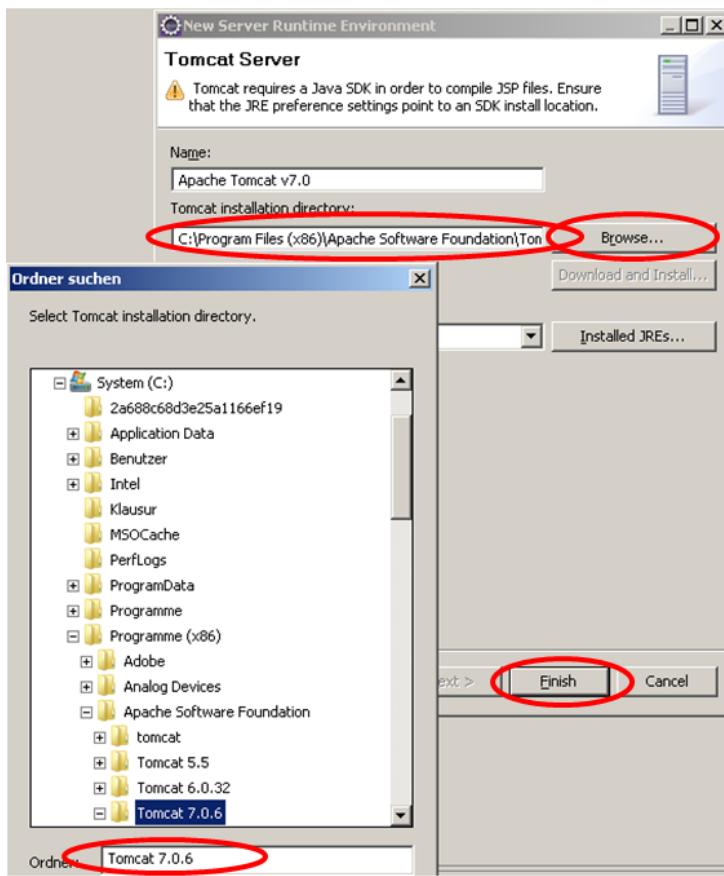
Falls bereits Server angegeben sind, können diese entsprechend 4.3 überprüft werden.

4.2 Installierte Server angeben

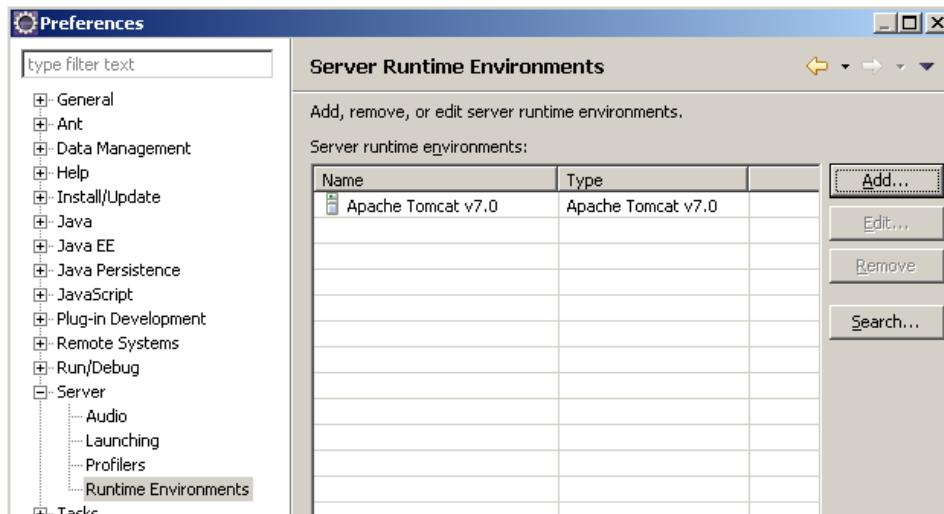
Die installierten Tomcat-Versionen müssen in Eclipse bekannt gemacht werden. Mit Add... öffnet sich das folgende Fenster. Wählen Sie die korrekte Tomcat-Version aus (vgl. 1.2 bzw. 1.4) und gehen Sie weiter mit Next >:



Im nächsten Fenster wird das Tomcat-Programmverzeichnis (vgl. 1.2 bzw. 1.4) angegeben oder mit Browse... ausgewählt:



Nach Bestätigung mit Finish ist unsere Tomcat-Version dem Eclipse-Workspace bekannt:



Durch Wiederholung der angegebenen Schritte können wir auch mehrere – auf unserem System parallel installierte – Tomcat Versionen eingeben. Das empfiehlt sich aber nur, wenn wir diese auch tatsächlich nutzen wollen.

Achtung: Im WiSe 11/12 waren auf den Pool PCs die Tomcat Versionen 7.0.6 und 6.0.32 installiert. Leider führte die Benutzung von Tomcat 7.0.6 zu Problemen. Es musste deshalb an dieser Stelle auch die Tomcat 6.0.32 Version dem Eclipse bekanntgemacht werden.

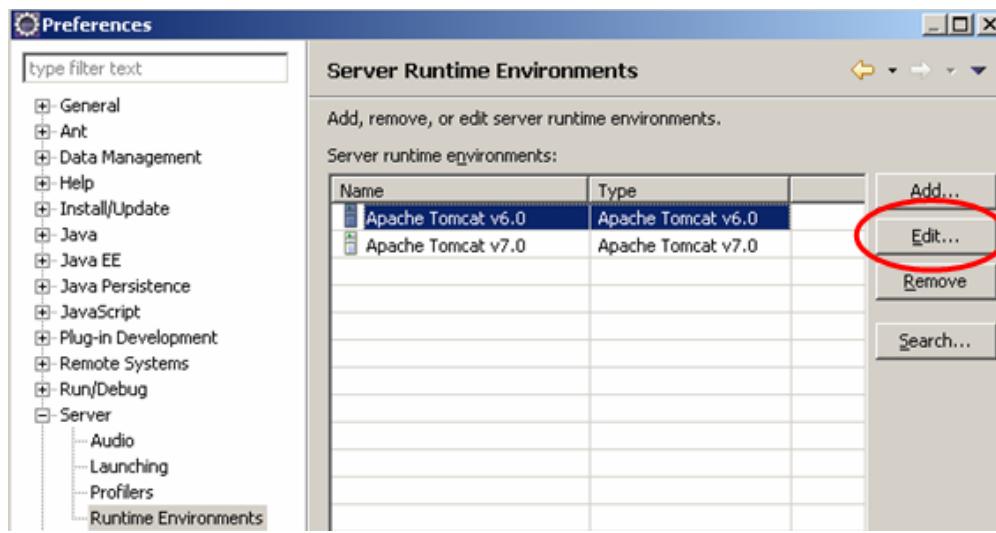
Solange dieser Hinweis hier steht, sollte auch weiterhin Tomcat 6 eingerichtet und verwendet werden.

4.3 Eingetragene Server überprüfen

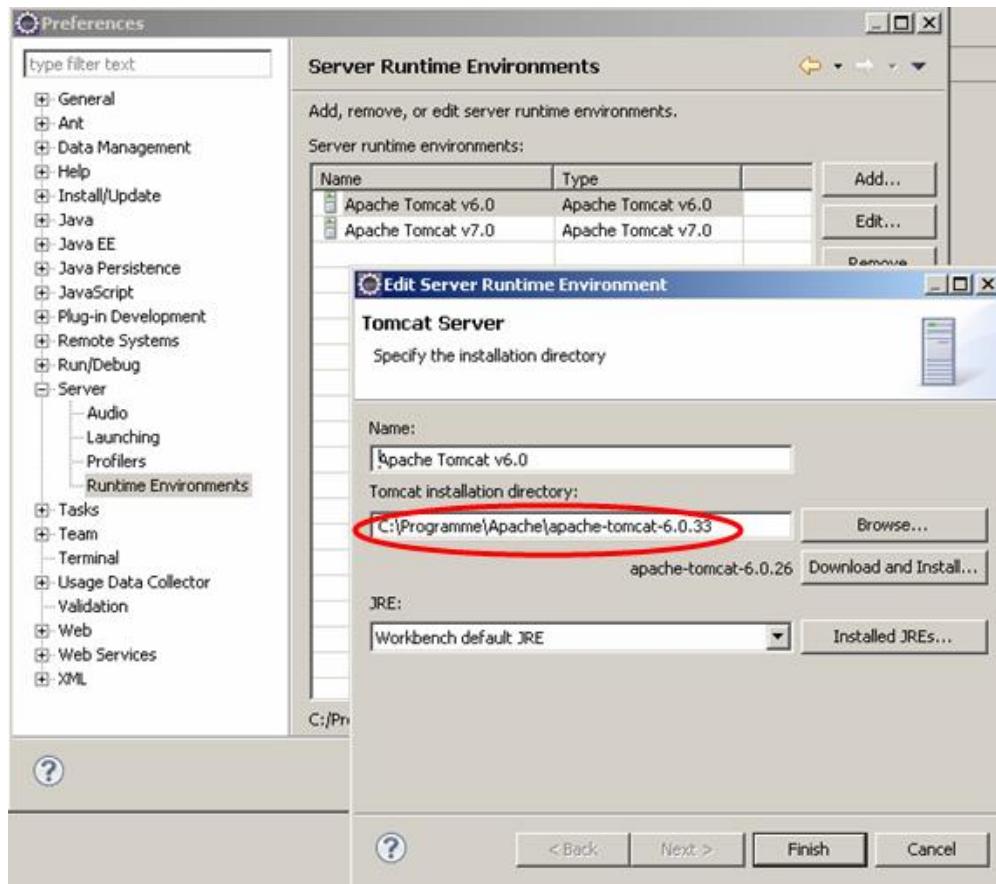
Bei Problemen mit der Serverkonfiguration kann es sinnvoll sein, die angegebenen Server zu überprüfen, ob sie auf dem System installiert sind und ob der angegebene Verzeichnispfad stimmt.

4.3.1 Serverkonfiguration öffnen

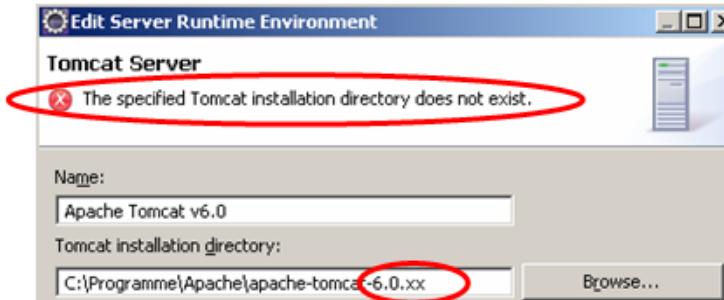
Server markieren und auf Edit... klicken.



Es öffnet sich die Serverkonfiguration, in der Tomcat-Version und –Installationspfad angegeben sind.



Wenn die Angaben fehlerhaft sind, erscheint in diesem Fenster eine Fehlermeldung, andernfalls ist die Einrichtung korrekt.



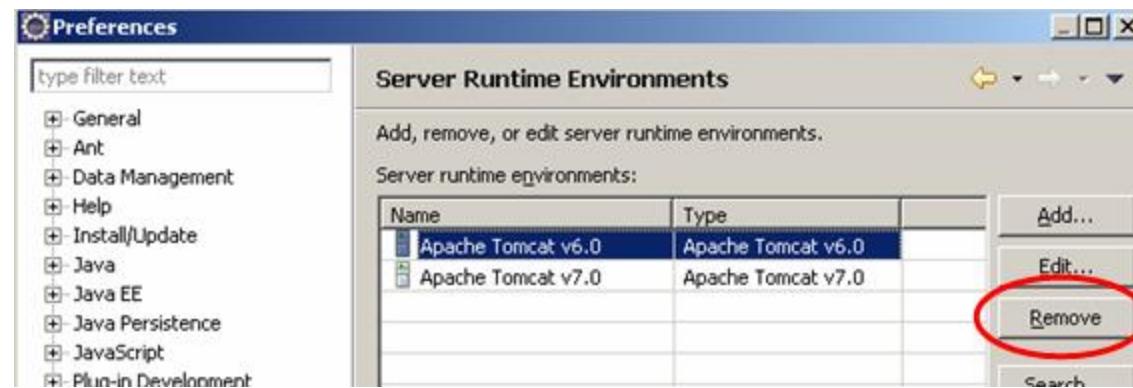
4.3.2 Fehlermöglichkeiten

Es bestehen die folgenden Fehlermöglichkeiten mit den entsprechenden Möglichkeiten zur Fehlerbehebung:

Fehler	Erkennung	Behebung
Angegebene Tomcat-Version ist nicht installiert	System prüfen, vgl. 1.2	Servereintrag entfernen, vgl. 4.3.3
Tomcat-Version ist installiert aber der angegebene Verzeichnispfad ist falsch	Eclipse-Fehlermeldung	Korrekte Pfad eintragen, vgl. 1.2 und 4.3.4

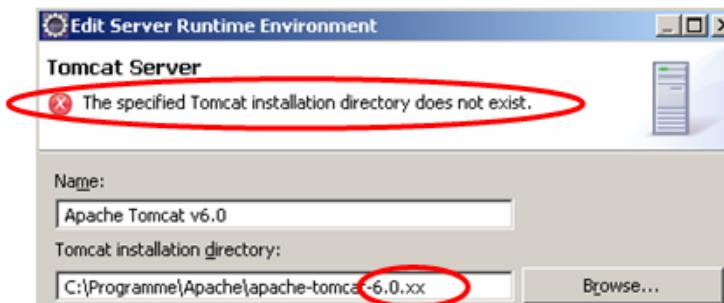
4.3.3 Nicht installierte Versionen entfernen

Falls eine falsche (nicht installierte) Tomcat-Version angegeben ist, kann diese mit „Remove“ gelöscht werden:

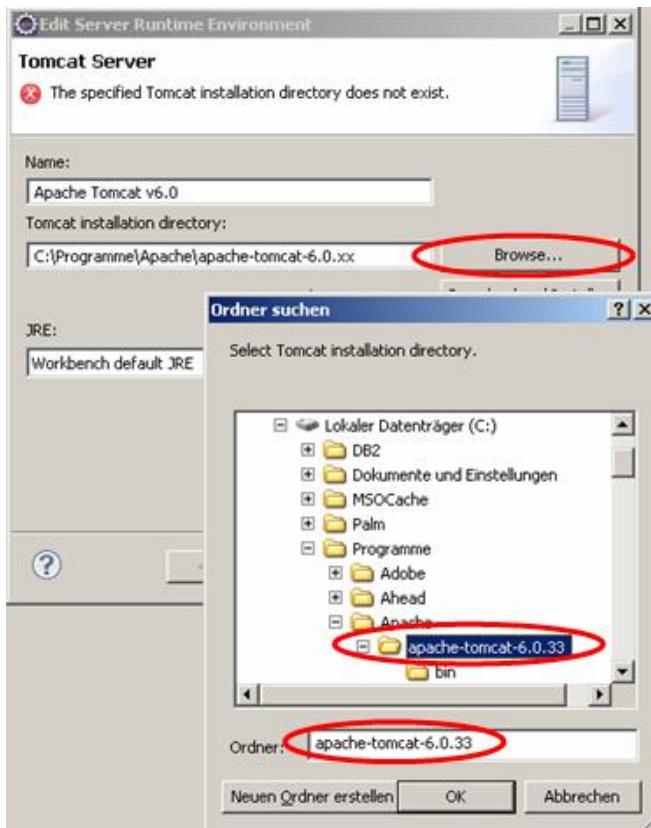


4.3.4 Korrekten Pfad eintragen

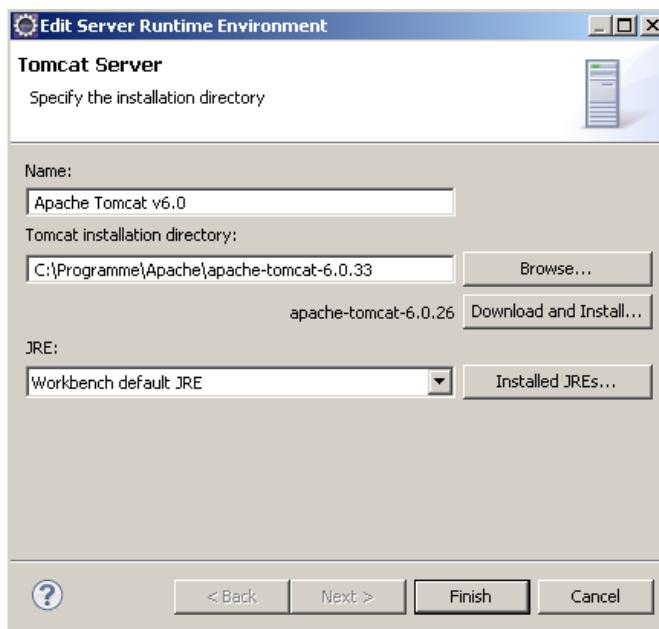
Falls der Pfad falsch war, aber der richtige Pfad bekannt ist (z.B. aus 1.2), dann kann dieser in der bereits in 4.3.1 geöffneten Serverkonfiguration angegeben werden:



Eclipse unterstützt das Suchen des Pfades mit Browse..., wodurch ein Dateienfenster zur Auswahl des korrekten Pfads erscheint:



Wenn die Fehlermeldung verschwindet, ist die Pfadangabe korrekt und kann mit Finish bestätigt werden.



4.3.5 Abschluss der Prüfung

Das Vorgehen 4.3.1 bis 4.3.4 ist für alle angegebenen Server zu wiederholen.

Name	Type	Add...
Apache Tomcat v6.0	Apache Tomcat v6.0	
Apache Tomcat v7.0	Apache Tomcat v7.0	Edit...

5 Eclipse-Projektkonfiguration

Zur Entwicklung von Webanwendungen in Eclipse benutzen wir ein „Dynamic Web Project“ mit dem Namen „O4S“. Der Name ist nicht entscheidend. Falls wir später ein weiteres Projekt benötigen, können wir auch einen anderen Namen verwenden, z.B. „O4Sa“.

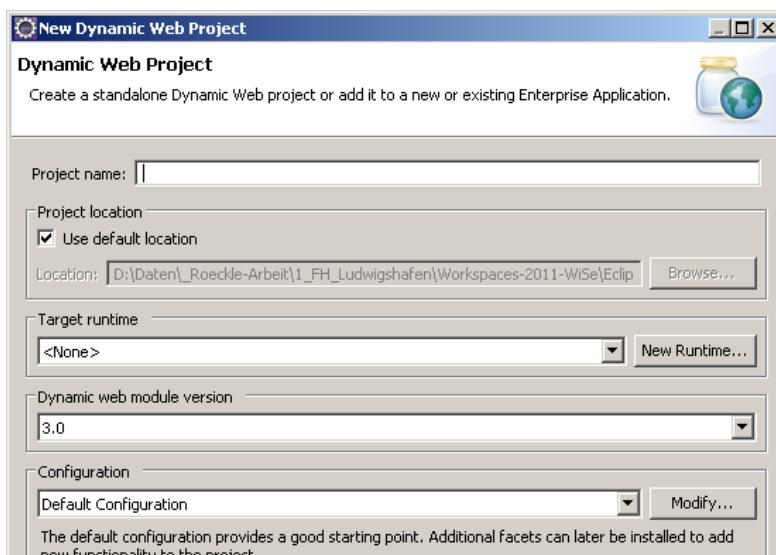
5.1 Projekt anlegen

5.1.1 Öffnen des Screenflows zur Projekteinrichtung

Zum Öffnen des Screenflows zur Projekteinrichtung gibt es verschiedene Möglichkeiten. Am einfachsten klickt man auf das Dreick neben dem Symbol „New“ und in dem sich darauf öffnenden Menü auf „Dynamic Web Project“:



Es öffnet sich das Fenster „New Dynamic Web Project“:

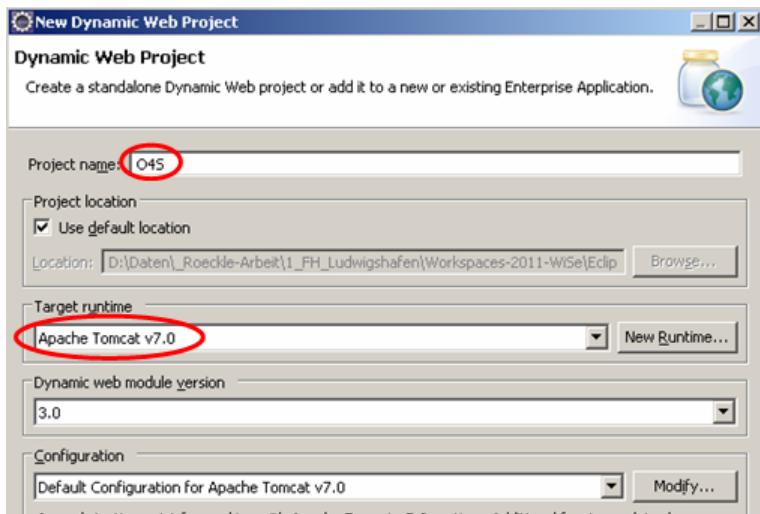


Alternativen zur Öffnung dieses Fensters sind

- Klick auf „New“ , dann im Auswahlfenster das Verzeichnis „Web“ und darin „Dynamic Web Project“.
- Menü: File → New → Dynamic Web Project
- Menü: File → New → Other... → Web → Dynamic Web Project

5.1.2 Angaben zur Projekteinrichtung

Geben Sie hier zunächst den Projektnamen (z.B. O4S) ein.



Achtung: Im WiSe 11/12 führte die Benutzung von Tomcat 7 / Dynamic web module version 3.0 zu Problemen. Es musste deshalb Tomcat 6 / Dynamic web module version 2.5 verwendet werden!

Solange dieser Hinweis hier steht, sollte auch weiterhin Tomcat 6 / Dynamic web module version 2.5 verwendet werden.

Beachten Sie außerdem:

- Als „Target Runtime“ wird die Tomcat Version eingetragen, die Sie unter 4.2 (bzw. 4.3) eingetragen haben.
- Es werden weitere Standard-Angaben (Dynamic web module version, Configuration) verwendet.

Im Normalfall sollte es reichen, jetzt zweimal Next > und einmal Finish zu klicken (vgl. 5.1.4). Ggf. berücksichtigen Sie vorher noch 5.1.3.

5.1.3 Weiteres zur Tomcat Version

Falls Sie in 4.2 (oder 4.3) mehrere Server eingetragen haben, sollten Sie prüfen, ob hier der richtige Server angegeben ist. Falls nicht, klicken Sie auf „New Runtime...“ um dem Projekt einen anderen Tomcat-Server zuzuordnen.

Dieser kann

- bereits eingerichtet sein (vgl. 4.2 oder 4.3) oder
- hier angegeben werden (installation directory)

Beachten Sie, dass der Server mit der angegebenen dynamic web module version zusammenpassen muss. Die folgenden Angaben haben bereits funktioniert:

Tomcat version	Dynamic web module version
Tomcat v6.0.32	2.5
Tomcat v7.0.22	3.0

Die folgenden Angaben haben bereits Fehler produziert:

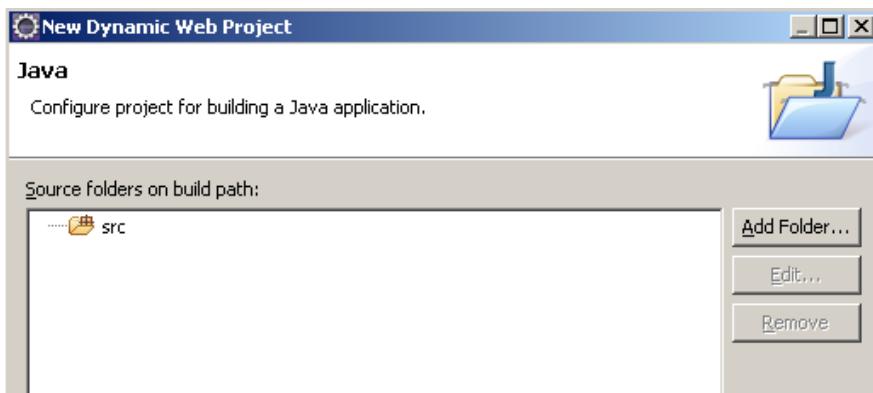
Tomcat version	Dynamic web module version
Tomcat v7.0.6	3.0

Falls hier Fehler entstehen, sind diese in der Regel nur durch Neuanlegen des Projekts zu beheben.

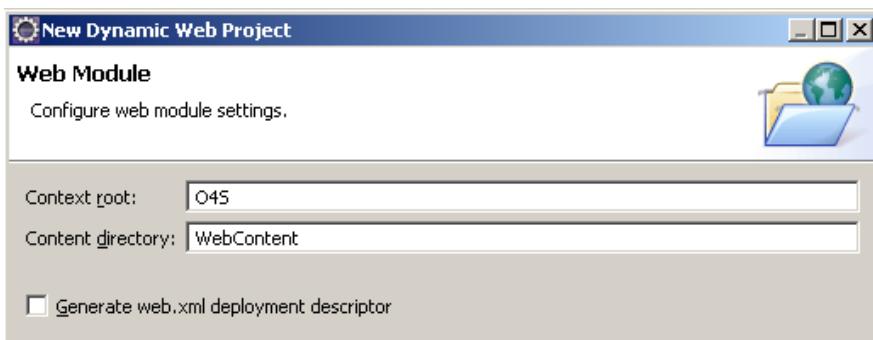
5.1.4 Weitere Schritte

Sie brauchen keine Änderungen an den weiteren Angaben vornehmen, d.h.:

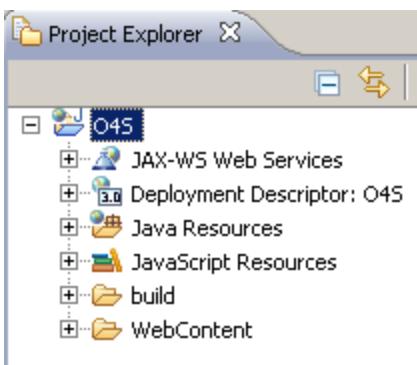
Bestätigen Sie die „Source folder“-Angaben mit Next >



Bestätigen Sie die „Web Module“-Angaben mit Finish



Anschließend sehen Sie Ihr Dynamic Web Project im Eclipse Project Explorer:



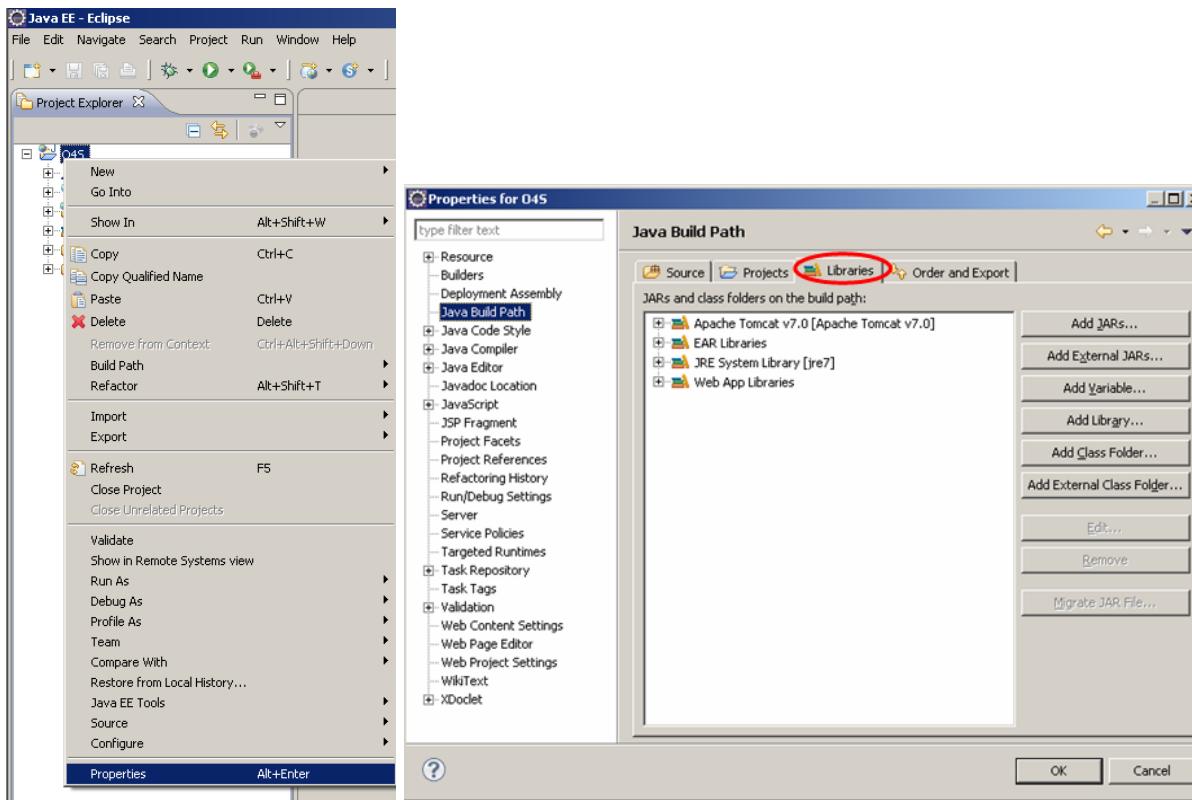
Falls es bei Ihnen anders aussieht, könnte es daran liegen, dass Sie anstatt eines „Dynamic Web Project“ ein „Java Project“ erzeugt haben. In dem Fall löschen Sie Ihr Projekt incl. der angelegten Daten und beginnen Sie 5.1 von Neuem.

5.2 Bibliotheken im Java-Kontext einbinden

Zur Nutzung von Java für Webanwendungen benötigen wir verschiedene Bibliotheken. Um diese einzubinden, konfigurieren wir die „Projekteigenschaften“, speziell den „Build Path“.

5.2.1 Fenster zur Pflege der Bibliotheken

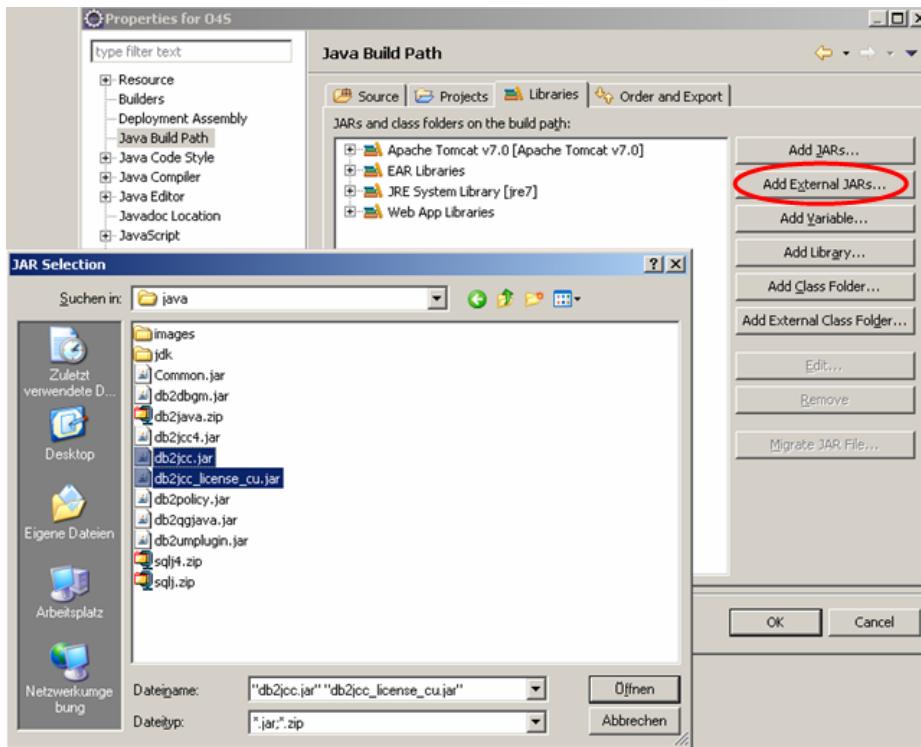
Klicken Sie mit der rechten Maustaste auf den Projektnamen. Es öffnet sich ein Kontextmenü. Klicken Sie auf „Properties“ (ganz unten), im Eigenschaftsfenster links auf „Java Build Path“ und dort auf den Reiter „Libraries“:



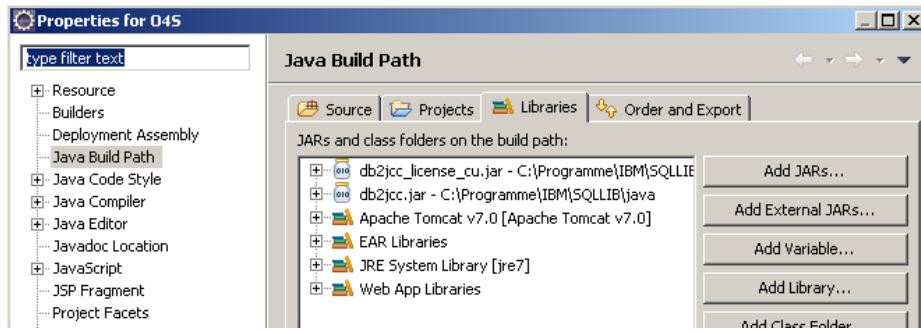
Alternativ kann das Fenster vom Kontextmenü aus erreicht werden mit Build Path → Configure Build Path... → Libraries.

5.2.2 Datenbanktreiber einbinden

Zur Anbindung einer Datenbank an Eclipse müssen Sie Datenbanktreiber einbinden. Verzeichnis und Dateinamen für DB2 sind unter 1.3 angegeben. Zum Einbinden klicken Sie im unter 5.2.1 dargestellten Fenster (Java Build Path) auf „Add External JARs...“ und wählen Sie die genannten JAR-Files aus. Sie können mehrere JAR-Files auf einmal anwählen:



Im Erfolgsfall werden die eingebundenen Pfade dann im Java Build Path Fenster angezeigt:



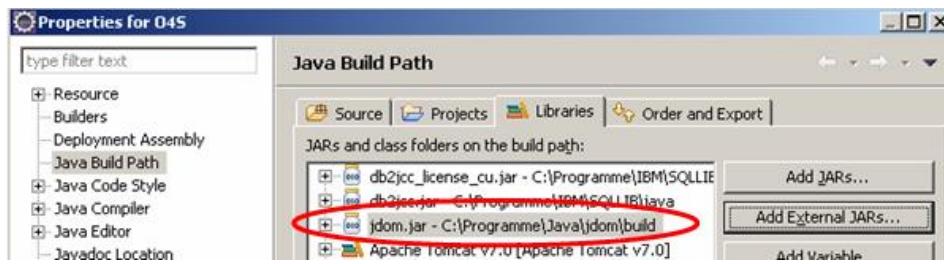
Bestätigen Sie dies noch mit „OK“.

Wenn Sie eine andere Datenbank einbinden möchten oder Ihre DB2-Datenbank anders installiert haben, können sich die Pfade unterscheiden (vgl. 1.4).

5.2.3 XML-Treiber einbinden

Analog zu 5.2.2 binden Sie die JDOM-Bibliothek in Ihr Projekt ein:

- Verzeichnis und JAR-File vgl. **Fehler! Verweisquelle konnte nicht gefunden werden.**
- Öffnen des Java Build Path
- Add External JARs...
- Auswahl der Bibliothek
- Bestätigen mit OK

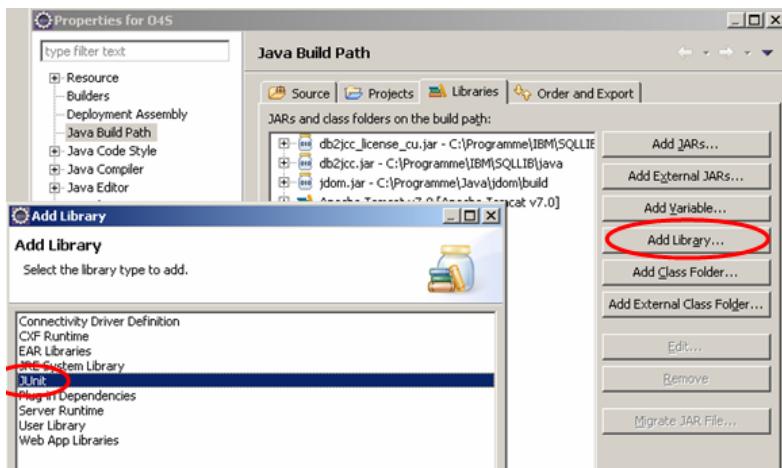


Falls die Xerces-Library benötigt wird, wird diese mit den Angaben aus **Fehler! Verweisquelle konnte nicht gefunden werden.** ebenso eingebunden.

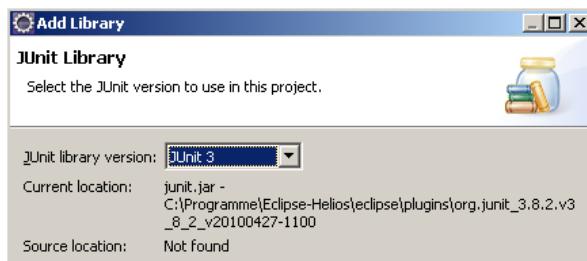
5.2.4 JUnit einbinden

Es wird eine JUnit-Bibliothek benötigt, die vom Fenster Java Build Path / Libraries aus eingebunden werden kann durch

- Klick auf „Add Library...“
- Auswahl von JUnit und Next >



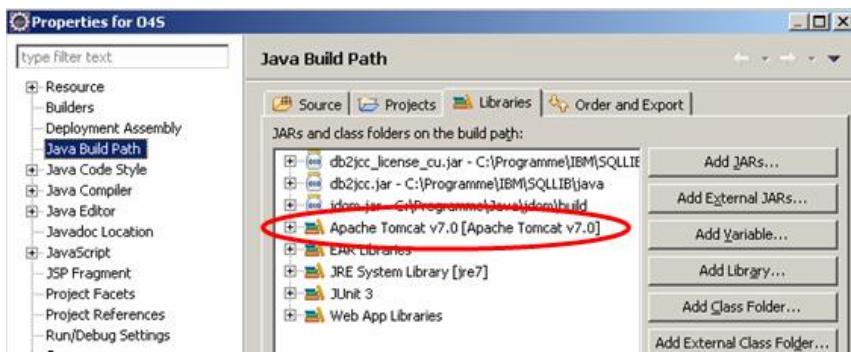
- Auswahl einer JUnit-Version, in unserem Fall JUnit 3 und Finish



5.2.5 Serverbibliothek prüfen

Die Java-Bibliothek des Tomcat Servers muss eingebunden werden, um während der Java-Entwicklung die Code-Vervollständigung angeboten zu bekommen. Je nach Eclipse-Version erfolgt dieser Schritt automatisch oder muss manuell vorgenommen werden.

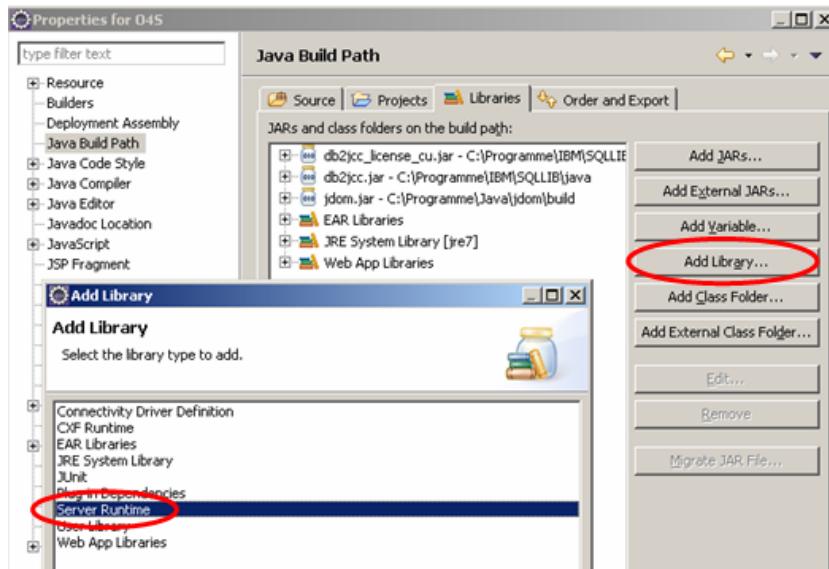
Wenn die Server-Bibliothek im Java Build Path / Libraries Fenster angezeigt wird, ist diese bereits eingebunden und es ist hier nichts mehr zu tun:



5.2.6 Serverbibliothek einbinden

Nur wenn die Server-Bibliothek noch nicht eingebunden ist (vgl. 5.2.5), muss dies ausgehend vom Fenster Java Build Path / Libraries aus getan werden:

- Klick auf „Add Library...“
- Auswahl von „Server Runtime“ und Next >

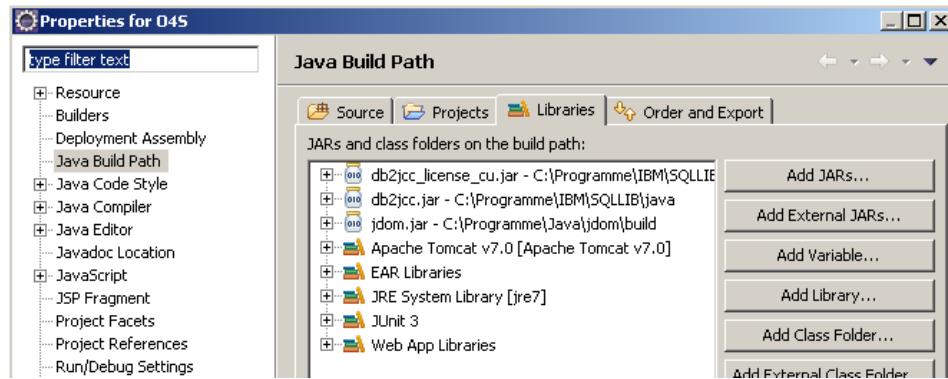


Es öffnet sich ein Fenster, in dem einer der unter 4.2 bzw. 4.3 eingerichteten Server ausgewählt werden kann. Evtl. wird nur ein Server angezeigt. Wählen Sie diesen aus und klicken Sie Finish.



5.2.7 Ergebnis

Es ergibt sich die folgende Build Path Konfiguration:



5.3 Bibliotheken im Web-Kontext einbinden

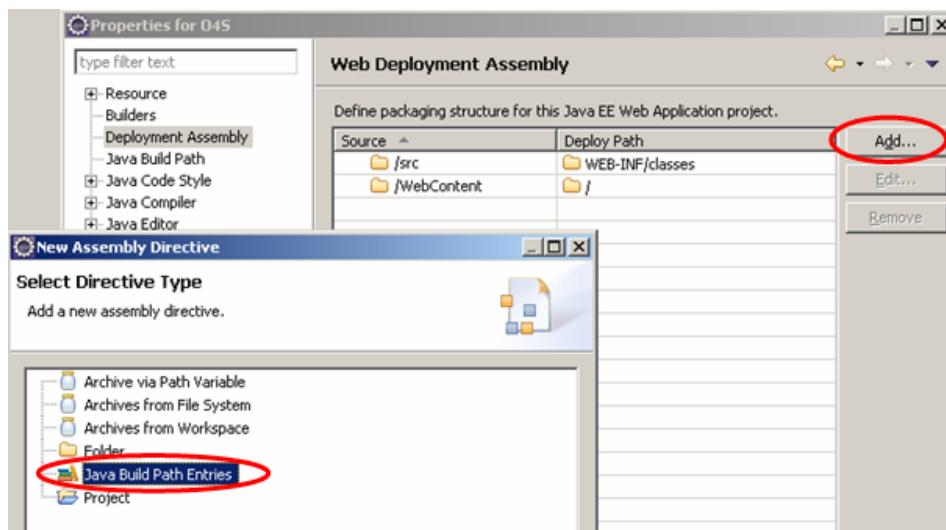
Die unter 5.2 eingebundenen Bibliotheken stehen nun für die Java-Entwicklung in unserem Dynamic Web Project zur Verfügung, allerdings noch nicht für den Aufruf aus Webseiten heraus.

Um auch von einer Webseite aus auf die eingebundenen Funktionalitäten zugreifen zu können, müssen die Bibliotheken auch in den Web-Kontext übernommen werden.

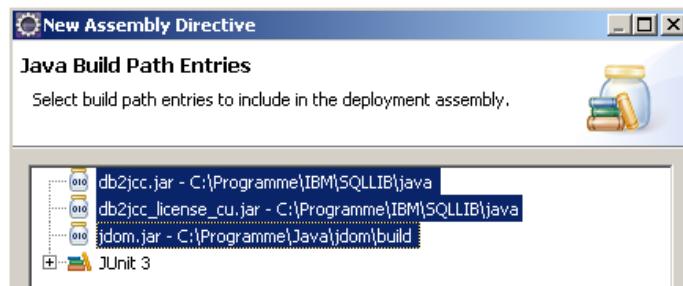
Klicken Sie wie in 5.2.1 mit der rechten Maustaste auf den Projektnamen und im Kontextmenü auf „Properties“. Im Eigenschaftsfenster wählen Sie dieses Mal links „Deployment Assembly“.

Anmerkung: In früheren Eclipse-Versionen hieß es anders, z.B. „Java EE Module Dependencies“ oder „J2EE Module Dependencies“.

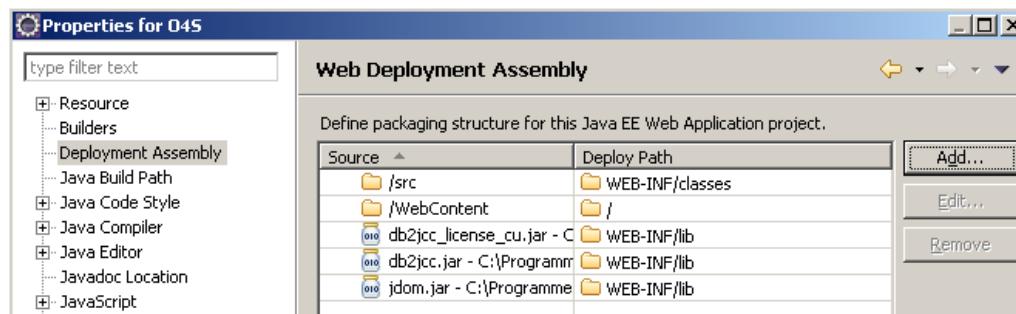
Klicken Sie im Fenster auf Add... und wählen Sie „Java Build Path Entries“ und klicken Sie Next >



Im öffnenden Fenster wählen Sie alle angegebenen JAR-Files aus (JUnit ist nicht nötig) und klicken Finish.



Im Ergebnis sieht die Web Deployment Assembly folgendermaßen aus:



6 Daten in den Workspace kopieren

Hier wird knapp beschrieben, wie Sie Daten der Veranstaltung „Anwendungssysteme“ in Ihren Workspace einspielen (kopieren) können.

6.1 Java-Packages und Klassen

Java-Klassen liegen in Package-Strukturen auf der M-Platte. Sie können diese entweder über Eclipse oder über den Explorer kopieren.

6.1.1 Kopieren in Eclipse

- Packages manuell anlegen
- Daten mit Drag & Drop in die Packages ziehen
- Am Schluss idealerweise den Workspace refreshen:
 - Kontextmenü → Refresh oder
 - Auf den Projektnamen klicken und F5 drücken.

6.1.2 Kopieren im Windows Explorer

- Herausfinden, wo die Daten im Workspace hingehören
- Vollständige Dateistrukturen an die richtige Stelle des Workspaces kopieren, dadurch entstehen implizit auch die Packagestrukturen im Eclipse
- Workspace unbedingt refreshen
 - Kontextmenü → Refresh oder
 - Auf den Projektnamen klicken und F5 drücken.

6.2 WebContent

Auch die WebContent Daten können im Eclipse oder im Explorer kopiert werden, es gibt aber keinen nennenswerten Unterschied.

Alle Dateien und Unterordner des Verzeichnisses WebContent von der M-Platte werden in das Verzeichnis WebContent im Workspace kopiert. Verzeichnisstrukturen sollen erhalten bleiben.

Am Ende den Workspace refreshen.

6.3 Zugangsdaten für Datenbank

In der Java-Klasse DB2Access müssen die korrekten Datenbankzugangsdaten eingetragen werden.
Dies sind die Werte aus 1.3 für die folgenden Variablen:

Variable	Beispielwert
dbUserId	z.B. USER34
dbPassword	db2users
dbURL	jdbc:db2://143.93.202.3:50000/OUEBDB
dbSchema	z.B. S654321
dbDriver	com.ibm.db2.jcc.DB2Driver

Bei einer Installation auf dem eigenen PC / Laptop sind die Daten entsprechend 1.4 anzupassen.

6.4 Ergebnis

Wenn Sie alle Daten richtig kopiert haben und alle Bibliotheken wie in 5.2 und 5.3 eingebunden haben, dann müsste alles so funktionieren wie auf dem Dozenten-PC.

7 Datenbankzugriff mit Eclipse

Die Datenbankzugriffskomponente von Eclipse erlaubt

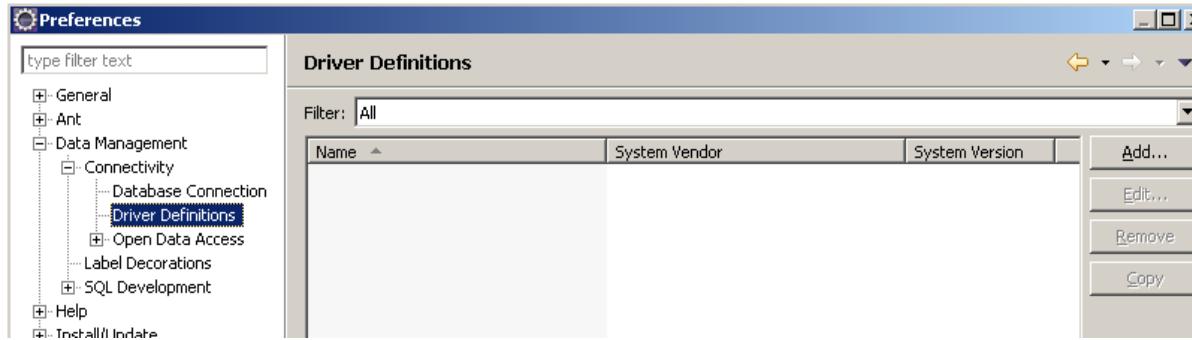
- die Speicherung von SQL-Kommandos und Stored Procedures in „SQL-Files“,
- die Ausführung der gespeicherten SQL-Kommandos und Stored Procedures

Voraussetzung dafür ist die

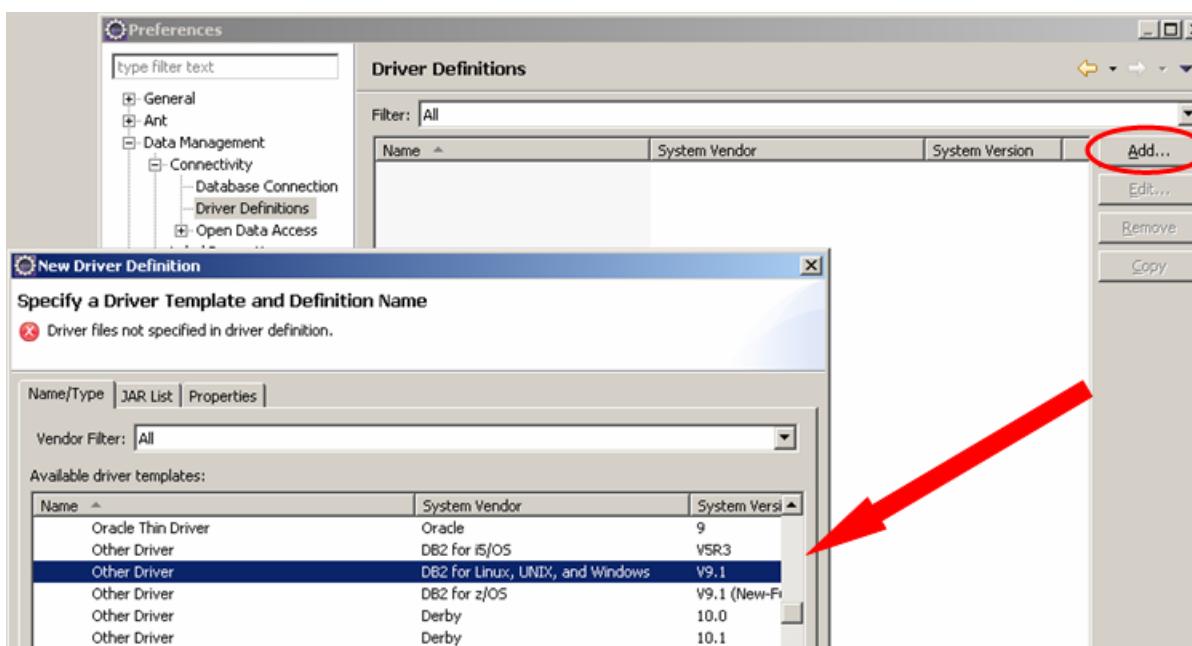
- Einrichtung einer „Driver Definition“ mit Angabe eines geeigneten Datenbanktreibers in Form von JAR-Files sowie Datenbankzugangsdaten (vgl. 1.3 bzw. 1.4) wie in 5.2.2 bzw. 6.3,
- Einrichtung einer „Database Connection“.

7.1 „Driver Definition“ in der Eclipse-Basiskonfiguration

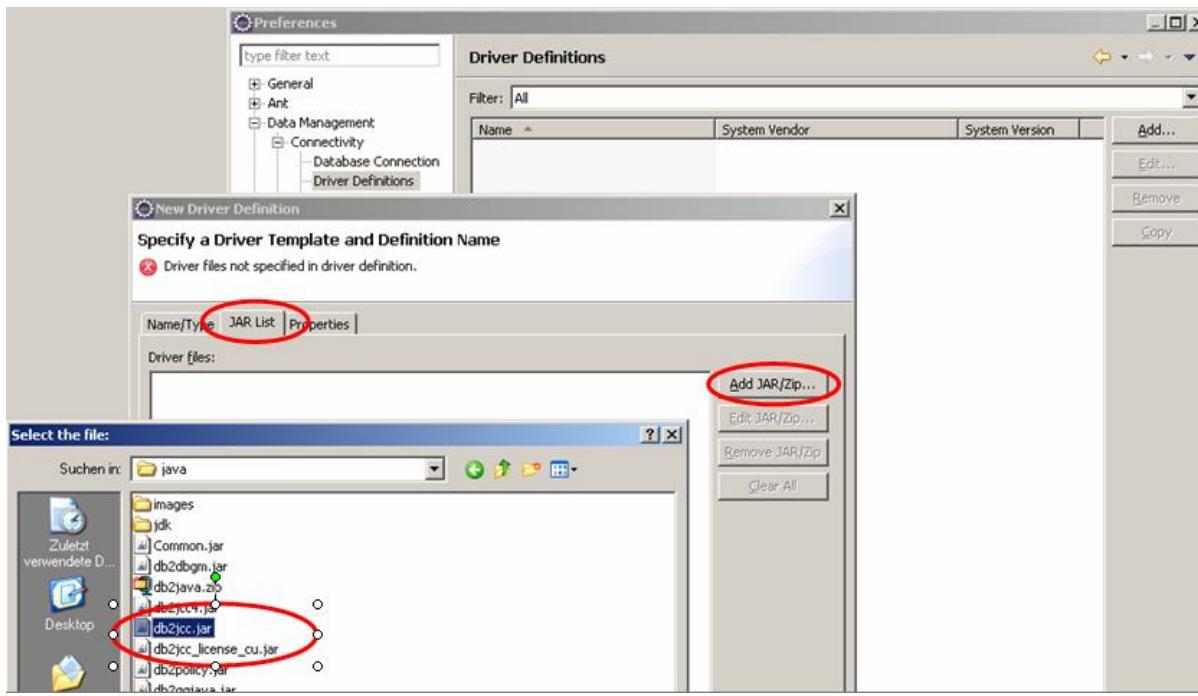
Wie in 4.1 angegeben ist die Eclipse-Basiskonfiguration über das Hauptmenü von Eclipse zu erreichen mit Windows → Preferences. Im dann erscheinenden Fenster „Preferences“ ist links auszuwählen „Data Management → Connectivity → Driver Definitions“:



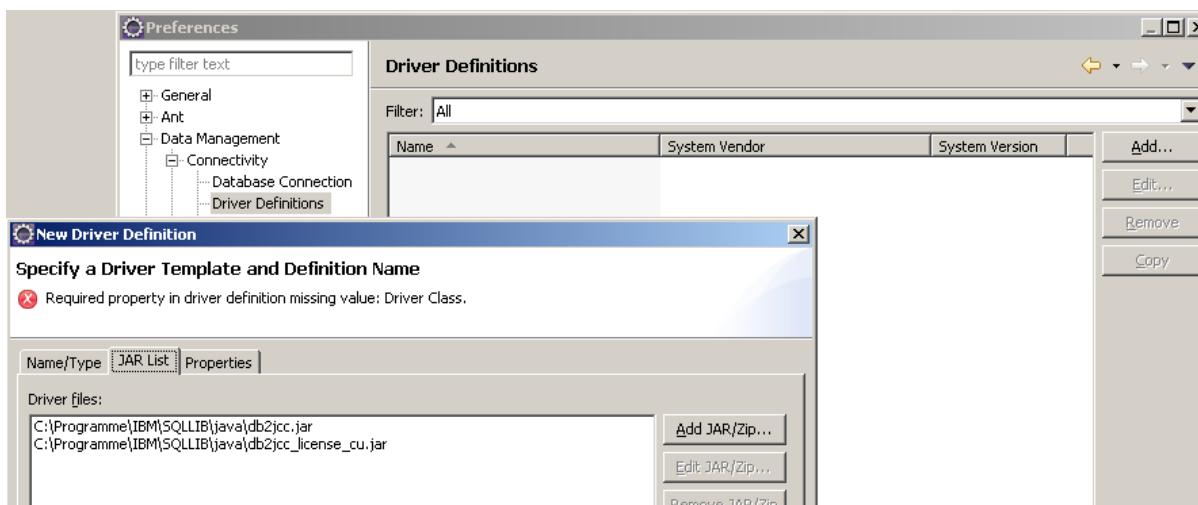
Nach Klick auf „Add...“ ist in der Liste der „driver templates“ auszuwählen „Other Driver, DB2 for Linux, UNIX, and Windows“. Es erscheint eine Fehlermeldung, weil noch keine Datenbanktreiber (JAR-Files) angegeben wurden.



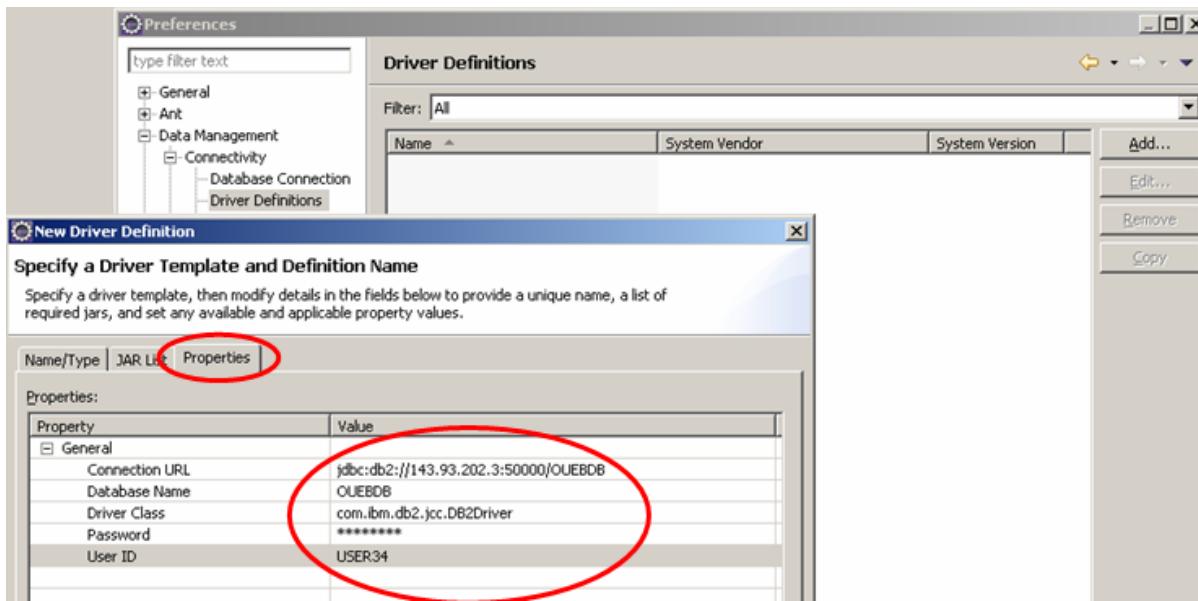
Anschließend ist der Reiter „JAR List“ auszuwählen. Der Inhalt des Fensters ist noch leer, deshalb ist auf „Add JAR/Zip...“ zu klicken. Im erscheinenden Dateiauswahl dialog sind die JAR-Files auszuwählen, die in 1.3 (bzw. 1.4) angegeben und bereits in 5.2.2 genutzt wurden:



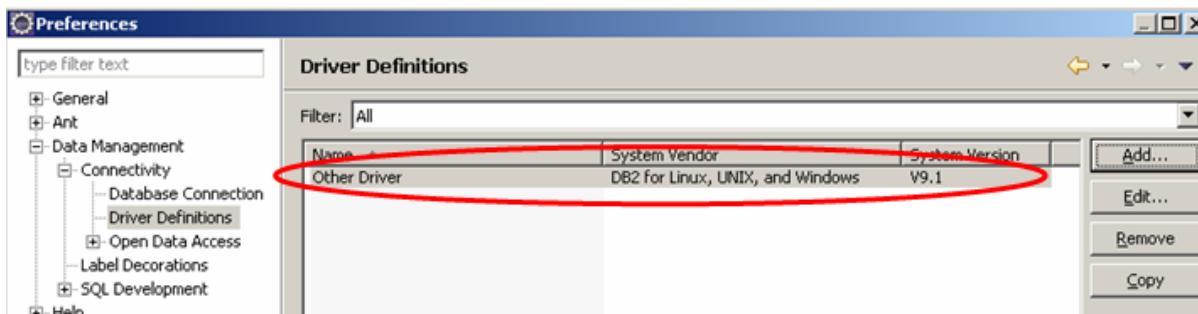
Da im Dateiauswahldialog nur eine Datei ausgewählt werden kann, muss dieser Schritt evtl. mehrmals (z.B. für DB2 zweimal) ausgeführt werden, bis die benötigten JAR-Files eingebunden sind. Dennoch erscheint noch eine Fehlermeldung, weil die zentrale Java-Klasse für die Datenbankverbindung noch nicht angegeben wurde.



Anschließend ist der Reiter „Properties“ auszuwählen und es sind die Datenbank-Zugangsdaten aus 1.3 bzw. 1.4 einzutragen:



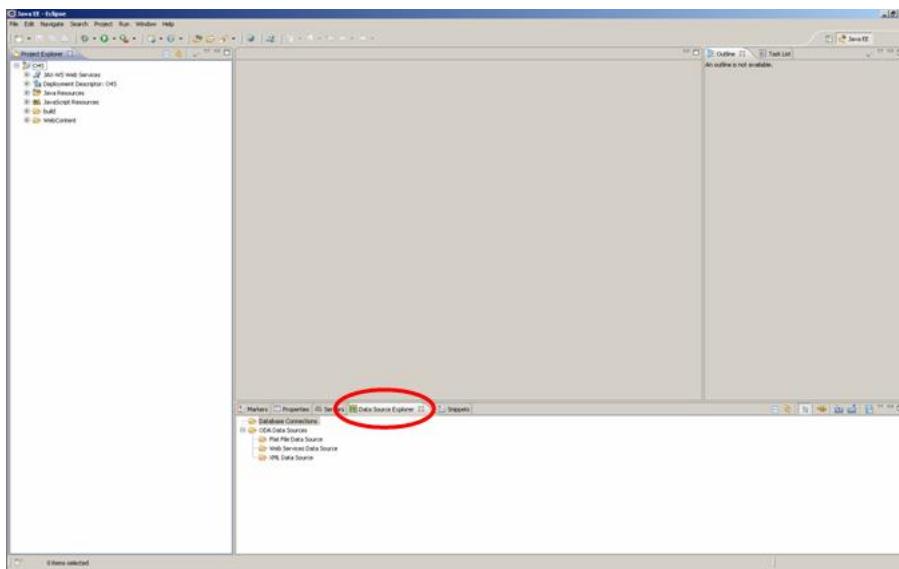
Im Ergebnis erhalten Sie eine fertig eingerichtete „Driver Definition“:



7.2 Erzeugen einer „Database Connection“ im Data Source Explorer

7.2.1 Finden / Öffnen des Data Source Explorer

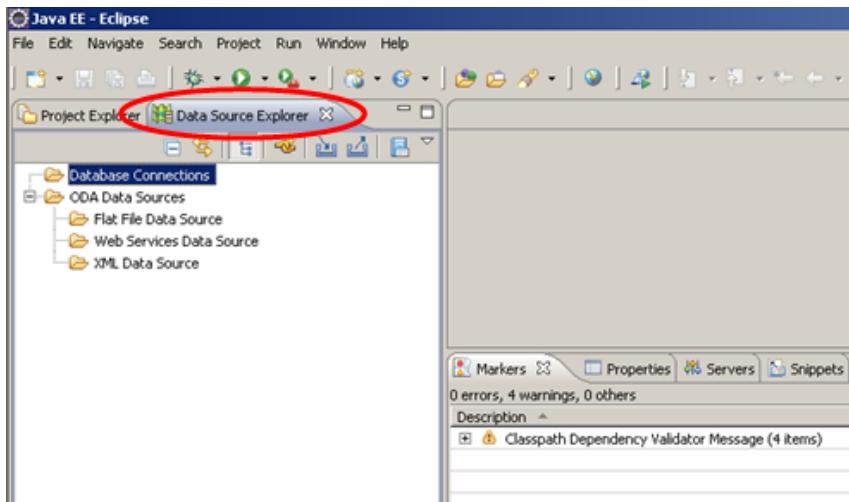
In der Standarddarstellung der Java EE Perspective eines Dynamic Web Project wird das Fenster „Data Source Explorer“ im unteren Bereich von Eclipse angezeigt:



Falls kein Data Source Explorer angezeigt wird, können Sie die Anzeige öffnen, indem Sie

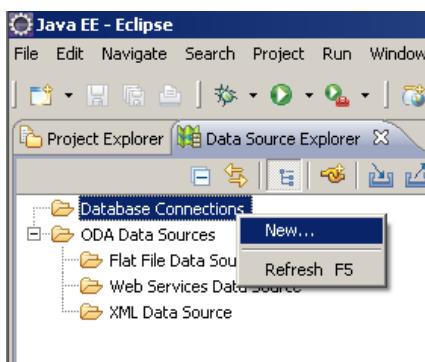
- entweder die Java EE Ansicht auf Ihre Standardeinstellung zurücksetzen:
- oder den Data Source Explorer explizit über das Eclipse-Menü öffnen mit
 - Window → Show View... → Data Source Explorer
 - oder Window → Show View... → Other... → Data Management → Data Source Explorer

Anschließend ist es empfehlenswert (für mehr Übersicht), den Data Source Explorer mit der Maus (Drag & Drop) nach links oben neben den Project Explorer zu ziehen:

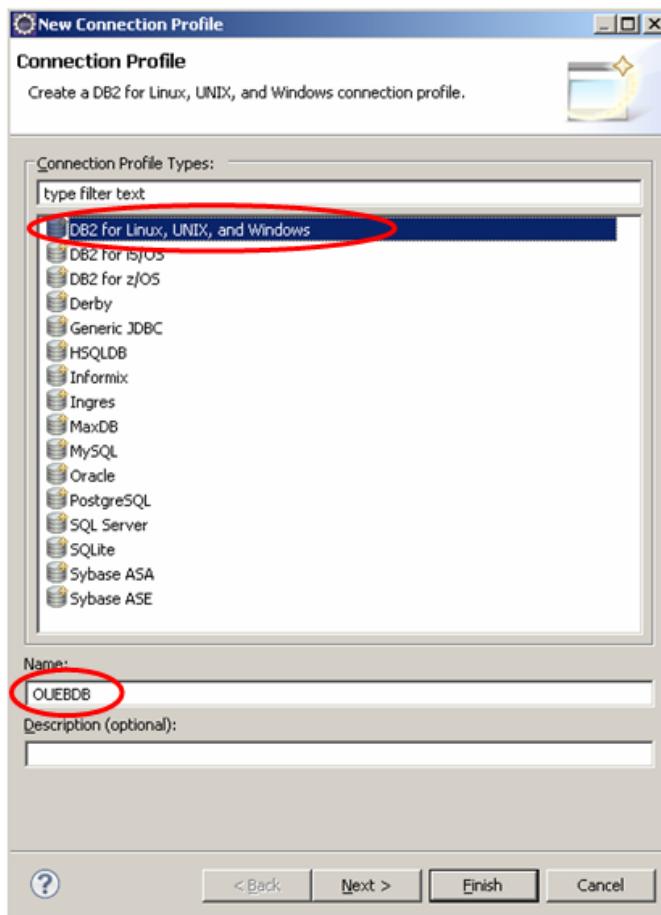


7.2.2 Anlegen einer Database Connection

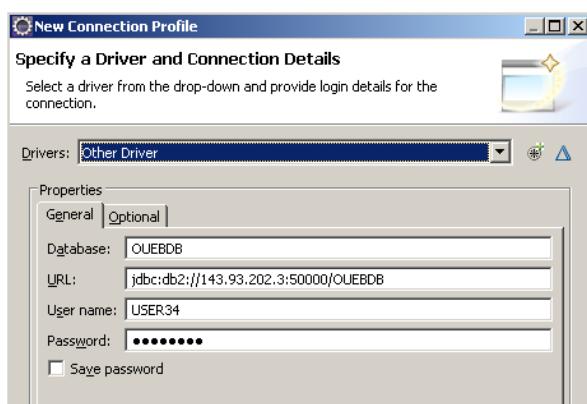
Klicken Sie im Data Source Explorer mit der rechten Maustaste auf den Eintrag Database Connections und im Kontextmenü auf New...



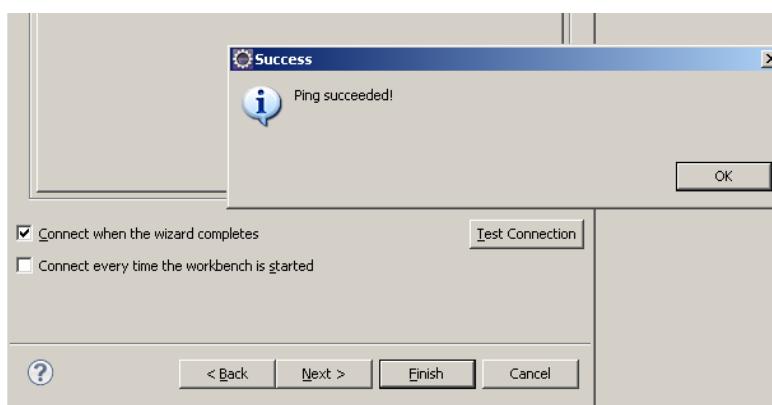
Im Fenster „New Connection Profile“ wählen Sie „DB2 for Linux, UNIX, and Windows“ und geben Sie als Name den Datenbanknamen ein (vgl. 1.3 bzw. 1.4) und klicken Sie „Next >“



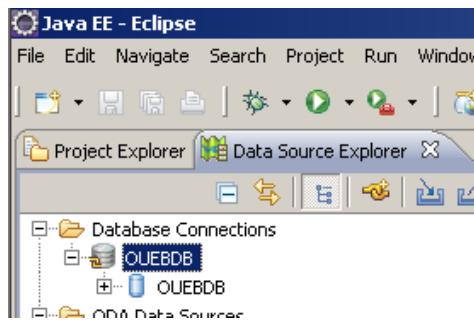
Im nächsten Fenster wählen Sie oben Ihre unter 7.1 angelegte Driver Definition aus. Die dort angegebenen Zugangsdaten werden automatisch darunter eingetragen.



Wenn die Datenbank aktiv und ansprechbar ist und die Zugangsdaten stimmen, müsste der auf dieser Seite mögliche Verbindungstest jetzt erfolgreich sein:



Beenden Sie den Screenflow mit Finish, damit die Database Connection angelegt wird:



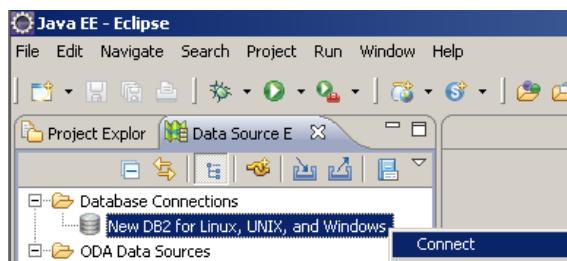
7.3 Anmelden an der Datenbank

Nach dem Anlegen der Database Connection ist der Eclipse Client standardmäßig an der Datenbank angemeldet, erkennbar an dem kleinen gelben Symbol wie im letzten Screenshot in 7.2.2.

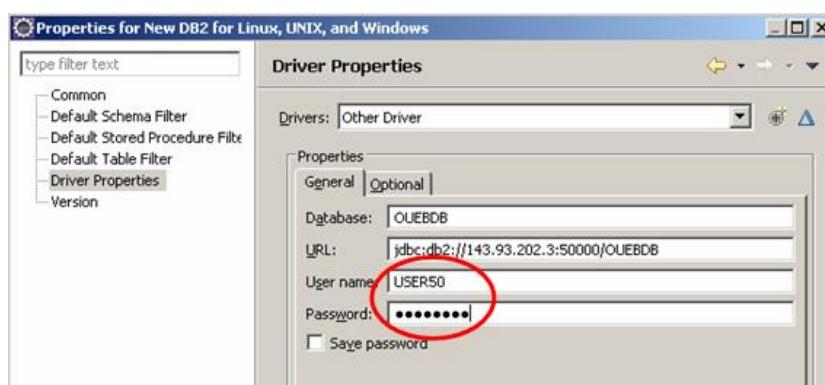
Bei einem späteren Start von Eclipse oder nach zwischenzeitlicher Abmeldung ist der Client nicht mehr angemeldet, erkennbar am Fehlen des kleinen gelben Symbols:



In dem Fall können Sie sich anmelden mit rechtem Mausklick auf die Database Connection und Auswahl von „Connect“:



Mit anschließende Eingabe bzw. Bestätigung der Zugangsdaten:



7.4 Arbeiten mit der Datenbank

Die Arbeit mit dem Eclipse-Datenbank-Client ist nicht Inhalt dieser Anleitung.

Veranstaltung „Web-Anwendungen“

HS Ludwigshafen, Prof. Dr. H. Röckle, Teil 1: HTML und CSS

1	Einleitung	2
1.1	Einordnung.....	2
1.2	Lernziele.....	2
1.3	Literatur.....	2
1.4	Inhalt	2
1.5	Zum „Praktikum Anwendungssysteme“	3
1.6	Benötigte Software.....	3
1.7	Eclipse – Getting Started.....	4
2	HTML.....	5
2.1	Einführung	5
2.2	Einige Details und Hintergründe.....	8
2.3	Einige Tags	11
2.4	HTML-Formulare	16
3	CSS.....	25
4	JSP.....	25
5	JavaScript	25
5.1	Einbinden von JavaScript in HTML	25
5.2	JavaScript als Programmiersprache	28
5.3	„Micky Maus“ – Beispiel.....	31
5.4	Anwendungen auf RegView.jsp.....	40
5.5	Ausblick	48

1 Einleitung

1.1 Einordnung

Im Curriculum des Bachelors Wirtschaftsinformatik an der Hochschule Ludwigshafen stellt die Veranstaltung „Web-Anwendungen“ die Fortsetzung des Zyklus Programmierung / Anwendungssysteme dar. Nachdem die Studierenden in den vorausgegangenen Semestern gelernt haben, wie Java Konsolprogramme entwickelt werden und wie auf SQL-Datenbanken zugegriffen werden kann, wird in dieser Veranstaltung behandelt,

- wie grafische Oberflächen mit Web-Techniken entwickelt werden,
- wie dynamische Web-Anwendungen entwickelt werden, indem Java-Code (Java Server Pages) mit grafischen Web-Oberflächen verbunden werden,
- wie Client-seitige und Server-seitige Web-Techniken zusammenspielen.

Die Entwicklung von Web-Anwendungen folgt dabei dem MVC-Pattern, um gut strukturierte Systemarchitekturen zu erhalten.

Im Folgesemester wird dies abgerundet durch die eigenständige Entwicklung umfangreicherer Web-Anwendung durch Studierendenteams sowie der Vermittlung der wichtigsten Elemente der Web-Applikationssicherheit.

1.2 Lernziele

Nach dem erfolgreichen Besuch der Veranstaltung sollen die Studierenden

- einfache Client- und Server-seitige Techniken und Verfahren der Web-basierten Anwendungsentwicklung
 - kennen und
 - in einfachen Fällen selbstständig anwenden können
- Web-basierte Anwendungsarchitekturen, speziell basierend auf MVC,
 - kennen und
 - bei der Entwicklung von Webanwendungen nutzen können
- Datenbanken in betriebswirtschaftliche Anwendungssysteme einbinden können

1.3 Literatur

Zur Entwicklung von Web-Anwendungen gibt es eine ausgesprochen große Zahl von Onlinequellen, die häufig eine hohe Qualität besitzt und oft sogar kostenfrei genutzt werden kann. Einige dieser Quellen sind

- de.selfhtml.org
- java.sun.com/apis

Weitere Quellen werden in den entsprechenden Detailkapiteln aufgeführt.

Vor diesem Hintergrund fällt die folgende Sammlung von Buchquellen eher knapp aus:

- Mark Lubkowitz: Webseiten programmieren und gestalten, Galileo Press, Bonn, 3. Aufl. (2007)
- Thomas Stark: Java EE 5: Einstieg für Anspruchsvolle, Addison Wesley (2006)
- JavaScript – Eine Einführung, Herdt-Verlag / RRZN
- Java und XML, Herdt-Verlag / RRZN

1.4 Inhalt

Das Thema Web-Anwendungen könnte einen ganzen Studiengang füllen. Im unter 1.1 genannten Rahmen halten wir den folgenden Inhalt für angemessen:

- Einführungen in Client-seitige Web-Techniken: HTML, CSS und JavaScript

- Java Server Pages als Beispiel für grundlegende Server-seitige Web-Techniken
- Umsetzung von MVC in Web-Anwendungen
- Datenbankeinbindung in MVC-basierten Web-Anwendungen
- Verfahren zur Optimierung der Kapselung, Wiederverwendung und Anwendungssteuerung

1.5 Zum „Praktikum Anwendungssysteme“

Im Stundenplan des folgenden Semesters steht die Veranstaltung „Praktikum Anwendungssysteme“. Dies ist so zu verstehen, dass Teams von Studierenden in der vorlesungsfreien Zeit zwischen diesem und dem folgenden Semester eine Web-Anwendung entwickeln, die im Laufe des kommenden Semesters präsentiert wird.

Teilnehmer dieser Veranstaltung können zur Vorbereitung an den Präsentationen des Vorsemesters teilnehmen. Ebenso können Studierende der Folgesemester an den im nächsten Semester folgenden Präsentation teilnehmen.

Hierzu gibt es einige Regeln:

- Anfertigung der Web-Anwendung
 - im Team mit 2 oder 3 Personen.
 - Studierende suchen sich selbst ein Thema, idealerweise in externer Firma
 - Anfertigung des Grobkonzepts während des Semesters
 - Entwicklung der Anwendung incl. Dokumentation hauptsächlich in der vorlesungsfreien Zeit zwischen diesem und nächstem Semester
- Minimalanforderungen
 - lauffähige Webanwendung, die einerseits HTML-Oberflächen besitzt, andererseits Daten speichert, üblicherweise in Datenbanken,
 - technische Dokumentation,
 - Präsentation.
- Beurteilung
 - Es werden die technische Dokumentation, die Präsentation, das Look&Feel der Anwendung sowie der technische Wert der Anwendung begutachtet.
 - Die Verwendung von MVC wird als selbstverständlich vorausgesetzt.
 - Die Verwendung möglichst vieler der in der Veranstaltung behandelten Techniken sowie selbst erarbeiteter weiterer Techniken oder Frameworks ist für den technischen Wert vorteilhaft.
- Abgabetermin ist Sonntag vor dem ersten Vorlesungstag des nächsten Semesters, 23:59h
- Die Präsentation findet in der Regel in der Mitte des Semesters statt. Auf Wunsch der Studierenden kann ggfs. ein Termin bereits zum Semesterbeginn stattfinden.
- Weitere Infos werden in der laufenden Veranstaltung, auf Anfrage oder auf den Webseiten des Dozenten gegeben.

1.6 Benötigte Software

1.6.1 Entwicklungsumgebung Eclipse

In dieser Veranstaltung wird die Entwicklungsumgebung „Eclipse“ verwendet. Zur Anfertigung von Web-Anwendung auf Basis von Java Server Pages benötigen wir die „Eclipse IDE for Java EE Developers“. Die „Eclipse IDE for Java Developers“ reicht nicht aus. Auch „Eclipse Classic“ reicht nicht.

Dagegen ist es weniger relevant, welche Eclipse-Version verwendet wird. Im Rechenzentrum der HS Lu wurde im WiSe 12/13 die Version Eclipse 3.6 („Helios“) verwendet. Zum SoSe 13 ist bereits Eclipse 4.2 („Juno“) aktuell. Grundsätzlich sind alle Eclipse-Versionen ab Eclipse 3.3 („Europa“) für

diese Veranstaltung geeignet. Es spricht daher nichts dagegen, auf dem eigenen Laptop die jeweils neueste Eclipse-Version zu verwenden.

Eine geeignete Eclipse-Version kann kostenlos von der Webseite <http://eclipse.org> heruntergeladen werden.

Eclipse wird als Archivdatei heruntergeladen und auf dem lokalen Rechner einfach entpackt. Am besten wird das Zielverzeichnis entsprechend der Eclipse-Version benannt, etwa C:\Programme\eclipse-4.2, dann können bei Bedarf mehrere Eclipse-Versionen parallel betrieben werden. Eclipse wird gestartet durch Ausführen der Datei eclipse.exe im Stammverzeichnis.

1.6.2 JDK

Eclipse benötigt zum Funktionieren ein Java Development Kit (JDK). Grundsätzlich können auf einem Rechner auch mehrere JDKs installiert sein. In Eclipse muss dann angegeben werden, welches JDK verwendet werden soll. Die JDK-Versionen ab Java 7 (evtl. sogar Java 6) sollten funktionieren.

Eine geeignete Java-JDK-Version kann kostenlos von der Webseite <http://oracle.com/technetwork/java/index.html>, (Weiterleitung von <http://java.sun.com>) heruntergeladen werden.

1.6.3 Apache TomCat

Für die Ausführung von Java Server Pages werden wir den Webserver TomCat benötigen, der als Teil des Apache Projects kostenlos von der Webseite <http://tomcat.apache.org> heruntergeladen werden kann.

Für die Nutzung mit Eclipse sollte die gepackte Version, z.B. „32-bit Windows.zip“, eingesetzt werden und nicht die „Windows Installer“ Version. Die Installation erfolgt dann einfach durch Auspacken. Das Starten und Stoppen des TomCat-Dienstes wird später durch Eclipse geregelt.

In der Vergangenheit gab es bereits Inkompabilitäten zwischen bestimmten Eclipse-Versionen und TomCat-Versionen. In Bezug auf die aktuellen Versionen sind derzeit keine Inkompabilitäten bekannt.

1.6.4 Wireshark

Zur Analyse von http-Netzwerktraffic werden wir den Netzwerk-Analyser „Wireshark“ verwenden. Die jeweils neueste Version kann kostenfrei heruntergeladen werden von <http://wireshark.org>.

1.6.5 Weitere

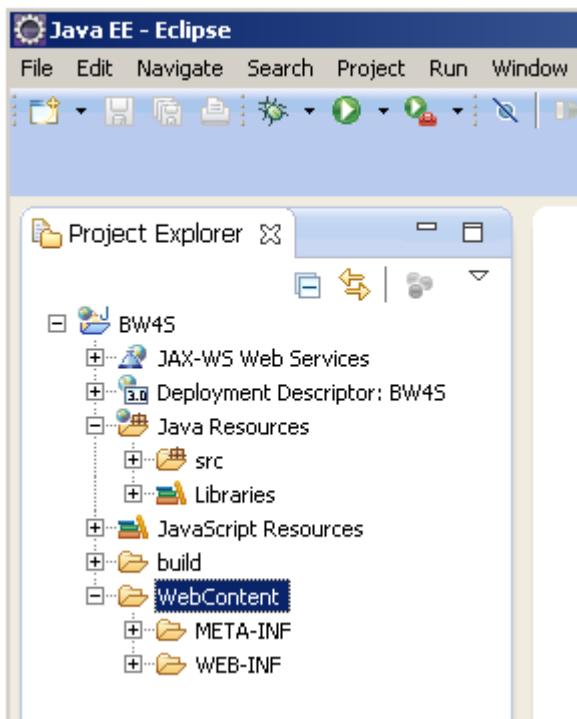
Weitere Software und deren Beschaffungsquellen werden ggfs. in der Veranstaltung genannt.

1.7 Eclipse – Getting Started

Nach dem Start von Eclipse benötigen wir ein „Dynamic Web Project“. In dieser Veranstaltung benennen wir dieses mit „BW4S“:

- Eclipse öffnen mit Workspace
 - Z:\Eclipse-Workspace-2013-SoSe
 - Ggf. diesen Ordner vorher anlegen
- Neues Eclipse-„Project“:
 - Typ: Web → Dynamic Web Project
 - Name: BW4S
 - Falls gefragt wird: Java EE perspective nutzen: Ja

Wir erreichen den folgenden Zustand:



Wir werden in den Verzeichnissen Java Resources/src und WebContent arbeiten.

2 HTML

Da viele Studierende heutzutage bereits HTML-Kenntnisse haben, beschränken wir uns in diesem Kapitel auf eine knappe Einführung und konzentrieren uns auf Elemente, speziell Formularelemente, die im Folgenden für uns wichtig sind. Zuvor geben wir einige Online-Quellen für das weitere Selbststudium an.

2.1 Einführung

2.1.1 Online-Quellen

In den 2000er Jahren war die zentrale Nachschlageressource für HTML im Internet „Self HTML“. Da SelfHTML aber seit 2007 nicht mehr gepflegt wird, werden neuere Entwicklungen wie HTML5 hier nicht mehr berücksichtigt. Trotzdem kann es in vielen Fällen noch zum Nachschlagen dienen:

- <http://www.selfhtml.org> bzw. <http://de.selfhtml.org>

Einige weitere kostenfreie Online-Tutorials sind z.B. die Folgenden. In der Regel enthalten diese auch Tutorials über CSS, PHP oder JavaScript:

- <http://www.planethtml.de/html/>
- <http://www.html-seminar.de/einsteiger.htm>
- <http://de.html.net/tutorials/>

2.1.2 Grundidee

Bei HTML handelt es sich um die Sprache, in der Webseiten formuliert werden. Eine HTML-Seite enthält

- die Inhalte der Seite,
- Tags, die die Seitenstruktur und die Art der Inhalte beschreiben,

HTML-Seiten werden von Web-Browsern (Client-Seite) abgerufen, interpretiert und dargestellt. Der Browser erkennt die Seitenstruktur anhand der Tags und nutzt diese Information zur grafischen Darstellung.

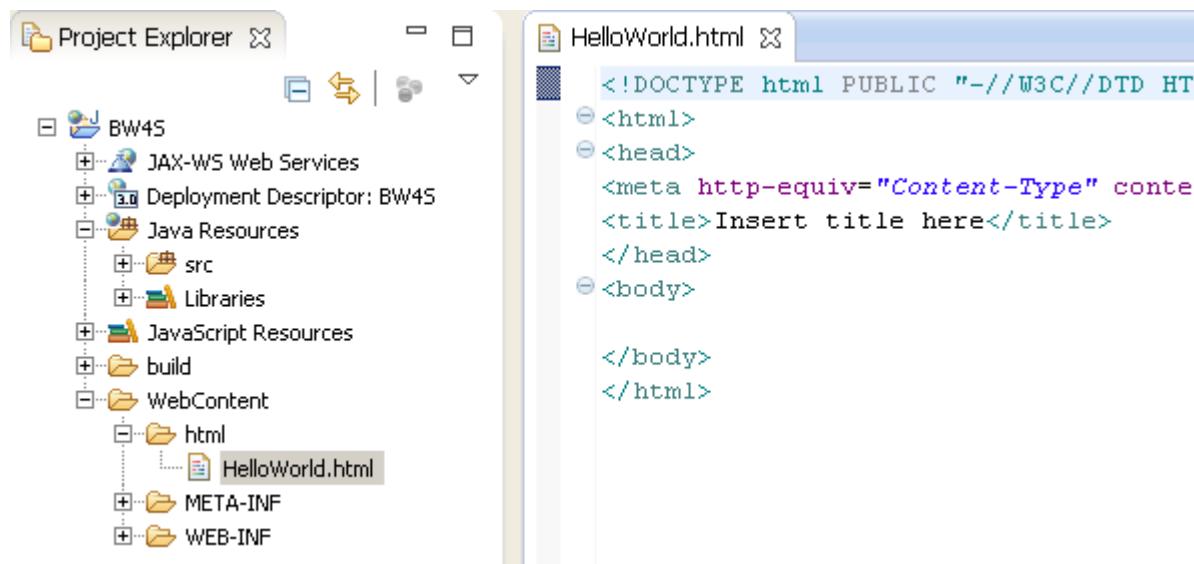
2.1.3 Erstes Beispiel: „Hello World“

Im folgenden Beispiel lernen Sie einige der allerelementarsten HTML-Tags kennen: Rahmen, Kopf, Rumpf, Titel, Überschriften und Absätze

Verzeichnis und Datei anlegen

Erzeugen Sie im Verzeichnis WebContent Ihres Eclipse-Workspace ein Unterverzeichnis „html“ und darin eine HTML-Datei „HelloWorld.html“:

- Rechter Mausklick auf „WebContent“ → New → Folder → Folder name = „html“
- Rechter Mausklick auf „html“ → New → HTML File → File name = „HelloWorld.html“



Titel eintragen

Tragen Sie zwischen <title> und </title> ein: Hello World:

```
<!DOCTYPE html PUBLIC "-//W3C//D
⊕ <html>
⊕ <head>
<meta http-equiv="Content-Type"
<title>Hello World</title>
</head>
⊕ <body>
```

Sie haben damit das <title>-Tag vervollständigt. Tags beschreiben die einzelnen Teile des Seiteninhalts. Zu jedem Tag gehört:

- Öffnendes Tag, z.B. <title>
- Inhalt des Tags, z.B. Hello World
- Schließendes Tag, z.B. </title>

Weitere Tags, die in diesem Screenshot (z.T. nur andeutungsweise) zu sehen sind, sind:

- <html> – Rahmen um die gesamte HTML-Seite mit Ausnahme der Zeile <!DOCTYPE...
- <head> – Kopf: Informationen zur allgemeinen Beschreibung der Seite
- <body> – Rumpf der Seite, in dem alle Inhalte stehen

HTML-Code eintragen

Tragen Sie zwischen <body> und </body> die folgenden Zeilen ein:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Hello World</title>
</head>
<body>
<h1>Ein Gruß an die Welt</h1>
<h2>Ein Gruß an die Welt</h2>
<h3>Ein Gruß an die Welt</h3>
<p>Dies ist ein Gruß an die Welt.  
Dem ist wenig hinzuzufügen.</p>
</body>
</html>

```

Im rechten Fenster sehen Sie den HTML-Code, der einerseits den Seiteninhalt (Text) enthält und andererseits die einzelnen Inhaltsteile in Form von Tags beschreibt. Die Bedeutung der neuen Tags ist:

- <h1> – Überschrift der Stufe 1 („headline“)
- <h2> – Überschrift der Stufe 2
- <h3> – Überschrift der Stufe 3
- <h4> – Überschrift der Stufe 4
- <h5> – Überschrift der Stufe 5
- <h6> – Überschrift der Stufe 6
- <p> – Absatz mit Textinhalt („paragraph“)

Überschriften der Stufen 3 bis 5 dürfen Sie als Übung gerne ebenfalls anlegen.

Webseite anzeigen

Wir können unsere HTML-Seite mit Eclipse anzeigen mit

- Rechter Mausklick auf die Datei „HelloWorld.html“ → Open With → Web Browser



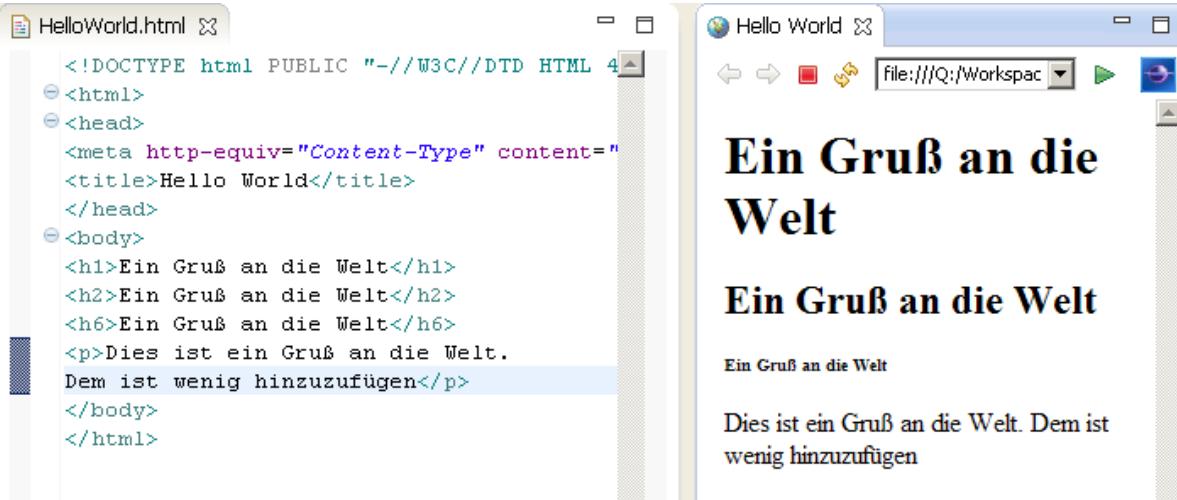
In Eclipse ist ein Web Browser integriert, der hier zum Einsatz kommt. Dieser Web Browser kann selbstständig entscheiden, wo und wie er den Titel, die unterschiedlichen Überschriften und den Fließtext anzeigt.

Mit Eclipse kann eine HTML-Seite auch in einem anderen Web-Browser, der auf dem Rechner installiert ist, angezeigt werden. Die Umstellung erfolgt mit dem Menüpunkt „Window“ → Web Browser → <Auswahl des Browsers>.

Vergleich zwischen HTML-Code und Webseite

In der folgenden Darstellung kann im Vergleich gesehen werden,

- wie HTML-Code als Webseite dargestellt wird und
- wie sich der Browser bei einer schmäleren Seite automatisch verhält.



2.2 Einige Details und Hintergründe

2.2.1 Leerzeichen und Zeilenumbrüche

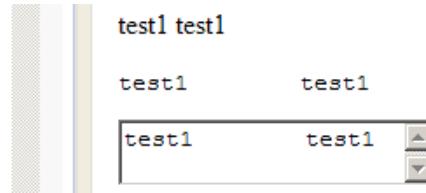
Wenn in einer HTML-Seite mehrere Leerzeichen hintereinander stehen, werden diese vom Browser als ein einziges Leerzeichen interpretiert. Zeilenumbrüche werden gleich behandelt wie Leerzeichen. Leerzeichen und Zeilenumbrüche können also nicht zur Formatierung einer HTML-Seite verwendet werden.

Ausnahmen: Innerhalb folgender Tags gilt dies nicht:

- `<pre>`: „Prä-Formatierung“ – wird in der Regel nicht verwendet
- `<textarea>`: Mehrzeiliges Textfeld – wird gelegentlich im Zusammenhang mit Formularen verwendet.

Beispiel:

```
<p>test1      test1</p>
<pre>test1      test1</pre>
<textarea>test1      test1</textarea>
```



2.2.2 Tags und Attribute

Tags bestehen grundsätzlich aus dem öffnenden Tag, z.B. `<p>`, dem schließenden Tag, z.B. `</p>` und dem Inhalt eines Tags, z.B. `<p>bla bla</p>`. Tags müssen nicht unbedingt einen Inhalt besitzen, z.B. im Fall eines Zeilenumbruchs `

`.

Tags können zusätzliche Informationen in Form von Attributen enthalten. Attribute sind name/value-Paare von Zeichenketten, die im öffnenden Tag in der Form `<name>=<value>` angegeben werden, z.B. im Fall eines Textfeldes in einem Formular:

```
<input type="text" name="userid" value=""></input>
```

Früher wurden sehr häufig Formatierungsinformationen in Form von Attributen angegeben, z.B. `<p align="center">Inhalt</p>`. Da Formatierungen heutzutage über CSS gemacht werden, sind solche Formatierungsattribute nicht mehr üblich und gelten als schlechter Stil.

2.2.3 Leere Tags

Ein Tag ohne Inhalt wird als „leeres“ Tag bezeichnet. Der Inhalt ist das, was zwischen dem öffnenden und dem schließenden Tag steht. Attribute, die ja innerhalb des öffnenden Tags stehen, zählen hier nicht. Beispiele sind demnach

-
</br>
- <input type="text" name="userid" value="" />

Da es in diesem Fall einen erhöhten Schreibaufwand bedeutet, das schließende Tag hinzuschreiben, ist die folgende Abkürzung erlaubt:

-
 ist gleichbedeutend mit
</br>
- <input type="text" name="userid" value="" /> ist ebenfalls vorschriftsmäßig geschlossen.

2.2.4 „Unsauberes“ HTML und Fehlertoleranz von Browsern

Wenn ein Browser ein HTML-Konstrukt nicht kennt oder nicht versteht, ignoriert er es. Damit können auch neue Seiten mit alten Browsern angesehen werden, allerdings werden die neuen Tags oder Attribute dann ignoriert und evtl. die Inhalte der Tags als Text betrachtet und auf der Seite angezeigt, was sehr unerwünscht ist. Eine Seite, die vergleicht, welche Browser-Versionen mit welchen Tags und Attributen umgehen können, ist z.B. <http://www.caniuse.com>.

Viele Web-Browser verstehen auch einige HTML-Konstrukte, die nicht sauber formuliert sind. Dazu gehören:

- Tags ohne Inhalt, die nicht geschlossen werden, z.B.

- Attributangaben ohne Gänsebeinchen („), z.B. type=text
- Attribute ohne value, z.B. <option selected> anstatt <option selected="selected">

Aus verschiedenen, ziemlich tiefliegenden, Gründen sollten solche Unsauberkeiten trotzdem nach Möglichkeit vermieden werden.

2.2.5 HTML-Versionen

HTML entwickelt sich seit den frühen 1990er Jahren durch den Markt getrieben. Standardversionen werden durch das World Wide Web Consortium (W3C, <http://www.w3c.org>) hingen der tatsächlichen Entwicklung oft hinterher. Versionshistorien finden sich in vielen Büchern oder in der Wikipedia (<http://de.wikipedia.org/wiki/Html#Versionen>).

In frühen HTML-Versionen waren die aufgeführten „unsauberen“ Konstrukte sogar erlaubt. Mit den XHTML-Versionen ab 2000 versuchte das W3C dies – leider mit wenig Erfolg – zu verhindern.

Mit HTML5 geht der Trend seit 2009 wieder dahin, diese zu erlauben.

2.2.6 DOCTYPE-Angabe

Jede HTML-Version umfasst einen unterschiedlichen Sprachumfang. Um prüfen zu können, ob eine HTML-Seite „versionsgerecht“ ist, sollte deshalb in der ersten Zeile angegeben werden, in welcher HTML-Version diese HTML-Seite entwickelt ist, z.B. für HTML 4.01 Transitional:

```
 HelloWorld.html &gt;
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

oder für HTML5 einfach:

```
<!DOCTYPE html>
```

2.2.7 XHTML und HTML5

Das W3C hat seit 2000 versucht, „saubere“ HTML-Codierung durchzusetzen. Dafür wurde gefordert, dass HTML auch XML-konform sein soll, das bedeutet:

- Jedes Tag wird grundsätzlich auch wieder geschlossen, z.B.
</br>

- Tags müssen hierarchisch strukturiert sein, d.h. ein inneres Tag muss geschlossen werden, bevor das äußere Tag geschlossen wird, z.B. <head><title>Hello World</title></head>
- Jedes Attribut hat einen value, der grundsätzlich immer in Gänsebeinchen steht.

XHTML hat sich leider nicht besonders gut durchgesetzt und wird seit 2009 von HTML5 abgelöst.

Mit HTML5 werden einige Abweichungen von den XML-Regeln wieder geduldet. Außerdem sollen einige Vereinheitlichungen und Erweiterungen standardisiert werden. Insbesondere geht es dabei auch um mobile Endgeräte. Im Detail geht dies leider über den Umfang dieser Veranstaltung hinaus.

2.2.8 Sonderzeichen und Entities

Da das Internet ursprünglich in den USA entwickelt wurde, waren deutsche Sonderzeichen wie ä, ö, ü, ß zunächst unbekannt und mussten über extra Mechanismen, sogenannte „Entities“, mit einem abgekürzten Befehl oder als Zahlcode, eingebunden werden. Auch weitere Sonderzeichen wurden auf diese Art bereitgestellt, z.B.

Zeichen	Entity	mit ISO-Code
"	"	"
&	&	&
<	<	<
>	>	>
Leerzeichen	 	
©	©	©
®	®	®
Ä	Ä	Ä
Ö	Ö	Ö
Ü	Ü	Ü
ß	ß	ß
ä	ä	ä
ö	ö	ö
ü	ü	ü

Sprachspezifische Zeichen werden heute über passende Zeichensätze eingebunden, Entities werden aber auch über die Definition einzelner Zeichen hinaus z.B. als Abkürzungen noch verwendet.

2.2.9 Goldene Regeln

Um Ihre Webseiten für Benutzer attraktiv zu machen, sollten Sie auf Sorgfalt achten. Die Zeiten von „freakiger Coolness“ im Web sind out. Zur Qualität gehören z.B.

- Klare Seitenstruktur (benutzerfreundlich)
- Konsistentes Design
- Auch im Web auf Rechtschreibung / Grammatik achten
- Nicht mit Farben und unterschiedlichen Stilen überladen (vgl. www.muellseite.de)
- Kurze Ladezeiten: Platz sparen, speziell für Bilder
- Baustellenschilder sind out
- Fehlerhafte Links und Skripte vermeiden

Eine Zeitlang wurden viele Webseiten mit sogenannten Framesets aufgebaut. Auch dies ist heutzutage nicht mehr üblich, weil Suchmaschinen im Web die Startseite besonders hoch bewerten.

2.3 Einige Tags

Sie sollten bisher die folgenden Tags kennengelernt und verstanden haben: <html>, <head>, <title>, <body>, <h1> – <h6>, <p>,
.

Die Tags in diesem Kapitel gehören ebenfalls zum elementaren Handwerkszeug für HTML.

2.3.1 Hyperlinks

Hyperlinks (oder einfach „Links“) geben dem Web seine Grundfunktionalität und sind für dessen Erfolg verantwortlich. Sie werden im Text einer HTML Seite einfach eingefügt, indem der Text, der verlinken soll, in ein Anchor-Tag <a> eingefügt wird.

Beispiel: Aus „<p>Klicken Sie hier, um zur FH Lu zu kommen</p>“ wird



<p>Klicken Sie [hier](http://www.hs-lu.de), um zur FH Lu zu kommen</p>

Klicken Sie [hier](http://www.hs-lu.de), um zur FH Lu zu kommen

Der verlinkende Text (in der Webseite unterstrichen) ist der Inhalt des <a>-Tags, in unserem Fall das Wort „hier“ ohne Komma und ohne Leerzeichen. Das Sprungziel des Links ist der Value des Attributs href = „Hyperlink-Reference“.

Mit dem <a>-Tag sind auch Mail-Verweise möglich. Dabei wird im href-Atribut nicht das http-Protokoll angesprochen, sondern das mailto-Protokoll. Auf dem ausführenden Rechner öffnet sich dann (falls vorhanden) die Standard-Mail-Anwendung. Syntax-Beispiele sind

- Schreiben Sie eine Mail an Gerd Mc Blubb
- href="mailto://gerdmcblubb@trashmail.com?subject=Terminanfrage"
- href="mailto://gerdmcblubb@trashmail.com?subject=Terminanfrage&body=würde%2013h%20passen?"

2.3.2 Listen

HTML kennt nummerierte Listen () und Listen ohne Nummern (). Innerhalb der Listen werden die einzelnen Einträge als List-Items () bezeichnet. Listen können verschachtelt werden.

Beispiel 1: Beachten Sie, wie Sie Ihren HTML-Code mit geeigneter Einrückung lesbar gestalten:



```
<ol>
  <li>Ernie</li>
  <li>Bert</li>
  <li>Krümelmonster</li>
</ol>
```

1. Ernie
2. Bert
3. Krümelmonster

Beispiel 2: Beachten Sie, dass z.B. die innere Liste „Brot / Käse“ komplett zum List-Item „Essen“ der äußeren Liste gehört:

```

<ul>
  <li>Essen
    <ul>
      <li>Brot</li>
      <li>Käse</li>
    </ul>
  </li>
  <li>Trinken
    <ul>
      <li>O-Saft</li>
      <li>Sprudel</li>
      <li>Cola</li>
    </ul>
  </li>
</ul>

```

- Essen
 - Brot
 - Käse
- Trinken
 - O-Saft
 - Sprudel
 - Cola

Nummerierte Listen und Listen ohne Nummern können auch kombiniert werden.

2.3.3 Tabellen

Grundlegende Umsetzung mit HTML

Eine Tabelle ist folgendermaßen strukturiert:

- Der Rahmen ist ein `<table>`-Tag.
- Darin stehen Tabellenzeilen (`<tr>` = table row)
- Jede Zeile enthält Zellen (`<td>` = table data)

Jede Zeile sollte prinzipiell gleich viele Zellen enthalten, dann werden diese als Spalten dargestellt. Andernfalls können – je nach Browser – seltsame Dinge dargestellt werden.

Beispiel:

```

<table>
  <tr>
    <td>Vorname</td>
    <td>Nachname</td>
  </tr>
  <tr>
    <td>Hans</td>
    <td>Matzka</td>
  </tr>
  <tr>
    <td>Bärbel</td>
    <td>Ecker</td>
  </tr>
</table>

```

Vorname	Nachname
Hans	Matzka
Bärbel	Ecker

Elementare Formatierung mit HTML

Das angegebene Beispiel zeigt die Daten tatsächlich tabellarisch an, unterlässt aber eine Formatierung. Wir geben einige HTML-Mechanismen zur Formatierungen an, möchten dabei aber nicht ins Detail gehen, weil Formate heutzutage in CSS definiert werden sollen:

- Das `<caption>`-Tag setzt eine Tabellenüberschrift
- Das `<th>`-Tag anstatt `<td>` kennzeichnet einen Tabellenkopf (table head)
- Das `border`-Attribut gibt der Tabelle einen Rahmen

Beachten Sie jeweils den genauen Ort der Tags bzw. des Attributs:

```

<table border="1">
    <caption>Freunde</caption>
    <tr>
        <th>Vorname</th>
        <th>Nachname</th>
    </tr>
    <tr>
        <td>Hans</td>
        <td>Matzka</td>
    </tr>
    <tr>
        <td>Bärbel</td>
        <td>Ecker</td>
    </tr>
</table>

```

Vorname	Nachname
Hans	Matzka
Bärbel	Ecker

Verbindung von Zellen

Mit den folgenden Attributen können neben- bzw. übereinander liegende Zellen verbunden werden:

- colspan="2" verbindet 2 nebeneinander liegende Zellen
- rowspan="2" verbindet 2 übereinander liegende Zellen

Beispiel 1 – so etwas wird sehr häufig benötigt:

```

<table border="1">
    <tr>
        <th colspan="2">Freunde</th>
    </tr>
    <tr>
        <td>Hans</td>
        <td>Matzka</td>
    </tr>

```

Freunde	
Hans	Matzka
Bärbel	Ecker

Beispiel 2 – interessante Übung, praktisch eher irrelevant:

```

<table border="1">
    <tr>
        <td colspan="2">1</td>
        <td rowspan="2">2</td>
    </tr>
    <tr>
        <td rowspan="2">3</td>
        <td>4</td>
    </tr>
    <tr>
        <td colspan="2">5</td>
    </tr>
</table>

```

1		2
3	4	
5		

Anmerkungen

- Bei HTML-Tabellen lohnt es sich besonders, auf konsistente Einrückungen zu achten, damit der HTML-Code lesbar und verständlich bleibt.

Zur Formatierung:

- Formatierungen sollten in HTML gemacht werden, wenn es um Aussagen zu der Struktur des Inhalts geht, z.B. <caption> oder <th>.
- Wenn es nur um die Darstellung geht (z.B. <border>), dann sollte dies nicht in HTML sondern in CSS gemacht werden.

Zum Einsatz von Tabellen:

- Tabellen sollten nur verwendet werden, wenn es sich wirklich um eine tabellarische Darstellung handelt.
- Einige Zeit wurden Tabellen verwendet, um das Seitenlayout einer Webseite festzulegen. Da die gesamte Seite aber in der Regel keine Tabelle ist, ist dies nicht sinnvoll. Auch Seitenlayouts sollten lieber mit CSS definiert werden.

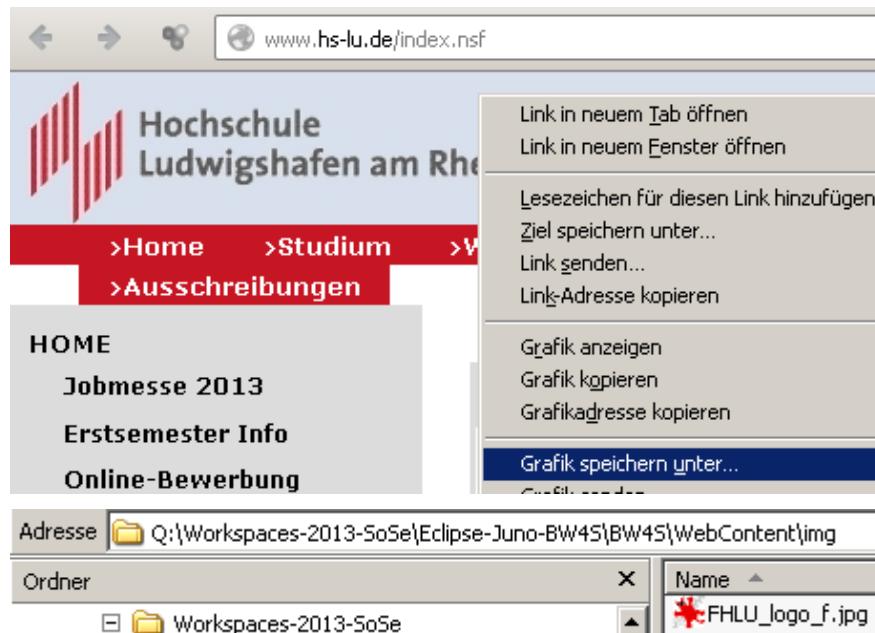
2.3.4 Bilder

Um ein Bild in HTML einzubinden, verwendet man das Tag (Image). Natürlich sollte zuerst ein Bild zum Einbinden vorhanden sein. Wir laden deshalb das Logo der FH in ein Verzeichnis „WebContent/img“ unseres Eclipse-Workspaces:

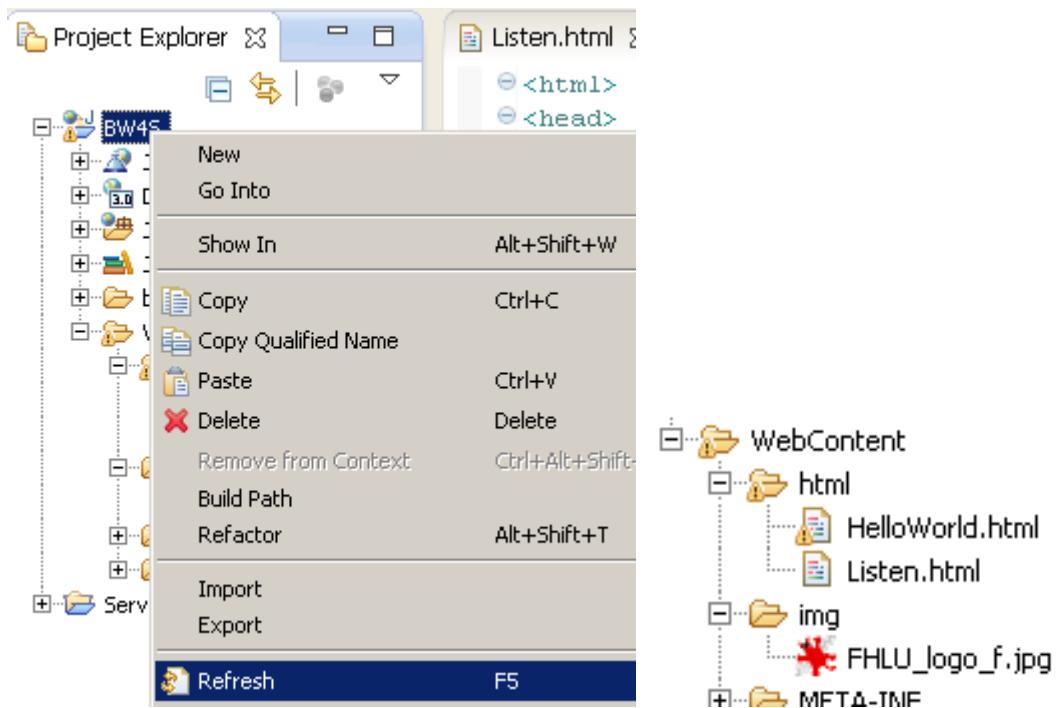
1. Schritt: Verzeichnis „img“ anlegen:



2. Schritt: Homepage der FH Lu anzeigen und Logo speichern in dem soeben angelegten Verzeichnis:



3. Schritt: Aktualisieren des gesamten Eclipse-Projekts (Projektname markieren) oder nur des img-Verzeichnisses (img markieren) mit F5 oder per Kontextmenü → Refresh. Nur wenn dies getan wurde, wird das neue Bild im Workspace angezeigt und kann verwendet werden.



4. Schritt: Einbinden im HTML mit Tag `` und den Attributen `src="<Dateiname>"` und `alt="Logo der Hochschule Ludwigshafen"`:

```

```

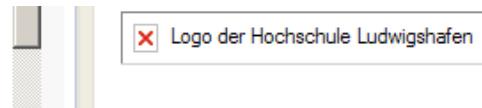


Anmerkungen:

- Wir adressieren das Bild lokal, indem wir vom Verzeichnis „html“ (in dem unsere HTML-Seite steht) ausgehen, dann zunächst mit „..“ eine Verzeichnisebene „nach oben“ gehen (auf das Verzeichnis „WebContent“) und von dort auf das Verzeichnis „img“ und darin auf das Bild zugreifen.
- Wir achten auf Groß- und Kleinschreibung, weil diese mindestens unter UNIX / Linux relevant ist.
- Das ``-Tag hat nur Attribute aber keinen Inhalt. Wir schließen es mit dem Abkürzungsmechanismus aus 2.2.3.
- Das Attribut „alt“ gibt einen Text an, der angezeigt wird, wenn das Bild (aus welchem Grund auch immer) nicht geladen werden konnte:

```

```



Einige Browser stellen den Text des alt-Attributs außerdem als Tooltip dar, wenn der Mauszeiger über dem Bild stand. Dieses Verhalten ist aber falsch! Einen Tooltip codieren Sie mit dem Attribut „title“:

```

```



Weitere optionale Attribute sind z.B. width und height zur Angabe der Bildgröße, die allerdings eher der Formatierung dienen und deshalb durch CSS ersetzt werden sollten.

2.3.5 Bild als Hyperlink

Ein Bild kann einfach zu einem Hyperlink gemacht werden, indem es mit einem `<a>`-Tag umgeben wird. Anders gesagt, wenn das Bild den Inhalt eines `<a>`-Tags darstellt:

```
<a href="http://www.hs-lu.de">
  
</a>
```



Wenn der Link aus einem Text besteht, wird er standardmäßig unterstrichen. Entsprechend wird das verlinkte Bild standardmäßig mit einem Rahmen versehen. Falls dieser Rahmen unerwünscht ist, sollte er mit CSS entfernt werden. In HTML würde es (Achtung: Schlechter Stil) mit einem Attribut `border="0"` im `img`-Tag auch gehen.

2.3.6 Meta-Angaben

Innerhalb des `<head>`-Tags einer HTML-Seite können mit `<meta>`-Tags verschiedene Angaben über die Seite eingetragen werden, sogenannte Meta-Angaben. Typische Beispiele sind:

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<meta name="description" content="HTML Testseite" />
<meta name="keywords" content="HTML, CSS, XML, JavaScript" />
<meta name="author" content="Haio Röckle" />
```

Die Angabe von Inhalt und Zeichensatz kann für die korrekte Darstellung im Browser sinnvoll sein, die Angabe von `description` und `keywords` ist speziell für Suchmaschinen interessant.

2.3.7 Zusammenfassung

Sie sollten bisher die folgenden Tags kennengelernt und verstanden haben: `<html>`, `<head>`, `<title>`, `<body>`, `<h1>` – `<h6>`, `<p>`, `
`, `<a>`, ``, ``, ``, `<table>`, `<tr>`, `<td>`, `<th>`, `<caption>`, ``, `<meta>`.

2.4 HTML-Formulare

Für unser Ziel, Oberflächen für dynamische Web-Anwendungen zu entwickeln, stellen HTML-Formulare und Formularelemente die wichtigsten Inhalte dar.

Formulare dienen in der Regel dazu, Informationen auf einer HTML-Seite zu sammeln (Eingabeseite) und durch Klick auf einen Button an eine andere Seite weiterzureichen (Verarbeitungsseite), wo die Eingaben ausgewertet werden sollen.

Zur Verarbeitung der Daten ist in der Regel eine dynamische Seite nötig, also eine Seite, die Programmcode enthält. Solche Seiten behandeln wir später (JSP).

2.4.1 Das Formular-Tag

Das Formular-Tag fasst alle Eingabefelder eines Formulars zusammen und besitzt ein Attribut „`action`“, das angibt, auf welcher Seite die Eingaben verarbeitet werden sollen sowie ein Attribut „`method`“, das angibt, welche http-Methode zur Übertragung genutzt werden soll. Da wir noch keine Verarbeitungsseite haben setzen wir zunächst `action=""` und solange wir http noch nicht besprochen haben, setzen wir `method="get"`:

```
<form action="" method="get" >

</form>
```

Innerhalb eines Formulars können unterschiedliche Elemente stehen. Die meisten Formularelemente werden durch das Tag `<input>` mit unterschiedlichen Typen implementiert. Außerdem behandeln wir die Formularelemente `<select>` für Auswahlfelder und `<textarea>` für mehrzeilige Textfelder.

Solange wir noch keine Formularelemente eingetragen haben, ist im Browser noch nichts von dem Formular zu sehen.

2.4.2 Input-Elemente

Allgemeines

Die Grundform eines input-Elements ist immer `<input type="" name="" value="" />`

Zu beachten ist:

- Erst durch die Angabe des input-Typs wird daraus ein konkretes Formularelement.
- Nur wenn ein name angegeben wird, wird die Benutzereingabe auch an die verarbeitende Seite weitergegeben.
- Das value-Attribut ist optional. Es ermöglicht die Vorbelegung eines Standardwertes, der (je nach type) durch den Benutzer überschrieben werden kann.

Auflistung

Aus http://www.w3schools.com/tags/att_input_type.asp stammt die folgende Aufstellung der „klassischen“ input-types:

Input-type Description

button	Defines a clickable button (mostly used with a JavaScript to activate a script)
checkbox	Defines a checkbox
file	Defines a file-select field and a "Browse..." button (for file uploads)
hidden	Defines a hidden input field
image	Defines an image as the submit button
password	Defines a password field (characters are masked)
radio	Defines a radio button
reset	Defines a reset button (resets all form values to default values)
submit	Defines a submit button
text	Default. Defines a single-line text field (default width is 20 characters)

Aus derselben Quelle stammen die folgenden „neuen“, d.h. HTML5-input-types, die aber noch nicht (Stand 03/2013) in allen Browern vollständig implementiert sind:

color	Defines a color picker
date	Defines a date control (year, month and day (no time))
datetime	Defines a date and time control (year, month, day, hour, minute, second, and fraction of a second, based on UTC time zone)
datetime-local	Defines a date and time control (year, month, day, hour, minute, second, and fraction of a second (no time zone))
email	Defines a field for an e-mail address
month	Defines a month and year control (no time zone)
number	Defines a field for entering a number
range	Defines a control for entering a number whose exact value is not important (like a slider control)
search	Defines a text field for entering a search string
tel	Defines a field for entering a telephone number

time	Defines a control for entering a time (no time zone)
url	Defines a field for entering a URL
week	Defines a week and year control (no time zone)

2.4.3 Ein einfaches Anmeldeformular

Entwickeln Sie eine HTML-Seite „Formulare.html“ und darin ein Formular, in dem ein Benutzer

- Seine Userid und sein Passwort eingeben kann und
- auf einen Login-Button klicken kann oder
- seine Angaben zurücksetzen (löschen) kann

Erster Versuch

Der HTML-Code

```
<form action="" method="get" >
    <input type="text"      name="userid"    value="" />
    <input type="password" name="password" value="" />
    <input type="submit"   name="login"     value="Login" />
    <input type="reset"    name="reset"     value="Reset" />
</form>
```

führt zu der Browser-Darstellung

Anmerkungen:

- input-type „text“ führt zu einem einzeiligen Textfeld,
- input-type „password“ führt zu einem einzeiligen Textfeld, in dem die Benutzereingabe durch Sternchen ersetzt wird,
- input-type „submit“ führt zu einem Button, der das Formular „ausführt“,
- input-type „reset“ führt zu einem Button, der die Eingaben löscht.

Im value-Attribut könnte ein Standardwert vorbelegt werden, z.B:

```
<form action="" method="get" >
    <input type="text"      name="userid"    value="testuser" />
    <input type="password" name="password" value="geheim" />
    <input type="submit"   name="login"     value="Login" />
    <input type="reset"    name="reset"     value="Reset" />
</form>
```

Dies führt zu der Browser-Darstellung

Anmerkungen:

- Natürlich können die Inhalte der Text- und Passwortfelder nun überschrieben werden.
- Bei submit- und reset-Buttons werden die unter value angegebenen Werte auf die Buttons geschrieben.

Zurücksetzen und Ausführen eines Formulars

Ein Formular wird zurückgesetzt, indem ein darin enthaltener „reset“-Button geklickt wird. Die Textfelder erhalten dann wieder ihren im HTML-Code angegebenen Standardwert.

Ein Formular wird „ausgeführt“ oder „abgeschickt“, indem ein darin enthaltener „submit“-Button angeklickt wird. In dem Fall werden die folgenden Inhalte an die Zielseite („action“-Seite) geschickt:

- alle enthaltenen Text- und Passwortfelder, die einen Namen besitzen, und ihre values,
- der angeklickte „submit“-Button mit seinem name und seinem value.

Wenn action="" angegeben ist, wird als Zielseite wieder dieselbe Seite ausgeführt. Die Übergabe der Formularinhalte erfolgt durch Anhängen an die URL:

Im Detail:

<ml/Formulare.html?userid=testuser&password=geheim&login=Login>

Wir erkennen

- den Namen der aufgerufenen und ausgeführten Seite,
- das Textfeld mit name="userid" und value="testuser",
- das Passwortfeld mit name="password" und value="geheim",
- den submit-Button mit name="login" und value="Login"

Beachten Sie die Groß- und Kleinschreibung:

- Die Namen der input-Elemente (userid, password, login) sollten immer klein geschrieben werden, weil diese später zu gleichnamigen lokalen Java-Variablen führen werden, die bekanntlich ebenfalls kleingeschrieben werden sollen.
- Die values der Textfelder kommen von der Benutzereingabe, so dass wir darauf keinen Einfluss haben
- Die values der submit-Buttons werden auf den Buttons angezeigt, so dass diese an das gewünschte Aussehen angepasst werden sollten.

Labels

Natürlich ist die bisherige Darstellung nicht benutzerfreundlich:

Wir möchten mindestens die Labels „BenutzerId“ und „Passwort“ unterbringen. Außerdem möchten wir nicht alles in einer Zeile haben. Wir versuchen es mit Zeilenumbrüchen

```
<form action="" method="get" >
    BenutzerId: <input type="text"      name="userid"      value="testuser" /><br />
    Passwort:  <input type="password"   name="password"   value="geheim"  /><br />
    <input type="submit"    name="login"     value="Login"   />
    <input type="reset"    name="reset"     value="Reset"   />
</form>
```

und erhalten:

Tabellarische Darstellung

Da dies auch nicht besonders schön ist, versuchen wir es mit einer tabellarischen Darstellung:

- Wir implementieren die gesamte Tabelle innerhalb des form-Tags.
- Die Tabelle bekommt drei Zeilen.

- In den ersten beiden Zeilen bekommen die Labels und die Eingabefelder jeweils eine Tabellenzelle.
- In der dritten Zeile fügen wir beide Buttons in derselben Zelle ein, die andere bleibt leer.
- Die oben benötigten Zeilenumbrüche
 werden wieder gelöscht.
- Wir achten auf ordentliche Einrückung:

```
<table>
  <tr>
    <td>BenutzerId:</td>
    <td><input type="text" name="userid" value="testuser" /></td>
  </tr>
  <tr>
    <td>Passwort:</td>
    <td><input type="password" name="password" value="geheim" /></td>
  </tr>
  <tr>
    <td></td>
    <td>
      <input type="submit" name="login" value="Login" />
      <input type="reset" name="reset" value="Reset" />
    </td>
  </tr>
</table>
```

Das Ergebnis ist deutlich hübscher:

BenutzerId:	<input type="text" value="testuser"/>
Passwort:	<input type="password" value="*****"/>
	<input type="button" value="Login"/> <input type="button" value="Reset"/>

Anmerkung zu <form> und <table>

Es ist möglich,

- eine <table> komplett in eine <form> einzubinden (wie wir gesehen haben) oder
- eine <form> komplett in eine Tabellenzelle <td> einzubinden.

Es ist aber nicht möglich, Tabellenelemente mit <form> zu mischen, d.h.

- <table> → <form> → <tr> → <td> ist ebenso verboten wie
- <table> → <tr> → <form> → <td>

Anmerkung zu Eingabefeldern

Die Textfelder input="text" und input="password" erlauben weitere interessante Attribute:

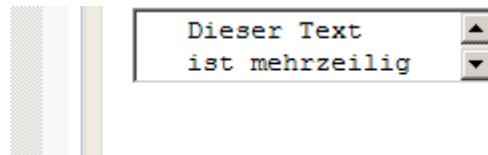
- Mit size="..." kann die Anzeigegröße geändert werden (default ist 20),
- mit maxlength="..." kann die Zeichenanzahl für das Feld eingeschränkt werden (size und maxlength sind voneinander unabhängig),
- auch ein Attribut readonly="readonly" ist erlaubt.

2.4.4 Mehr zu Textfeldern und Buttons

<textarea>

Das Tag <textarea> kann einerseits für die Anzeige von Text dienen, ist dafür aber nicht unbedingt notwendig oder sinnvoll, weil dies auch in einem normalen Absatz (<p>) erfolgen kann. Leerzeichen und Zeilenumbrüche innerhalb eines textarea-Elements werden mit angezeigt. Ein Beispiel:

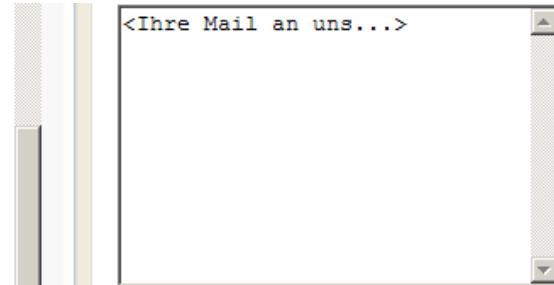
```
<textarea>
  Dieser Text
  ist mehrzeilig
</textarea>
```



Die Größe eines `<textarea>`-Elements kann mit den Attributen „cols“ und „rows“ (Spalten und Zeilen) gesteuert werden. Da es sich dabei um eine Layoutfrage handelt, ist es aber besser, diese Formatierung mit CSS zu erledigen.

`<textarea>` kann auch als Formularelement für mehrzeilige Texteingabe dienen, z.B. für Mail-Formulare. In diesem Fall braucht es ein name-Attribut, damit der Text auch abgeschickt wird. Als value wird der eingegebene Text verwendet. Wenn ein Text vorbelegt werden soll, wird dieser nicht in einem value, sondern als Inhalt des textarea-Elements formuliert:

```
<textarea name=mailBody cols="30" rows="10">
<? Ihre Mail an uns...?>
</textarea>
```



Anmerkungen:

- „<“ und „>“ können in einer HTML-Seite nicht als einfache Zeichen verwendet werden, weil sie dort eine eigene Bedeutung haben (Tag öffnen / schließen). Deshalb müssen diese Zeichen als Entities eingetragen werden, vgl. 2.2.8.
- Der vorgegebene Text darf hier nicht eingerückt werden (um den Code lesbarer zu gestalten), weil die Einrückung sonst selbst als Text betrachtet wird.
- Wird dieses Formular abgeschickt, wird der Text folgendermaßen in die URL eingetragen:
`?e.html?mailBody=%3C!Ihre+Mail+an+uns...%3E%0D%0A&u:`
Erkennbar ist zunächst der name der textarea „mailBody“. Da eine URL keine Leerzeichen und Sonderzeichen enthalten darf, werden diese wieder speziell dargestellt.
- Sonderzeichen werden mit % und Hexadezimalcode dargestellt, In diesem Beispiel stehen die Hexadezimalcodes %3C und %3E für < und > und %0D%0A für einen Zeilenumbruch.
- Leerzeichen erscheinen entweder als Hexadezimalcode %20 oder (so wie hier) als + (Pluszeichen).

Buttons

In 2.4.3 haben wir Buttons als input-Felder mit `type="submit"` (bzw. `type="reset"`) implementiert. Dies ist die „klassische“ HTML-Variante, die im Normalfall immer noch sinnvoll ist. Allerdings gibt es noch andere Varianten:

Input-Feld mit `type="button"`

HTML-Beispiel:

```
<input type="button" name="buttonKlick" value="Klick Mich!" /><br />
```

Darstellung:

Bei Klick auf den Button wird die Form nicht ausgeführt. Solche Buttons sind nur sinnvoll, wenn sie JavaScript-Funktionen auslösen, sonst passiert einfach nichts!

Input-Feld mit type="image"

HTML-Beispiel:

```
<input type="image" src="../img/FHLU_logo_f.jpg" name="imageKlick" /><br />
```

Darstellung:



Es wird kein Button angezeigt, sondern nur das Bild. Bei Klick auf das Bild wird die Form ausgeführt, also die unter action="..." angegebene Seite aufgerufen. (In unserem Fall die Seite Formulare.html und nicht die Homepage der FH).

Dieses Input-Feld braucht einen name (hier: name="imageKlick") aber keinen value. Wenn auf das Bild geklickt wird, wird an die Zielseite übertragen, wo auf das Bild geklickt wurde:

[re.html?imageKlick.x=194&imageKlick.y=48](#)

<button>-Tag

HTML-Beispiel:

```
<button name="buttonTag" value="buttonClicked" >  
    Hey Ho, FH Button!<br />  
      
</button><br />
```

Darstellung:



Das <button>-Tag funktioniert im Wesentlichen gleich wie ein Input-Feld mit type="submit". Es wird ein name und ein value angegeben und wenn auf den Button geklickt wird, wird die Form ausgeführt und beide werden an die Zielseite übertragen:

[re.html?buttonTag=buttonClicked](#)

Da das <button>-Tag einen Inhalt haben kann, kann beliebiger HTML-Inhalt auf den Button gedruckt werden, meistens ein Image. Dadurch ist das <button>-Tag leistungsfähiger als ein Input-Feld. Da das <button>-Tag aber (zumindest in der Vergangenheit) nicht mit allen Browsern gleich funktioniert, sollte es nur verwendet werden, wenn es unbedingt benötigt wird.

2.4.5 Radio Buttons und Checkboxen

Radio Buttons und Checkboxen sind klassische Elemente von grafischen Benutzeroberflächen, nicht nur im Web. Der Unterschied liegt darin, dass

- aus einer Gruppe von Radio Buttons nur höchstens einer und
- aus einer Gruppe von Checkboxen auch mehrere

ausgewählt sein dürfen. Es ist deshalb wichtig, zu markieren, wer zu derselben Gruppe gehört. Im folgenden Beispiel gehören offensichtlich die ersten drei und die letzten drei Einträge jeweils zu derselben Gruppe:

weiblich
 männlich
 unbekannt
 Visacard
 Master Card
 American Express

In HTML werden beide Arten durch Input-Elemente mit speziellen types implementiert:

```

<input type="radio" name="geschlecht" value="w" />weiblich<br />
<input type="radio" name="geschlecht" value="m" />männlich<br />
<input type="radio" name="geschlecht" value="u" />unbekannt<br />
<input type="checkbox" name="kk" value="v" />Visacard<br />
<input type="checkbox" name="kk" value="m" />Master Card<br />
<input type="checkbox" name="kk" value="a" />American Express<br />

```

Anmerkungen:

- Damit diese Felder wirksam werden, müssen sie natürlich in eine <form> integriert werden.
- Mit type="radio" ergibt sich ein Radio-Button, mit type="checkbox" ergibt sich eine Checkbox.
- Beide Arten brauchen einen name und einen value im HTML-Code. Felder mit demselben Namen bilden eine Gruppe. Der Browser überprüft nicht, ob die Namensangaben sinnvoll bzw. konsistent sind.
- Wenn die <form> ausgeführt wird, wird für jedes ausgewählte Feld ein name/value-Paar übertragen. Wenn mehrere checkboxen ausgewählt sind, werden zu einem name mehrere name/value-Paare übertragen, hier z.B. kk=v und kk=a

[re.html?geschlecht=m&kk=v&kk=a&abschicken=Abschicken](#)

- Der angezeigte Text ist nicht Bestandteil des Radio Buttons bzw. der Checkbox und muss separat hingeschrieben werden, ebenso ein Zeilenumbruch falls erwünscht.

Weitere Details:

- Radio Buttons und Checkboxen können mit dem Attribut checked="checked" bereits beim Aufruf der Seite automatisch ausgewählt werden. Speziell bei Radio Buttons kann damit verhindert werden, dass gar keine der Optionen ausgewählt wird:

```
<input type="radio" name="geschlecht" value="u" checked="checked" />unbekannt<br />
```

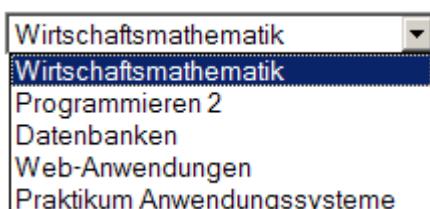
2.4.6 Auswahlfelder mit <select>

Select-Felder haben prinzipiell dieselbe Funktion wie Radio Buttons und Checkboxen allerdings mit anderer Darstellung und Syntax.

Standard: Nur eine Option auswählbar

Die Standarddarstellung für Select-Felder ist

- einzeitig zum Aufklappen
- es kann nur eine Option ausgewählt werden, was der Funktion von Radio Buttons entspricht:



Der zugehörige HTML-Code ist:

```

<select name="veranstaltungen" >
    <option value="BW120">Wirtschaftsmathematik</option>
    <option value="BW241">Programmieren 2</option>
    <option value="BW342">Datenbanken</option>
    <option value="BW441">Web-Anwendungen</option>
    <option value="BW520">Praktikum Anwendungssysteme</option>
</select>

```

Ein Vorteil gegenüber Radio Buttons besteht darin, dass

- der name nur einmal angegeben werden kann und damit automatisch für alle Optionen gilt.
- Die Values werden für jede Option einzeln angegeben.
- Wenn ein value nicht explizit angegeben wird, wird der Inhalt des <option>-Tags als value verwendet (z.B. „Datenbanken“).

Wenn eine Option ausgewählt und das Formular ausgeführt wird, wird das name/value-Paar übertragen, z.B.:

e.html?veranstaltungen=BW342

Erweiterungen

Select-Felder können mit folgenden Attributen erweitert werden:

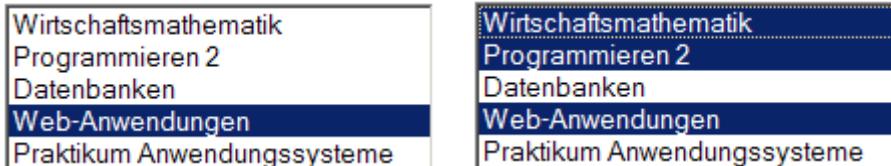
- size="..." – im <select>-Tag, wenn mehrere (oder alle) Optionen sichtbar sein sollen,
- multiple="multiple" – im <select>-Tag, wenn mehrere Optionen auswählbar sein sollen,
- selected="selected" – in einem <option>-Tag, wenn dieses vorausgewählt sein soll:

```

<select name="veranstaltungen" size="5" multiple="multiple">
    <option value="BW120">Wirtschaftsmathematik</option>
    <option value="BW241">Programmieren 2</option>
    <option value="BW342">Datenbanken</option>
    <option value="BW441" selected="selected">Web-Anwendungen</option>
    <option value="BW520">Praktikum Anwendungssysteme</option>
</select>

```

Darstellung:



Übertragung der ausgewählten Optionen:

e.html?veranstaltungen=BW120&veranstaltungen=BW241&veranstaltungen=BW441

Anmerkungen:

- Select-Felder mit Mehrfachauswahl (multiple) entsprechen der Funktion von Checkboxen
- Achtung: Was bei Radio Buttons und Checkboxen checked="checked" heißt, heißt bei Select-Feldern selected="selected"

2.4.7 Tastatursteuerung und weitere input-types

Tastatursteuerung

Formularelemente können mit einem Attribut tabindex="..." versehen werden, mit dem eine Reihenfolge zur Tabulatursteuerung des Formulars vorgegeben werden kann.

Außerdem kann mit einem Attribut accesskey=“...“ festgelegt werden, dass bestimmte Tastendrucke bestimmte Formularelemente aktivieren. Dies werden aber an der Oberfläche nur sichtbar, wenn sie dort explizit markiert werden.

Weitere input-types

Entsprechend 2.4.2 gibt es noch weitere und speziell im Zusammenhang mit HTML5 einige neue input-Felder.

Schon länger besteht das Input-Feld mit type=“hidden“. Dieses erlaubt, ein name/value-Paar mit einem Formular an die Zielseite als Information mitzuschicken, ohne dass es im Formular angezeigt wird. Komplexere Web-Anwendungen machen hiervon häufig Gebrauch.

Die neueren input-types „email“, „date“ und „number“ versprechen, einige alte Fehlerquellen zu schließen. Z.B. war es immer aufwändig, von Hand zu prüfen, ob eingegebene Mail-Adressen tatsächlich Mail-Adressen sind. Diese Prüfung würde mit type=“email“ an den Browser delegiert.

In Bezug auf type=“date“ und type=“number“ ist zu beachten, dass es sich bei HTML-Eingaben immer um Zeichenketten handelt. Auf der Zielseite kommen diese also immer als Datentyp „String“ an und müssen dort entsprechend umgewandelt werden. Die neuen HTML-Input-Types können uns aber helfen, die Werte in den richtigen Formaten an die Zielseiten zu schicken.

3 CSS

Vgl. Foliensatz.

4 JSP

Vgl. Foliensätze.

5 JavaScript

JavaScript ist eine Programmiersprache zur Erweiterung von HTML. JavaScript wird mit der HTML-Seite vom Server geladen, aber auf dem Client ausgeführt. Interessant ist, dass mit JavaScript auf die Elemente der zugehörigen HTML-Seiten zugegriffen werden kann und dass diese damit dynamisch verändert werden können.

Ursprünglich wurde JavaScript oft für grafische Spielereien eingesetzt, z.B. blinkende Inhalte, Mouseover-Effekte oder Aufklappmenüs. Solche Effekte können aber heute meistens bereits mit Standard-CSS oder –HTML erreicht werden. Die wichtigsten Funktionen von JavaScript sind deshalb heutzutage

- die Prüfung von Benutzereingaben,
- das Nachladen von Inhalten,
- das Nachladen von Werbung,
- das Ansprechen externer Schnittstellen.

5.1 Einbinden von JavaScript in HTML

Das Einbinden von JavaScript in HTML-Seiten funktioniert ähnlich wie bei CSS. Üblicherweise wird

- JavaScript-Code in Form von functions
- in eine oder mehrere externe Dateien geschrieben,
- diese Datei(en) im HTML-Header geladen und
- die functions bei Bedarf aufgerufen und ausgeführt.

Es müssen aber nicht zwingend externe Dateien oder functions verwendet werden.

5.1.1 JavaScript-Code direkt hinschreiben

Im Header oder im Body eines HTML Dokuments kann JavaScript-Code eingebunden werden mit einem script-Tag

```
<script type="text/javascript" >alert("Hello World");</script>
```

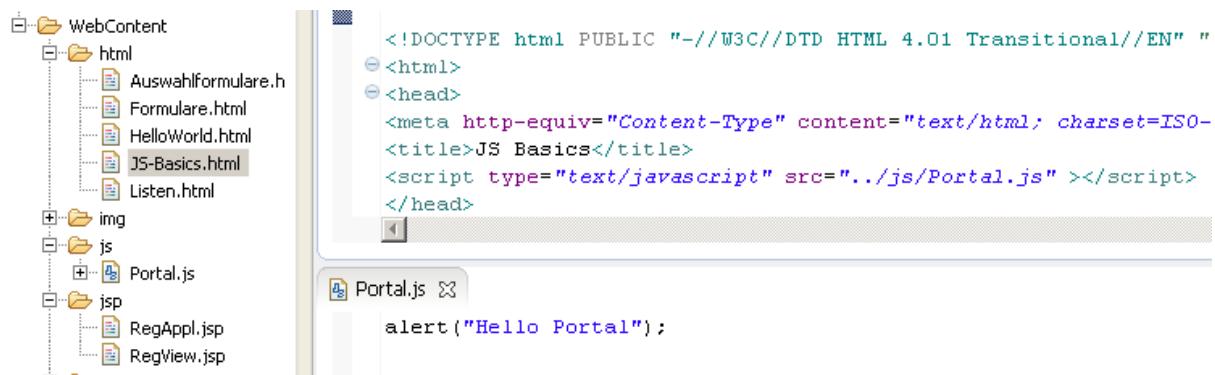
Beachten Sie:

- Das Tag heißt (einfach nur) „script“ und hat das Attribut type="text/javascript"
- Als Inhalt des Tags kann JavaScript-Code formuliert werden. Im dargestellten Beispiel steht der wohl beliebteste Testbefehl alert("Hello World"); Beachten Sie, dass JavaScript-Befehle immer mit Semikolon abgeschlossen werden.
- alert("Hello World"); wird in dem Moment wo es gelesen wird sofort ausgeführt und bringt eine Messagebox auf den Bildschirm, die erst „weggeklickt“ werden muss, bevor es weitergeht.

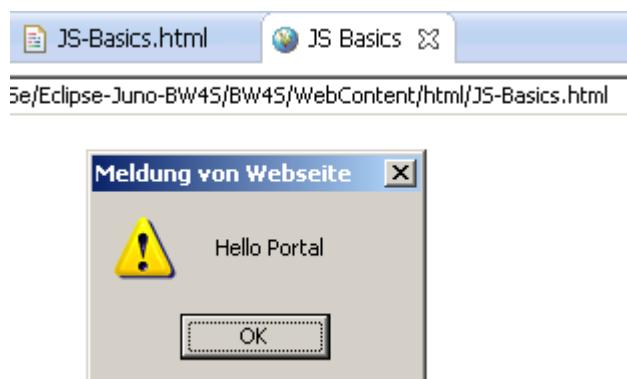
5.1.2 JavaScript-Code in eine externe Datei auslagern

Mit demselben Tag aber einem zusätzlichen src-Attribut kann JavaScript-Code aus einer externen Datei geladen werden (die zuvor angelegt worden sein sollte). Wir verwenden die Datei „Portal.js“ und speichern diese in einem Verzeichnis: „WebContent/js/“.

Anschließend legen wir eine neue HTML-Seite JS-Basics.html an und binden die Portal.js im Header von JS-Basics.html ein:



Der JavaScript-Code aus der externen Datei wird in dem Moment ausgeführt, wo er gelesen wird. Damit ist es vom Ablauf der Skripte her völlig irrelevant, ob der Code in der HTML-Seite oder in einer externen Datei gespeichert wird. Bei der nächsten Anzeige / Aktualisierung wird die alert-Box angezeigt:



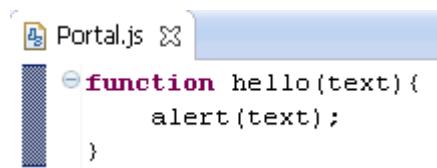
Anmerkungen:

- Um den Ablauf der Skripte zu beeinflussen, werden functions verwendet.
- Üblicherweise wird JavaScript-Code in externen Dateien gespeichert, um die Übersicht im HTML-Code zu bewahren.

5.1.3 JavaScript-functions verwenden

JavaScript function entwickeln

Zur Demonstration einer ersten JavaScript-function ersetzen wir den alert(...)-Code in unserer Portal.js durch



```
Portal.js
function hello(text) {
    alert(text);
}
```

Dabei ist

- „hello“ der Funktionsname
- text der Name einer Variable, die eingegeben werden kann.
- alert(text) der Befehl, der ausgeführt wird.

Beachten Sie, dass Variablen in JavaScript keinen vorgegebenen Datentyp besitzen (ungetypte Sprache). Das macht die Programmierung zunächst einfacher, kann aber später zu mehr Fehlern führen.

JavaScript function einbinden

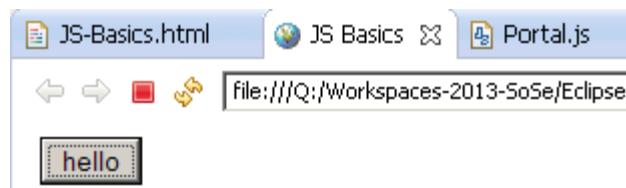
Wir binden die Datei Portal.js in die Seite JS-Basics.html ein wie oben. Wir führen JS-Basics aus und erkennen zunächst, dass von dem JavaScript nichts zu sehen ist: Die Funktion wird nicht ausgeführt.

JavaScript function ausführen

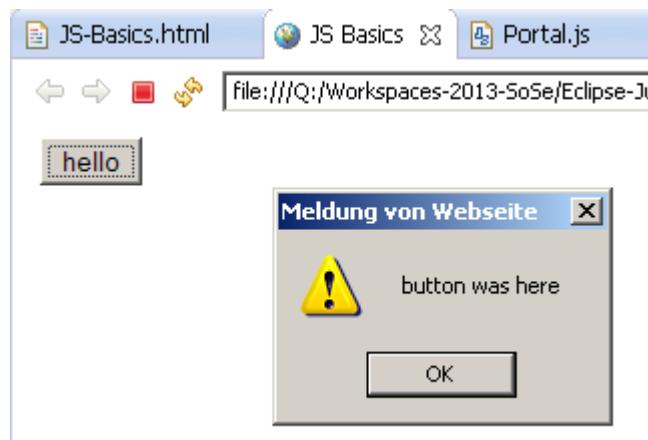
Nun erzeugen wir in JS-Basics.html (nur zum Test) einen Button (z.B. direkt am Anfang des Body), den wir mit der Datei hello verbinden:

```
<body>
<input type="button" value="hello" onclick="hello('button was here')"/>
```

Wenn wir nun die JS-Basics.html ausführen, erscheint der Button:



und wenn wir den Button klicken, erscheint die Alert-Box:



Beachten Sie:

- Das input-Element „button“ braucht kein Formular, da es die action sowieso nicht ausführt. Ein button dient – so wie hier – zum Ausführen von JavaScript.
- Um den Button mit der JavaScript function „hello“ zu verbinden, benutzen wir ein „Ereignis“, nämlich das Ereignis „onclick“.
- Es gibt sehr viele Ereignisse, die das Zusammenspiel zwischen Browser und JavaScript steuern, z.B. onmouseover, etc. Für Geräte mit Touchscreen gibt es seit wenigen Jahren weitere Ereignisse, z.B. touchstart oder touchend.
- Um sauberes HTML zu erzeugen, schreiben wir onclick="...". Der JavaScript-Funktionsaufruf steht also innerhalb der „...“.
- Im Funktionsaufruf wollen wir aber die String-Konstante „button was here“ formulieren. Da wir hier kein doppeltes Gänsebeinchen verwenden dürfen (da sonst der value von onclick beendet wäre), ist es erlaubt, einfache Gänsebeinchen zu verwenden (single quotes: 'button was here'). Das funktioniert übrigens in PHP ganz genau so.

5.2 JavaScript als Programmiersprache

5.2.1 Grundlegendes

Wir geben hier nur einen sehr knappen Überblick über einige Sprachkonstrukte von JavaScript. Die konkrete Syntax und die Funktionsweise im Detail kann in zahlreichen Büchern und an vielen Stellen im Internet nachgeschlagen werden.

Wir wollen hier auch nicht alle Details mit Beispielen belegen. Die LeserInnen sind herzlich aufgefordert, eigene Tests, z.B. in den Dateien JS-Basics.html und Portal.js, durchzuführen.

Ausgabe

Daten und Ergebnisse werden entweder mit alert() ausgegeben oder mit anderen Mechanismen in die zugehörige HTML-Seite geschrieben. Dazu muss das Script auf Elemente der HTML-Seite zugreifen. Der einfachste solche Zugriff ist der Befehl

- document.write("test")

mit dem der angegebene String an der aktuellen Stelle in die HTML-Seite geschrieben wird.

Variablen

In JavaScript werden Variablen ohne Angabe von Datentyp deklariert, z.B.

- var name="Caesar"; // „var“ definiert lokale Variablen
- anzahl = 5; // ohne „var“ werden globale Variablen definiert.

Lokale Variablen haben ihre Gültigkeit analog zu Java. Globale Variablen sollte man meistens nicht benutzen.

Basiskonstrukte

Die meisten bekannten Konstrukte aus Programmiersprachen wie Java oder aus Skriptsprachen wie PHP funktionieren in JavaScript auch, z.B.

- mathematische Berechnungen
- if / else oder switch
- for-, while- oder do...while-Schleifen

Stringverarbeitung

Wie die meisten Skriptsprachen kennt JavaScript viele wirkungsvolle Befehle zur Stringverarbeitung. Grundsätzlich ist ein String ein Objekt (s.u.), so dass die Syntax den folgenden Beispielen entspricht:

- str.charAt(x)
- str.concat(text)

- str.indexOf(zeichen), str.lastIndexOf(zeichen)
- str.substr(p,l), str.slice(p,q)
- str.toLowerCase(), str.toUpperCase()

Arrays

In JavaScript können Arrays ohne Angabe von Länge und Datentyp deklariert werden, z.B.

- var hauptstaedte = new Array();

Der Zugriff kann mit Indizes oder assoziativ erfolgen, z.B.

- hauptstaedte[0] = "Berlin";
- hauptstaedte["Rheinland-Pfalz"] = ["Mainz"];

Im Gegensatz zu Java sind Arrays in JavaScript selbst Objekte (s.u.) und können damit wieder Methoden haben, z.B.

- arr.concat(arr1, arr2, ...), arr.join(zeichen)
- arr.pop(), arr.push(zus1, zus2, ...), arr.shift()
- arr.reverse(), arr.sort()
- arr.slice(von, bis), arr.splice(start, ende, wert1, wert2, ...)

5.2.2 Objektorientierung

Fähigkeit

JavaScript ist generell objektorientiert. Objekte besitzen Attribute und Methoden. Die wichtigsten Objekte sind die Elemente der zugrundeliegenden HTML-Seite, die in dem sogenannten Domain Object Model DOM vorliegen.

Darüberhinaus können Entwickler eigene Objekte anlegen, die Syntax hierfür ist allerdings etwas gewöhnungsbedürftig und muss auch für Anwendungen des DOM nicht unbedingt beherrscht werden.

Klassen

In JavaScript werden keine expliziten Klassen definiert. Stattdessen wird eine Konstruktorfunktion angelegt, in der Objektvariablen (Attribute) mit dem Schlüsselwort this deklariert werden, z.B.

- function Kunde(kunu, vona, nana, adr){

 this.kundnr = kunu;

 this.vorname = vona;

 this.nachname = nana;

 this.adresse = adr;

}

Eine solche Konstruktorfunktion kann dann mit new aufgerufen werden und führt zu einem neuen Objekt, z.B.

- var myKunde = new Kunde("K001", "Hans", "Müller", "Maiweg 1");

Ein Objekt kann auch ganz ohne Klassen- oder Konstruktornamen erzeugt werden (anonymes Objekt):

- var nextKunde =

 {kundnr:"K002", vorname:"Buffalo", nachname:"Bill", adresse:"Wilder Westen 1"};

Objektattribute

Objekte erhalten zunächst die Attribute, die in der Konstruktorfunktion angegeben werden. Auf Attribute wird von innen zugegriffen mit z.B.

- this.vorname

und von außen z.B. mit

- myKunde.vorname

Einem Objekt können später zusätzliche Attribute hinzugefügt werden, z.B.

- myKunde.geburtstag = new Date(1972, 9, 17);

Objektmethoden

Wie in Java können Objekte natürlich auch Methoden enthalten. Diese werden zunächst als function definiert und dann dem Objekt unter einem Namen zugewiesen. Eine function kundeToString kann einem Kundenobjekt z.B. im Konstruktor zugewiesen werden mit

- this.toString = kundeToString;

oder von außerhalb des Objektes mit

- myKunde.toString = kundeToString;

5.2.3 Das Domain Object Model DOM

Jedes JavaScript, das in eine HTML-Seite eingebunden ist, hat Zugriff auf die HTML-Elemente dieser Webseite. Diese Elemente liegen in einer Struktur vor, die Domain Object Model (DOM) genannt wird und der Art ähnelt, wie ein XML-Dokument im Hauptspeicher vorliegt.

Zur Untersuchung des DOM kann zum Beispiel in Firefox das kostenlose PlugIn Firebug installiert werden.

Hierarchie

Das HTML-DOM kennt zunächst die folgenden Top Level Objekte:

- navigator
- screen
- window

Jedes dieser Objekte hat bestimmte Eigenschaften und Methoden, die mit JavaScript abgefragt und bei Bedarf genutzt werden können, z.B.

Eig. / Methode	Erklärung
navigator.appName	Name des Browsers
navigator.appVersion	Version des Browsers
navigator.userAgent	Vollständige Browser-Bezeichnung
navigator.platform	Computer-Plattform (Win32 / Linux, etc.)
navigator.javaEnabled()	Liefert true/false ja nach Browser-Einstellung

und

Eigenschaften	Erklärung
screen.height, screen.width	Information über Höhe und Breite des Bildschirms (Auflösung) in Pixeln
screen.availHeight, screen.availWidth	tatsächlich zur Verfügung stehende Größe ohne Taskleisten, etc.
screen.colorDepth	Verwendete Farbtiefe in Bits pro Pixel.

„window“ stellt den Kontext (this) der JavaScript-Ausführungsumgebung in HTML dar. Alle Methoden, die nicht an ein spezielles Objekt geschickt werden, gehen implizit an window, z.B. ist alert("Hello World") gleichbedeutend mit window.alert("Hello World").

Interessante Methoden von window sind z.B.

Methoden	Erklärung
alert()	Meldungsfenster
confirm()	Bestätigungsfenster anzeigen
prompt()	Texteingabe
blur()	Deaktivierung des Fensters → Hintergrund
focus()	Aktivieren des Fensters → Vordergrund
close()	Fenster schließen
open(url,name,opt)	Öffnet <url> in neuem Fenster mit <name>.
setTimeout()	Setzt / löscht Timer, nach dessen Ablauf ein JavaScript-Ausdruck ausgeführt wird.
clearTimeout()	

window hat außerdem einige interessante Unterobjekte:

- Das Unterobjekt „window.location“ liefert Informationen über die zurzeit geöffnete Seite. Außerdem kann mit location = “...“ auf eine andere Seite gesprungen werden (ebenso mit open(...))
- Das Unterobjekt „window.document“, das wir bereits in document.write("...") kennengelernt haben, öffnet den Zugriff auf die weiteren DOM-Elemente.

Das „document“ Objekt

Das Objekt „document“ bzw. „window.document“ besitzt einige grafische Eigenschaften, die wir heutzutage aber eher durch CSS ersetzen. Außerdem ermöglicht es den Zugriff auf

- die aktuelle URL: „document.URL“ entspricht „window.location“, aber „document.location“ funktioniert (manchmal?) auch
- Cookies

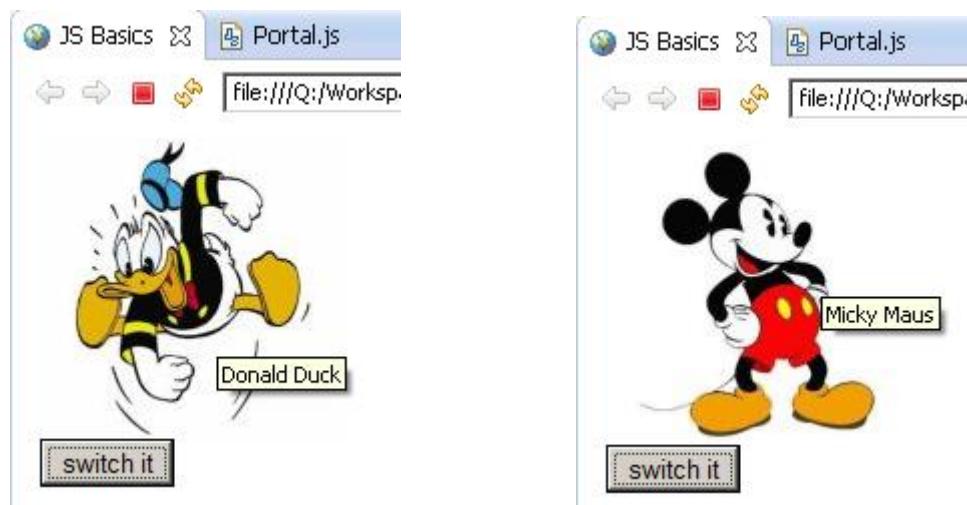
Für uns interessanter ist aber, dass über „document“ der Zugriff auf alle anderen Seiterelemente möglich ist, z.B. liefert

- document.images ein Array aller Bilder der Webseite, also aller HTML-Elemente, die mit <img...>-Tags erzeugt wurden.
- document.links ein Array aller Links der Webseite ()
- document.forms ein Array aller Forms (<form ...>) der Webseite

5.3 „Micky Maus“ – Beispiel

5.3.1 Standardvariante

Im Folgenden möchten wir die gelernten Zugriffsmöglichkeiten an einem Beispiel darstellen: Der Button „switch it“ soll zwischen zwei Bildern hin und her schalten:



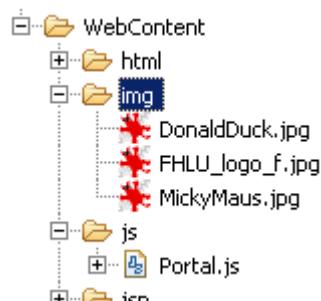
Die Grundidee ist die Folgende:

- Wir lesen das bestehende Bild über das document-Objekt.
- Wenn es „DonaldDuck“ ist, ersetzen wir es durch „MickyMaus“ und umgekehrt.

Schritt 1:

Um zwischen Bildern hin und her schalten zu können, müssen wir diese Bilder erst einmal haben. Besorgen Sie sich also zwei Bilder und speichern Sie diese im Verzeichnis WebContent/img Ihres Eclipse-Workspaces. Um die Bilder im Workspace zu sehen, müssen Sie das Verzeichnis eventuell aktualisieren (F5 oder rechter Mausklick und „Aktualisieren“ oder „Refresh“).

Speichern Sie die Bilder am besten ohne Leerzeichen ab, um spätere Codierungsprobleme zu vermeiden:



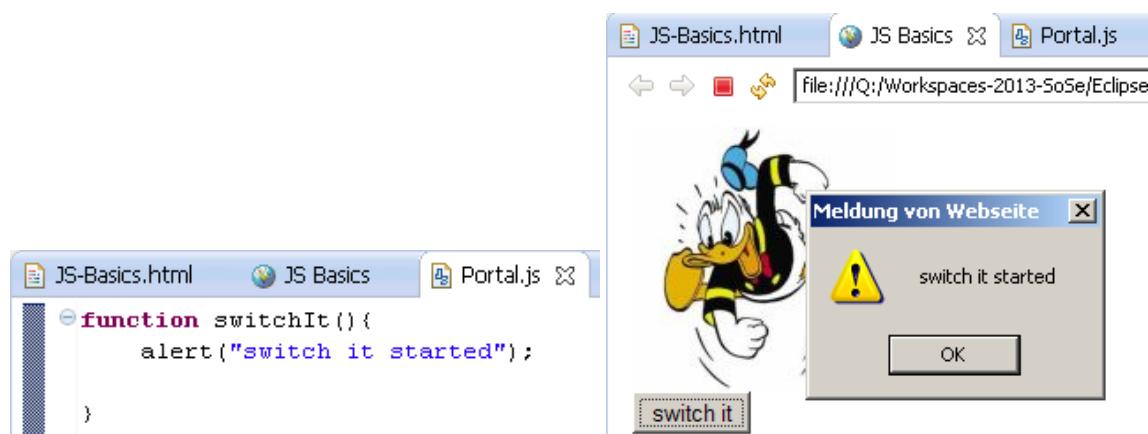
Schritt 2:

Implementieren Sie in Ihrer Webseite JS-Basics.html das img-Element, das zunächst eines der Bilder anzeigt. Setzen Sie auch die Attribute alt und title wie in 2.3.4.

Setzen Sie unter das Bild einen Button mit der entsprechenden Aufschrift und geben Sie diesem ein onclick-Ereignis. Denken Sie sich einen geeigneten Namen für eine JavaScript-Funktion zum Bildertausch aus und verbinden Sie das onclick-Ereignis mit dieser Funktion:

```
<body>
<br />
<input type="button" value="switch it" onclick="switchIt()" /><br />
```

Bereiten Sie außerdem in Ihrer Portal.js eine solche Funktion vor. Beginnen Sie die Funktion mit einem alert(“switch it started“); um erkennen zu können, ob der Aufruf klappt. Führen Sie dann Ihre Webseite JS-Basics.html aus und klicken Sie den „switch it“-Button:



Schritt 3:

Wir beginnen nun damit, das bestehende Bild zu lesen:

- Über das document-Objekt bekommen wir alle Bilder der HTML-Seite als Array. Da die Webseite nur ein Bild enthält, bekommen wir dieses Bild als JavaScript-Objekt mit „document.images[0]“.
- Da uns interessiert, welches Bild aktuell dargestellt wird, lesen wir aus dem Bild-Objekt das Attribut „src“.
- Um zu prüfen, ob das geklappt hat, zeigen wir es mit einem alert an.

```
function switchIt() {
    var myImage = document.images[0];
    var imgSrc = myImage.src;
    alert(imgSrc);

}
```

Anschließend aktualisieren wir die HTML-Seite und klicken den Button:



Beobachtungen:

- Offensichtlich hat das soweit geklappt.
- Die src des Bildes wird nicht nur als „DonaldDuck.jpg“, sondern als voll qualifizierter Pfad angegeben.

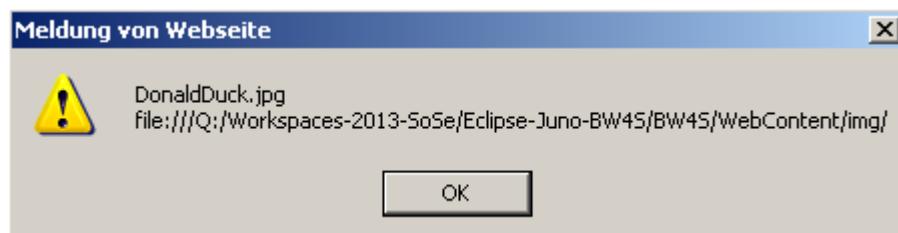
Da wir nur den Dateinamen ändern wollen, schneiden wir diesen aus dem src-String. Um später den vollständigen Dateinamen des anderen Bildes zusammenbauen zu können, brauchen wir auch die Pfadangabe. Zum Aufspalten des Strings verwenden wir die bereits in 5.2.1 vorgestellten Funktionen:

- str.lastIndexOf(<Teilstring>)
- str.substring(start, ende)

Um zu prüfen, ob wir Date- und Pfadnamen korrekt gefunden haben, prüfen wir wieder beides mit einem alert:

```
function switchIt() {
    var myImage = document.images[0];
    var imgSrc = myImage.src;
    var slashPos = imgSrc.lastIndexOf('/');
    var imgSrcFilename = imgSrc.substring(slashPos + 1, imgSrc.length);
    var imgSrcPathname = imgSrc.substr(0, slashPos + 1);
    alert(imgSrcFilename + "/n" + imgSrcPathname);

}
```



Schritt 4:

Wir können uns jetzt an das Tauschen der Bilder machen. Wenn der Dateiname „DonaldDuck.jpg“ ist, soll er gegen „MickyMaus.jpg“ ausgetauscht werden und umgekehrt. Das Ergebnis wird als voll qualifizierter Pfad wieder in das src-Attribut des Bildes zurückgeschrieben. Auch alt und title werden entsprechend angepasst.

Wir gehen so vor, dass wir zunächst die neuen Attributwerte ermitteln:

```
var newSrcFilename;
var newTitle;
if (imgSrcFilename == "DonaldDuck.jpg") {
    newSrcFilename = "MickyMaus.jpg";
    newTitle = "Micky Maus";
} else {
    newSrcFilename = "DonaldDuck.jpg";
    newTitle = "Donald Duck";
}
```

und diese dann in die Attribute des Bildes eintragen:

```
myImage.src = imgSrcPathname + newSrcFilename;
myImage.alt = newTitle;
myImage.title = newTitle;
```

Nach Aktualisieren der JS-Basics.html stellen wir fest: Es klappt in beide Richtungen!

5.3.2 „Micky Maus“ – Logger mit getElementById() und innerHTML

Wir möchten unser Micky Maus-Beispiel erweitern, indem wir alle Änderungen in einem Protokoll festhalten. Wir brauchen dafür

- ein Anzeigeelement für dasProtokoll und eine Möglichkeit, auf dieses Anzeigeelement zuzugreifen,
- eine Möglichkeit, in dieses Anzeigeelement zu schreiben.

Schritt 5:

Als Anzeigeelement kommen z.B. die mehrzeiligen Elemente <textarea>, <div> oder einfach <p> in Frage. Wir verwenden einen einfachen Absatz <p> und platzieren diesen unter dem „switch“-Button.

Um auf diesen Absatz zugreifen zu können, ist die einfachste Metode, diesem eine eindeutige id zuzuweisen, .B. „mickyLogArea“:

```
<body>

</p>
```

Schritt 6:

Wir erweitern unsere JavaScript Methode switchIt() folgendermaßen:

```
var myLogArea = document.getElementById("mickyLogArea");
var mySwitched = "switched from " + imgSrcFilename +
                 " to " + newSrcFilename + "<br />";
myLogArea.innerHTML = mySwitched + myLogArea.innerHTML;
```

Beschreibung:

- Wir besorgen uns zunächst den Absatz „mickyLogArea“ als JavaScript Objekt mit dem Befehl document.getElementById(„mickyLogArea“);
- Anschließend ersetzen wir den kompletten Inhalt des Absatzes durch einen angemessenen Protokollsatz.

Wir erhalten – nach Aktualisierung der Seite und einem Klicken – die folgende Ausgabe. Beachten Sie dabei, dass der jeweils aktuellste Protokollsatz oben steht.



```
switched from DonaldDuck.jpg to MickyMaus.jpg
switched from MickyMaus.jpg to DonaldDuck.jpg
switched from DonaldDuck.jpg to MickyMaus.jpg
switched from MickyMaus.jpg to DonaldDuck.jpg
switched from DonaldDuck.jpg to MickyMaus.jpg
```

Anmerkung:

- Da mit innerHTML = ... der gesamte Inhalt des Absatzes jeweils verarbeitet wird, kann dies bei vielen Protokollsätzen zu großem Aufwand führen. Es ist aber leicht zu verstehen und leicht zu programmieren.

5.3.3 Verschönerung mit Zeit und Farbe

Die Log-Einträge wirken noch ein bisschen unausgereift. Wir wollen deshalb einen aktuellen Zeitstempel davor setzen und die Zeilen unterschiedlich einfärben.

Schritt 7:

Wir entwickeln eine weitere JavaScript function, die uns die jeweils aktuelle Uhrzeit samt Datum schön formatiert als String zurückgibt:

```
function getDateTimeString(){
    var myDate      = new Date();
    var myDay       = toTwoDigits(myDate.getDate());
    var myMonth     = toTwoDigits(myDate.getMonth() + 1);
    var myYear      = myDate.getFullYear();
    var myHours     = toTwoDigits(myDate.getHours());
    var myMinutes   = toTwoDigits(myDate.getMinutes());
    var mySeconds   = toTwoDigits(myDate.getSeconds());

    return myDay + "." + myMonth + "." + myYear + ", "
        + myHours + ":" + myMinutes + ":" + mySeconds;
}

function toTwoDigits(dateNum){
    dateNum = "00" + dateNum;
    dateNum = dateNum.substr(dateNum.length - 2, 2);
    return dateNum;
}
```

Beschreibung:

- Mit new Date() bekommen wir ein Date-Objekt, das Datum und Zeit enthält. Aus diesem Date-Objekt erhalten wir
 - mit getDate() den aktuellen Tag im Monat,
 - mit getMonth() den aktuellen monat im Jahr, allerdings bei 0 beginnend,
 - mit getFullYear() die 4-stellige Jahresangabe,

- mit getHours(), getMinutes() und getSeconds() die Stunden, Minuten und Sekunden.
- Da die Rückgabewerte alles Zahlen sind, müssen wir sie in Strings umwandeln. Das tut JavaScript meistens automatisch. Allerdings können dabei einstellige Strings herauskommen, die in der grafischen Darstellung nicht schön sind. Deshalb verwenden wir die zusätzliche Umwandlungsmethode toTwoDigits(), mit der wir einzelne Zahlen in zweistellige Strings mit ggfs. führenden Nullen umwandeln.

Nachdem wir die Erzeugung des Datumsstrings geregelt haben, können wir diesen in unseren Protokollsatz einbinden:

```
myLogArea.innerHTML = getDateTimeString() + " - " + mySwitched + myLogArea.innerHTML;
```

Wir erhalten:



```
08.03.2013, 11:42:12 - switched from MickyMaus.jpg to DonaldDuck.jpg
08.03.2013, 11:42:09 - switched from DonaldDuck.jpg to MickyMaus.jpg
08.03.2013, 11:42:09 - switched from MickyMaus.jpg to DonaldDuck.jpg
08.03.2013, 11:42:08 - switched from DonaldDuck.jpg to MickyMaus.jpg
```

Schritt 8:

Zur farblichen Darstellung der einzelnen Protokollzeilen wollen wir in ungerade und gerade Protokollzeilen unterscheiden. Wir führen dafür intern einen booleschen Parameter „ungerade“, der angibt, ob der aktuelle switch-Vorgang ungerade oder gerade ist.

Um die einzelnen Protokollsätze mit CSS ansprechen zu können, packen wir diese in je ein div-Tag und geben diesen div-Tags ein Attribut class=“gerade“ bzw. class=“ungerade“. Darauf aufbauend ist die Formatierung mit CSS kein Problem mehr.

Um bei jedem Klick noch auf den alten Wert des booleschen Parameters „ungerade“ zugreifen zu können, müssen wir diesen als globalen Parameter (ohne „var“) außerhalb der Funktion switchIt() definieren:

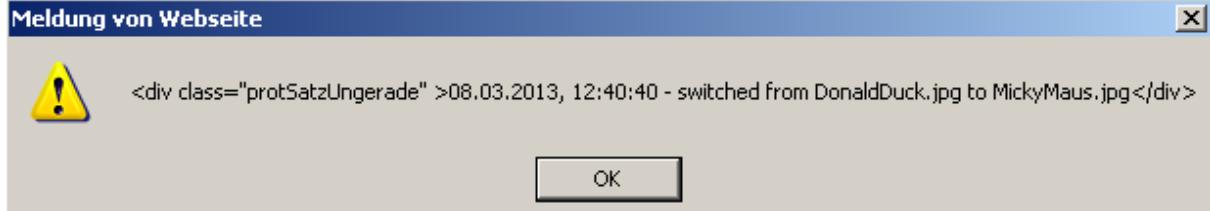
```
ungerade = true;
function switchIt(){
  var myImage = document.images[0];
  ...
```

Den HTML-Code des hinzuzufügenden div-Parameters konstruieren wir wie folgt:

```
var mySwitched = "switched from " + imgSrcFilename +
                 " to " + newSrcFilename;
var myDivContent = getDateTimeString() + " - " + mySwitched;
var myClassString;
if (ungerade) myClassString = "protSatzUngerade";
else myClassString = "protSatzGerade";
ungerade = !ungerade;
var myDivString = "<div class=\"" + myClassString + "\">" + myDivContent + "</div>";
```

Beschreibung:

- mySwitched ist wie in Schritt 6 (nur ohne Zeilenumbruch).
- myDivContent fügt entsprechend Schritt 7 den Datumsstring hinzu.
- myClassString wird je nach Stand der Variable „ungerade“ auf „protSatzUngerade“ oder „protSatzGerade“ gesetzt. Daraufhin wird die boolesche Variable „ungerade“ umgesetzt
- Schließlich wird das div-Tag mit seinem class-Attribut zusammengebaut und in der Variable myDivString gespeichert und erstmal mit alert angezeigt:



Anschließend kann myDivString wie oben am Anfang von myLogArea eingebettet werden:

```
myLogArea.innerHTML = myDivString + myLogArea.innerHTML;
```

Problematischerweise beherrscht der im Eclipse eingebaute (alte IE) Browser dies nicht. Offensichtlich frisst der keine innerHTMLs mit eigenen Tags. Im aktuellen Firefox hat es funktioniert und sah zunächst genauso aus wie nach Schritt 7, weil wir das CSS ja noch nicht definiert haben.

Schritt 9:

Wir binden eine CSS-Datei Portal.css in unsere JS-Basics.html mit ein:

```
<title>JS Basics</title>
<link rel="stylesheet" type="text/css" href="../css/Portal.css" />
<script type="text/javascript" src="../js/Portal.js" ></script>
</head>
```

und schreiben darin:

```
*.protSatzUngerade{
    background-color: lime;
}
*.protSatzGerade{
    background-color: white;
}
```

Und erhalten bei Aktualisierung der JS-Basics.html im Firefox nach einigen switches:



08.03.2013, 12:47:48 - switched from MickyMaus.jpg to DonaldDuck.jpg

08.03.2013, 12:47:21 - switched from DonaldDuck.jpg to MickyMaus.jpg

08.03.2013, 12:47:19 - switched from MickyMaus.jpg to DonaldDuck.jpg

08.03.2013, 12:47:17 - switched from DonaldDuck.jpg to MickyMaus.jpg

08.03.2013, 12:47:14 - switched from MickyMaus.jpg to DonaldDuck.jpg

08.03.2013, 12:47:12 - switched from DonaldDuck.jpg to MickyMaus.jpg

5.3.4 Kompatibilität durch DOM-Techniken

Der innerHTML-Mechanismus hat offensichtlich Schwächen hinsichtlich der Browser-Kompatibilität. Er wird auch nicht vom W3C empfohlen. Stattdessen sollen Mechanismen verwendet werden, die der Programmierung mit XML ähneln. Wir stellen diese der Vollständigkeit dar.

Grundidee

Generell geht es darum, innerhalb des <p> Tags myLogArea Untertags vom Typ <div> anzulegen.

Mit XML-Techniken funktioniert das für jeden Protokollsatz so:

1. Text, der in dem <div>-Knoten stehen soll, konstruieren wie oben. Außerdem den Wert des class-Attributes ermitteln. Beides hat noch nichts mit DOM zu tun.
2. Pointer auf den <p>-Knoten behalten (s.o.: myLogArea),
3. Neuen Textknoten mit dem richtigen Text anlegen,
4. Neuen <div>-Knoten anlegen,
5. Textknoten in den <div>-Knoten einhängen,
6. Class-Attribut mit dem richtigen value innerhalb des <div>-Knotens anlegen,
7. <div>-Knoten (am Anfang!) in den <p>-Knoten einhängen.

Schritt 10:

Wir setzen dies mit JavaScript um. Zunächst lassen wir myLogArea, mySwitched, myDivContent und myClassString unverändert. myDivString brauchen wir nicht mehr:

```
var mySwitched = "switched from " + imgSrcFilename +
                 " to " + newSrcFilename;
var myDivContent = getDateTimeString() + " - " + mySwitched;
var myClassString;
if (ungerade) myClassString = "protSatzUngerade";
else myClassString = "protSatzGerade";
ungerade = !ungerade;
// var myDivString = "<div class=\"" + myClassString + "\">" +
```

Die Punkte 3. – 6. implementieren wir eins-zu-eins:

```
var textNode = document.createTextNode(myDivContent);
var divNode = document.createElement("div");
divNode.appendChild(textNode);
divNode.setAttribute("class", myClassString);
```

Für Punkt 7 können wir (im Gegensatz zu Punkt 5) nicht appendChild(...) verwenden, weil der neue <div>-Knoten sonst am Ende des <p>-Elements eingefügt werden würde. Um den <div>-Knoten am Anfang einzufügen, brauchen wir:

```
myLogArea.insertBefore(divNode, myLogArea.firstChild);
```

Beim Ausprobieren erhalten wir das gewünschte Ergebnis (vgl. Schritt 9) auch im eingebauten Eclipse-Browser.

Beobachtung:

- Die DOM-Konstruktion mag auf den ersten Blick anspruchsvoll aussehen, sie ist aber sehr stringent und durch das Wegfallen von myDivString (s.o.) müssen wir uns auch nicht mehr mit Gänsebeinchen in Strings herumschlagen.

5.3.5 Automatisierung mit Timern

Wir wollen erzielen, dass das Bild nach wenigen Skunden automatisch getauscht wird. JavaScript bietet 2 Möglichkeiten zur Timer-gesteuerten Automatisierung. Beide basieren auf Methoden des Objekts window:

setInterval()

Mit

- var myInterval = window.setInterval("switchIt()", 2000)

wird alle 2 Sekunden (2000 Millisekunden) die Funktion switchIt() aufgerufen. Gleichzeitig wird der Intervallfunktion der Name myInterval gegeben, damit die Ausführung auch wieder gestoppt werden kann, und zwar mit

- window.clearInterval(myInterval)

Wenn wir diese Funktion nutzen möchten, müssen wir genau überlegen, wo wir die Funktion setInterval(...) aufrufen:

- setInterval(...) muss außerhalb der function switchIt() aufgerufen werden, weil switchIt() sich sonst selbst zu oft aufruft.
- setInterval(...) sollte aber durchaus in einer Methode aufgerufen werden, wir brauchen also eine neue „Starter“-Methode, z.B. startIt().
- Um das Intervall wieder stoppen zu können, sollten wir für clearInterval() eine Methode „stopIt()“ anlegen.
- Da wir für das clearInterval() die Variable benötigen, die beim setInterval() erzeugt wurde, dies aber in unterschiedlichen functions erfolgt, muss diese Variable global angelegt werden:

```
function startIt() {  
    switchingInterval = window.setInterval("switchIt()", 2000);  
}  
function stopIt() {  
    window.clearInterval(switchingInterval);  
}
```

Im HTML müssen wir folgende Änderungen vornehmen:

- Der bereits vorhandene Button „switch“ muss per onclick mit der Methode startIt() verbunden werden, anstatt mit switchIt().
- Wir implementieren einen „stop it“-Button, den wir mit der Methode „stopIt()“ verbinden:

setTimeout()

Mit

- var myTimeout = window.setTimeout("switchIt()", 2000)

wird 2 Sekunden (2000 Millisekunden) gewartet und dann ein einziges Mal die Funktion switchIt() aufgerufen. Während der Timeout-läuft, also noch vor der Ausführung der Funktion, kann dieser auch wieder abgebrochen werden, und zwar mit

- window.clearTimeout(myTimeout)

Der Unterschied zu setInterval(..) besteht also darin, dass der Timeout nur zu einer einmaligen Ausführung der Funktion führt. Wenn wir die Funktion mehrmals ausführen möchten, müssen wir also den Timeout auch wiederholt setzen, z.B. am Ende der switchIt()-Methode.

Um dieselbe Funktionalität zu erzeugen, die wir mit setInterval() erzielt haben, sind die folgenden Schritte notwendig:

- Aufruf von setTimeout(...) am Ende der Methode switchIt() implementieren
- switch-Button wieder mit switchIt() verbinden
- Ggf. die startIt()-Methode löschen, weil diese jetzt nicht mehr gebraucht wird.

Wenn der switch-Button geklickt wird, wird switchIt() einmal ausgeführt und am Ende wird der Timeout für die nächste Ausführung gestellt. Wenn der Stop-Button gedrückt wird, wird der aktuelle Timeout abgebrochen und die Schleife stoppt, andernfalls wird switchIt() wieder ausgeführt und am Ende auch ein neuer Timeout gesetzt.

5.4 Anwendungen auf RegView.jsp

Nachdem wir JavaScript jetzt schon ganz gut gelernt haben, kommen wir zu einigen Anwendungen, die im „echten“, meistens kommerziellen, Einsatz von Webseiten häufig verwendet werden.

5.4.1 Eingabeprüfung für Formulare

Das Ereignis „onsubmit“

Wir gehen jetzt von unserer Seite RegView.jsp aus. Genauer: Wir gehen davon aus, dass ein Benutzer seine Eingaben getätigt hat und die Form durch Klicken eines submit-Buttons ausführen möchte.

Bevor die Form tatsächlich an den Server geschickt wird, löst der Browser im form-Element das Ereignis „onsubmit“ aus und wir haben die Möglichkeit, dieses Ereignis mit einer JavaScript function „inputCheck()“ zu verknüpfen:

```
<form action=". /RegAppl.jsp" method="post"
      onsubmit="return inputCheck(this)" >
```

Beschreibung:

- Jeder „submit“ Button, der innerhalb der Form definiert ist, löst das Ereignis „onsubmit“ aus.
- Der Parameter „this“ bezeichnet an dieser Stelle die form selbst. Diese kann damit als Parameter an die aufgerufene JavaScript-function übergeben werden. Das ist sehr wichtig, weil wir in der function von der form auf die Form-Elemente übergehen wollen.
- Der Rückgabewert der function inputCheck(...) sollte boolean sein. Wenn dieser false ist, wird die Ausführung der Form abgebrochen, die Daten werden also nicht an den Server geschickt.
- Wenn inputCheck() true zurückgibt, werden die Daten abgeschickt. Wir hätten aber in unserer JavaScript-function noch die Möglichkeit, die Daten vor dem Abschicken zu ändern.

Anmerkung:

- Wenn wir das DOM-Objekt, das die <form> repräsentiert, nicht mit „this“ übergeben hätten, wäre es immer noch möglich gewesen, dieses, soweit die Seite nur eine form enthält, in der JavaScript function mit document.forms[0] zu bekommen.

Was wir prüfen wollen

In inputCheck() wollen wir die folgenden Prüfungen durchführen:

- Die userid (Nickname) muss gesetzt sein: null und <leerer String> sind verboten.
- Username muss gesetzt sein (s.o.)
- Das Passwort muss gesetzt sein und mindestens 6 Zeichen haben. Im produktiven Einsatz gibt es hier sicher noch mehr und strengere Anforderungen.
- Die E-Mail Adresse darf leer sein. Wenn sie aber gesetzt ist, muss sie die folgenden Anforderungen erfüllen:
 - Es muss genau ein ‘@’ enthalten sein,
 - hinter dem ‘@’ muss mindestens ein ‘.’ enthalten sein,
 - Vor dem ‘@’, zwischen ‘@’ und dem nächsten Punkt sowie hinter dem letzten Punkt muss jeweils mindestens ein Zeichen stehen,

Wenn eine dieser Anforderungen verletzt wird, wird die Form nicht ausgeführt, andernfalls wird sie ausgeführt.

Rückmeldung der Ergebnisse an den Benutzer

Wie wir gleich sehen werden, ist die Durchführung der beschriebenen Prüfungen keine besondere Schwierigkeit. Komplizierter ist die Frage, wie wir unsere Prüfergebnisse an den Benutzer zurückmelden, z.B. in die HTML-Seite eingebettet.

Wir vereinfachen die Situation in diesem Abschnitt damit, dass wir die Ergebnisse einfach mittels einem alert ausgeben. Im nächsten Abschnitt werden wir einen ausgefeilten Mechanismus ausarbeiten.

Zugriff auf die zu prüfenden Felder

Wie oben beschrieben, möchten wir die Benutzereingaben für userid, username, etc. prüfen. Bisher haben wir aber nur das form-Objekt (in der Variable myForm). Der Zugriff auf die eingegebene userid ist nun möglich mit zwei Schritten:

- Zugriff auf das komplette input-Eingabefeld mit Namen „userid“ durch „myForm.userid“
- Zugriff auf den eingegebenen Wert dieses input-Feldes mit „....value“

Zusammengesetzt erhalten wir die eingegebene userid mit

- myForm.userid.value

Analog erhalten wir die anderen zu prüfenden Felder aus den jeweiligen input-Elementen mit

- myForm.username.value
- myForm.password.value
- myForm.email.value

Noch ein Trick zur if-Abfrage

Bevor wir die Prüfung implementieren, stellen wir noch einen Trick vor, den Javascript bereitstellt: Wenn „myVar“ eine Variable ist, die evtl. einen String oder eine Zahl enthält, kann eine Abfrage der folgenden Form getätigt werden:

- if (text){...} else {...}

Die Bedingung if (text) wird als wahr betrachtet, wenn text einen nichtleeren String oder eine Zahl ungleich 0 enthält. In den Fällen null, undefined, <leerer String>, 0, NaN oder false wird die Bedingung als falsch betrachtet.

Durchführung der Prüfungen: inputCheck()

Wir wollen die oben genannten Prüfungen modular ausführen und für jedes zu prüfende Feld eine eigene function entwickeln. Jede dieser functions soll im Fehlerfall eine Message (Typ String) zurückgeben:

- checkUserId(),
- checkUsername(),
- checkPassword(),
- checkEmail().

Wir wollen alle Prüfungen auf einmal durchlaufen und nicht schon beim ersten Fehler abbrechen, deshalb sammeln wir alle zurückgegebenen Messages in einer Variablen und geben diese am Schluss mit einem alert aus.

Um zu ermitteln, ob die Prüfung insgesamt fehlgeschlagen ist, prüfen wir, ob die Message insgesamt leer ist. Falls ja, dann liegt auch kein Fehler vor und die Methode endet mit der Rückgabe „true“, andernfalls wird die Message angezeigt und die Methode endet mit der Rückgabe „false“:

```

function inputCheck(myForm) {
    var msg = "";
    msg += checkUserId(myForm.userid.value);
    msg += checkUsername(myForm.username.value);
    msg += checkPassword(myForm.password.value);
    msg += checkEmail(myForm.email.value);
    if (!msg) {
        return true;
    } else{
        alert(msg);
        return false;
    }
}

```

Detailprüfungen

Die functions checkUserId() und checkUsername() nutzen wieder die oben beschriebene Technik zur Prüfung, ob ein String leer ist. Beide functions arbeiten gleich, müssen aber wegen der unterschiedlichen Fehlermeldungen getrennt behandelt werden:

```

function checkUserId(userid) {
    if (!userid) return "UserId darf nicht leer sein\n";
    else return "";
}
function checkUsername(username) {
    if (!username) return "Username darf nicht leer sein\n";
    else return "";
}

```

Zur Passwortprüfung muss außerdem noch die Länge geprüft werden:

```

function checkPassword(password) {
    if (!password) return "Passwort darf nicht leer sein\n";
    else if (password.length < 6) return "Passwort muss mindestens 6 Zeichen besitzen\n";
    else return "";
}

```

Die Prüfung der E-Mail Adresse ist etwas aufwändiger, verläuft in der folgenden Implementierung aber sehr geradlinig:

```

function checkEmail(email){
    if (!email) return ""; // leerer String ist erlaubt
    else{
        var atPos = email.indexOf('@');
        // kommt ein '@' vor, aber nicht am Anfang oder Ende?
        if (atPos == -1 || atPos == 0 || atPos == email.length - 1){
            return "E-Mail Adresse ist fehlerhaft\n";
        }else{
            var emailParts = email.split('@');
            if (emailParts.length > 2){// zu viele '@'
                return "E-Mail Adresse ist fehlerhaft\n";
            }else{
                var hinterTeil = emailParts[1];
                var dotFirstPos = hinterTeil.indexOf('.');
                // kommt ein Punkt vor, aber nicht direkt hinter '@'?
                if (dotFirstPos == -1 || dotFirstPos == 0){
                    return "E-Mail Adresse ist fehlerhaft\n";
                }else{
                    var dotLastPos = hinterTeil.lastIndexOf('.');
                    // steht der letzte Punkt am Ende?
                    if (dotLastPos == hinterTeil.length - 1){
                        return "E-Mail Adresse ist fehlerhaft\n";
                    }else{
                        return "";
                    }
                }
            }
        }
    }
}

```

Die Prüfungen im Einzelnen:

- Wenn die E-Mail Adresse leer ist, ist die Prüfung bestanden und beendet.
- Andernfalls wird zunächst der '@' gesucht, indem mit `email.indexOf('@')` ermittelt wird, an welcher Position er steht. Die Fehlermöglichkeiten sind:
 - -1: kein '@' vorhanden,
 - 0: '@' ist das erste Zeichen,
 - `email.length - 1`: '@' ist das letzte Zeichen,
- Falls ein '@' vorhanden ist, der nicht am Anfang und nicht am Ende steht, kann der String `email` an diesem '@' aufgeteilt werden mit `email.split('@')`. Das Ergebnis ist ein Array von Teilen, das wir `emailParts` nennen.
- Da bei der Aufteilung nicht mehr als zwei Teilstrings herauskommen dürfen, darf `emailParts.length` höchstens 2 sein.
- Wenn dies erfüllt ist, reicht es, den zweiten Teil der Mail-Adresse (den Teil hinter dem '@') weiter zu untersuchen. Wir geben diesem den Namen „`hinterTeil`“.
- Wir prüfen wieder mit `indexOf(...)`, ob ein '.' enthalten ist. Der '.' darf auch nicht ganz am Anfang von `hinterTeil` stehen, weil dies in der Mail-Adresse direkt nach dem '@' wäre.
- Falls der erste Punkt in Ordnung ist, prüfen wir mit `lastIndexOf('.')`, dass der letzte Punkt nicht am Ende steht. Damit sind die oben genannten Prüfungen erledigt.

Test

Für einen ordentlichen Test müssten sämtliche möglichen Kombinationen der verschiedenen korrekten und fehlerhaften Fälle durchgetestet werden. In diesem Skript beschränken wir uns an dieser Stelle auf ein Beispiel:

The screenshot shows a user registration form with fields for Userid, Username, Password, and E-Mail. A JavaScript error dialog box is overlaid on the form, displaying the message: "Passwort muss mindestens 6 Zeichen besitzen" (Password must have at least 6 characters) and "E-Mail Adresse ist fehlerhaft" (Email address is invalid). The dialog has an OK button.

Userid:	<input type="text" value="bla"/>
Username:	<input type="text" value="bla"/>
Password:	<input type="password" value="***"/>
E-Mail:	<input type="text" value="abc@def"/>
<input type="button" value="Zum Login"/> <input type="button" value="Registrieren"/>	

Warnung

Mit solchen JavaScript-Techniken können Fehleingaben von Benutzern reduziert werden. Es handelt sich bei solchen Client-seitigen Prüfungen aber **nicht** um Sicherheitsmaßnahmen! Ein Angreifer kann jederzeit einen http-Request konstruieren und an den Server senden, ohne die Client-seitigen (JavaScript-)Prüfungen zu durchlaufen. In einem solchen manipulierten oder handgefertigten Request lassen sich beliebige Eingabedaten als name/value-Paare simulieren.

5.4.2 Eingabeunterstützung in Formularen

Mit den dargestellten Prüfungen können wir einem Benutzer **nach** der Eingabe sagen, welche Fehler er gemacht hat. Allerdings ist die Meldungsausgabe mit alert etwas grob.

Interessanter wäre es eventuell, den Benutzer

- schon **während** der Eingabe zu unterstützen und
- die Fehler jeweils dort anzuzeigen, wo sie stehen.

Wir illustrieren dies anhand der folgenden Schritte:

- Für jedes Eingabefeld wird eine Position definiert, wo eine eventuelle Fehlermeldung angezeigt wird.
- Bei der ersten Anzeige des Formulars wird der Cursor (Eingabefokus) auf userid gesetzt. Nach einem fehlgeschlagenen submit wird der Cursor auf das erste Feld gesetzt, zu dem eine Fehlermeldung vorliegt.
- Jeweils wenn sich ein Eingabefeld ändert, wird die zugehörige Prüfung ausgeführt und eine eventuell entstehende Meldung im entsprechenden Fehlerfeld angezeigt.

Meldungsfelder

Die Definition der Meldungsfelder sollte sich nach den Vorgaben der Webdesigner richten. In unserem Fall wollen wir der Einfachheit halber die Tabelle, die die Eingabefelder enthält, um eine weitere Spalte erweitern. Auf diese Weise können wir die Fehlermeldungen immer direkt neben das zugehörige Eingabefeld setzen. Um die Felder leicht mit JavaScript ansprechen zu können, geben wir ihnen id-Attribute. Um den Text farbig hervorheben zu können, bekommen die Felder auch ein class-Attribut (Darstellung nur für userid):

```

<tr>
    <td>UserId:</td>
    <td><input type="text" name="userid"
        value=<jsp:getProperty name="ub" property="userid" />" /></td>
    <td id="useridMsg" class="errorMsg" ></td>
</tr>

```

Um später schnell prüfen zu können, ob ein Eingabefeld fehlerhaft ist, geben wir den <td>-Elementen außerdem noch jeweils ein Attribut „fehler“ mit, die wir beim Laden der Seite direkt auf „false“ setzen. Wir verwenden dafür eine init-Methode in JavaScript:

```
function initMsgFields() {
    document.getElementById("useridMsg").setAttribute("fehler", false);
    document.getElementById("usernameMsg").setAttribute("fehler", false);
    document.getElementById("passwordMsg").setAttribute("fehler", false);
    document.getElementById("emailMsg").setAttribute("fehler", false);
}
```

Diese Methode rufen wir sofort nach dem Laden der Seite auf. Dafür steht das onload-Ereignis des <body>-Tags zur Verfügung, das eintritt, wenn eine Seite mit allen Unterseiten (Skripten, Bildern) vollständig geladen ist:

```
<body onload="initMsgFields()">
```

Fokus beim ersten Ausführen

Ebenfalls sofort nach dem Laden der Seite, also auch in der function initMsgFields(), setzen wir den Eingabefokus auf das input-Feld userid, das wir (wie in 5.4.1) über die Form und den Namen des input-Felds erreichen:

```
function initRegView() {
    document.getElementById("useridMsg").setAttribute("fehler", false);
    document.getElementById("usernameMsg").setAttribute("fehler", false);
    document.getElementById("passwordMsg").setAttribute("fehler", false);
    document.getElementById("emailMsg").setAttribute("fehler", false);
    document.forms[0].userid.focus();
}
```

Prüfungen beim Verlassen eines Feldes

Beim Verlassen eines input-Feldes wird für dieses Feld das Ereignis „onblur“ ausgelöst. Mit diesem Ereignis verbinden wir die Prüfung des jeweiligen Feldes. Den Kern der jeweiligen Prüfung haben wir bereits in 5.4.1 („Detailprüfungen“) implementiert. Da wir dieses Mal aber die Anzeige des Ergebnisses anders regeln möchten, brauchen wir eine weitere function pro input-Feld, die beim Ereignis onblur aufgerufen wird:

```
<td><input type="text" name="userid"
    value=<jsp:getProperty name="ub" property="userid" />
    onblur="checkUserIdFrame(this.value)"
/></td>
<td id="useridMsg" class="errorMsg" ></td>
```

Diese Prüffunktion „checkUserIdFrame(...)“ bekommt den aktuellen Wert des input-Feldes übergeben und ruft damit zunächst die Prüffunktion checkUserId(...) aus 5.4.1 auf:

```
function checkUserIdFrame(userid) {
    var msg = checkUserId(userid);
```

Anschließend ist zu prüfen, ob eine Fehlermeldung zurückgegeben wurde, dann wird diese angezeigt, andernfalls wird eine evtl. Fehlermeldung gelöscht. Außerdem wird das fehler-Attribut, das wir dem Message-<td> zugewiesen haben, entsprechend gesetzt:

```

function checkUserIdFrame(userid) {
    var msg = checkUserId(userid);

    if (msg) {
        document.getElementById("useridMsg").innerHTML = msg;
        document.getElementById("useridMsg").fehler = true;
    } else{
        document.getElementById("useridMsg").innerHTML = "";
        document.getElementById("useridMsg").fehler = false;
    }
}

```

Genauso können wir für alle anderen input-Felder vorgehen. Da wir aber in diesem Fall sehr oft ähnlichen Code aufrufen müssten, lagern wir diesen in eine eigene Methode aus.
checkUserIdFrame(...) reduziert sich dann auf:

```

function checkUserIdFrame(userid) {
    var msg = checkUserId(userid);
    setMsg(document.getElementById("useridMsg"), msg);
    return msg;
}

```

Das Setzen der eigentlichen Meldung erfolgt mit dem folgenden ausgelagerten Code. Wir müssen dann für jedes input-Feld nur noch einmal document.getElementById(...) ausführen.

```

function setMsg(msgField, msg) {
    if (msg) {
        msgField.innerHTML = msg;
        msgField.fehler = true;
    } else{
        msgField.innerHTML = "";
        msgField.fehler = false;
    }
}

```

Nachdem wir dies für alle vier input-Felder erledigt haben, können wir noch die Darstellung der Fehlermeldungen mit CSS anpassen:

```

gView.jsp      Portal.js      Portal.css ✘
@CHARSET "ISO-8859-1";
*.errorMsg{
    text-color: red;
}

```

Wir führen nun die RegView.jsp aus und sehen zunächst, dass der Cursor sofort im Feld „userid“ steht (zunächst ohne Fehlermeldung):

Willkommen am O4S-Portal

Bitte melden Sie sich an

Userid:	<input type="text"/>
Username:	<input type="text"/>

Wenn wir nun keine userid eingeben und auf username klicken oder irgendeine andere Aktion ausführen, die dafür sorgt, dass das userid-Feld den Fokus verliert, erhalten wir unsere Fehlermeldung:

Willkommen am O4S-Portal

Bitte melden Sie sich an

UserId:	<input type="text"/>	Userid darf nicht leer sein
Username:	<input type="text"/>	

Nach erfolgreicher Eingabe verschwindet diese wieder:

Willkommen am O4S-Portal

Bitte melden Sie sich an

UserId:	<input type="text" value="roeckle"/>
Username:	<input type="text" value="Haio Röckle"/>
Passwort:	<input type="password"/>

Prüfungen bei Änderung eines Feldes

Bisher wird die Prüfung nur ausgeführt, wenn ein Feld den Fokus verliert. Um die Prüfung noch schneller auszuführen, können wir andere Ereignishandler verwenden, z.B. onkeydown, onkeypress, onkeyup. Wir verwenden hier „onkeyup“, um dieselbe Prüfung aufzurufen wie bei onblur:

```
<td>Passwort:</td>
<td><input type="password" name="password"
  value=""
  onkeyup="checkPasswordFrame(this.value)"
  onblur="checkPasswordFrame(this.value)"
/></td>
<td id="passwordMsg" class="errorMsg"></td>
```

Wir erreichen damit, dass z.B. bei der Passworteingabe bis zum 5. Zeichen noch die Fehlermeldung stehenbleibt, nach Eingabe des 6. Zeichens aber verschwindet:

Passwort:	<input type="password" value="*****"/>	Passwort muss mindestens 6 Zeichen besitzen
Passwort:	<input type="password" value="***** "/>	

Prüfungen bei submit und Fokus

Wenn ein Button geklickt wird, können wie in 5.4.1 alle Prüfungen auf einmal ausgeführt werden:

```
function inputCheck(myForm) {
  checkUserIdFrame(myForm.userid.value);
  checkUsernameFrame(myForm.username.value);
  checkPasswordFrame(myForm.password.value);
  checkEmailFrame(myForm.email.value);
```

Da hierbei die einzelnen Messages bereits innerhalb der Prüfmethoden verwaltet werden, sind diese nicht mehr relevant. Für das Gesamtergebnis, ob die form in Ordnung ist und abgeschickt werden darf, ziehen wir deshalb die fehler-Attribute der Messagefelder hinzu. Dabei implementieren wir auch die Funktionalität, dass der Eingabefokus auf das erste fehlerhafte Feld gesetzt werden soll:

```

if (document.getElementById("useridMsg").fehler) myForm.userid.focus();
else if (document.getElementById("usernameMsg").fehler) myForm.username.focus();
else if (document.getElementById("passwordMsg").fehler) myForm.password.focus();
else if (document.getElementById("emailMsg").fehler) myForm.email.focus();
else return true;

return false;

```

Wenn gar kein Fehler eintritt, gibt die Methode true zurück und die form wird ausgeführt, andernfalls gibt die Methode false zurück (was natürlich auch schon in die if- bzw. else if-Zweige codiert werden könnte).

Vorgabewerte

Formularfelder im Internet sind häufig mit Vorgaben oder Hinweisen gefüllt, z.B. mit einem grauen Text „Nickname“. Wenn dann in das Feld geklickt wird, dann verschwindet der Text häufig oder wird komplett markiert, damit er bei der ersten Eingabe eines Zeichens verschwindet.

Dies kann ebenfalls mit Javascript implementiert werden und zwar:

- Das Verschwinden mit onfocus=“this.value = ”; “
- Das Markieren mit onfocus=“this.select(); “

Auch das onclick Ereignis könnte hierfür verwendet werden. Dabei ist das Verschwinden vorsichtig einzusetzen, da die Ereignisse jedes Mal ausgeführt werden, wenn das Feld betreten wird und damit evtl. zu oft gelöscht wird. Es müsste also evtl. geprüft werden, ob der aktuelle feldinhalt wirklich der Vorgabewert ist, bevor gelöscht wird.

5.5 Ausblick

Die folgenden Themen haben wir bisher noch nicht behandelt. Für ein interessiertes Selbststudium wären diese einen Blick wert:

- JSON
- JavaScript-Frameworks, z.B. jQuery
- Asynchrone JS-Requests (Ajax)

Software Engineering 2

Thema 2: CSS

**Fachhochschule Ludwigshafen
University of Applied Sciences**

- CSS = Cascading Style Sheets
- Erinnerung: HTML ist eine Auszeichnungssprache, mit der man Inhalt und Darstellung trennen soll, um übersichtlichere Seiten zu schaffen.
- Beispiel ohne CSS:



Ein Gruß an die Welt

Ein Gruß an die Welt

```
<h1 align="center"><b><font color="green" face="monospace" size="6">
    Ein Gruß an die Welt
</font></b></h1>
<h2 align="center"><i><font color="magenta" face="cursive" size="4">
    Ein Gruß an die Welt
</font></i></h2>
<h1 class="o4s">Ein Gruß an die Welt</h1>
<h2 class="o4s">Ein Gruß an die Welt</h2>
```

Übersichtlich???

Übersichtlichkeit

- Beobachtung:
 - Wenn viele Formatierungen benutzt werden, wird der HTML-Code unübersichtlich.
 - Eigentlich sollten (z.B.) alle `<h1>`-Überschriften gleich aussehen.
 - Jeder Web-Designer strebt aber nach Individualität, deshalb gibt es so viele Formatierungsmöglichkeiten.
- Idee:
 - Formatvorlage, die für alle Webseiten einer Website gilt.
 - Englisch: „Style Sheet“, eingedeutscht: „Stylesheet“
 - Jeder Web-Designer kann seine eigenen Stylesheets machen
- Vorteile:
 - Trennen der Formatvorlage vom Text → Übersichtlichkeit
 - Individualität
 - Erlaubt getrennte Bearbeitung von Content und Design
 - Formate können einfach auf andere Seiten übertragen werden.

Lösungsansatz

Alt	Neu
<pre> <html> <head> <title>Beispiel</title> </head> <body> <h1 align="center"> Ein Gruß an die Welt </h1> <h2 align="center"><i> Ein Gruß an die Welt </i></h2> </body> </html> </pre>	<pre> h1 {text-align:center; font-weight:bold; color:green; font-family:monospace; font-size:6; } Ein Gruß an die Welt h2 {text-align:center; font-style:italic; color:magenta; font-family:cursive; font-size:4; } -----</pre> <pre> <html> <head> <title>Beispiel</title> </head> <body> <h1>Ein Gruß an die Welt</h1> <h2>Ein Gruß an die Welt</h2> </body> </html> </pre>

Übersichtlich!!!

Wohin mit der Formatvorlage?

- Es gibt 3 Möglichkeiten:

1. Direkte Formatierung: style-Attribut im Tag:

```
<h1 style="text-align:center; font-weight:bold;  
          color:green; font-family:monospace;  
          font-size:6;">  
    Hello World!  
</h1>
```

→ Doofe Idee, weil die Formatierung wieder unübersichtlich mitten im HTML steht.

→ Nur geeignet zum Ausprobieren!

2. Zentrale Formatierung:

- Das Stylesheet steht im Header, dann sind Format und Inhalt getrennt.
- Für den Fall, dass der Browser kein CSS kennt, wird das Stylesheet in Kommentare gesetzt.

```
<head>  
  <meta http-equiv="Content-Type"  
  <title>Hello World</title>  
  <style type="text/css"><!--  
    h1 {text-align:center;  
        font-weight:bold;  
        color:green;  
        font-family:monospace;  
        font-size:6; }  
    Ein Gruß an die Welt  
    h2 {text-align:center;  
        font-style:italic;  
        color:magenta;  
        font-family:cursive;  
        font-size:4; }  
  //-->  
  </style>  
</head>
```

Wohin mit der Formatvorlage?

3. Möglichkeit: Externe Formatierung

- Das Stylesheet kommt in eine eigene Datei, z.B. WebContent/css/o4s.css
- Mit einem <link>-Tag wird es in den Header der HTML-Datei eingebunden

```
<head>
    <meta http-equiv="Content-Type"
    <title>Hello World</title>
    <link rel="stylesheet"
          type="text/css"
          href="../../css/o4s.css" />
</head>

<h1>Ein Gruß an die Welt</h1>
<h2>Ein Gruß an die Welt</h2>
```



```
*helloWorld.html
o4s.css X

@CHARSET "ISO-8859-1";
h1 {text-align:center;
    font-weight:bold;
    color:green;
    font-family:monospace;
    font-size:6; }
h2 {text-align:center;
    font-style:italic;
    color:magenta;
    font-family:cursive;
    font-size:4; }
```

Ein Gruß an die Welt

Ein Gruß an die Welt

Inhalt des Style Sheets

- Grundsätzlich:
 - Irgendwelche **Eigenschaften bzw. Formate** werden irgendwelchen **HTML-Elementen** zugewiesen.
- Damit gibt es zwei Aufgaben:
 1. Welche Eigenschaften bzw. Formate gibt es?
 2. Welche Möglichkeit der Zuweisung an HTML-Elemente gibt es?
 - Diese Aufgaben können getrennt betrachtet werden.
- Einfachste Möglichkeit:
 - `<Element> {<Eigenschaften>}`
 - Beispiel: Vorhergehende Folie.
- Anmerkung:
 - Wenn man direkt mit dem `style`-Attribut in einem HTML-Tag arbeitet, bezieht sich dies natürlich auf das Tag und braucht keine Zuordnung mehr.
 - Beispiel: `<h1 style="color:green;">Hello World!</h1>`

Zuweisungsmöglichkeiten

- Erweitert: Mehrere Elemente ...
`h1, h2, p {text-decoration:underline;
text-transform: lowercase;}`
- Alle Elemente:
`* {font-family: Arial, sans-serif;}`
- Element nur wenn es ein bestimmtes Attribut hat:
`h1[align=center]{margin 1cm 2cm;}`
- Alle Elemente, die ein bestimmtes Attribut haben:
`*[align=center]{background-color: rgb(64,255,64);}`
- Verschachtelte Elemente (z.B. em nur innerhalb von p):
`p em{font-weight:cursive;}`
- Individuelle Formate (kommt noch):
`p#myId{background-image:url(fhlogo.jpg);}`
- Alle Elemente einer bestimmten Klasse (kommt noch):
`*.o4s{word-spacing:3pt;}`

Individuelle Zuweisung

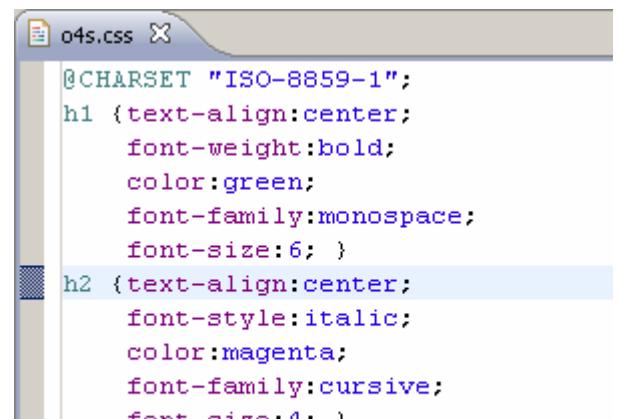
- Obwohl es eigentlich gegen das Grundprinzip verstößt,
 - wollen Web-Designer ab und zu einen Absatz bzw. ein HTML-Element, das anders ist als andere...
- Möglichkeiten:
 - Individuelle Formatierung mit HTML oder `style="..."` im Element.
 - Individuelle Kennzeichnung des Elements durch ein Id-Attribut
- Anmerkung:
 - Eine Id soll in einer HTML-Datei nur einmal vorkommen.
- Bsp.

```
<h1 class="o4s">Ein Gruß an die Welt</h1>
<h2 class="o4s">Ein Gruß an die Welt</h2>
<h3 id="special-o4s">Ein Gruß an die Welt</h3>
```

Ein Gruß an die Welt

Ein Gruß an die Welt

~~Ein Gruß an die Welt~~



```
@CHARSET "ISO-8859-1";
h1 {text-align:center;
    font-weight:bold;
    color:green;
    font-family:monospace;
    font-size:6; }

h2 {text-align:center;
    font-style:italic;
    color:magenta;
    font-family:cursive;
    font-size:4; }

#special-o4s{text-align:center;
              text-decoration:line-through;}
```

Klassenweise Zuweisung

- Mit Klassen kann man in einer HTML-Datei
 - unterschiedliche Arten desselben HTML-Elements festlegen.
- Im Gegensatz zu Ids
 - darf man Klassen in einer HTML-Datei mehrmals verwenden.
- Bsp.:

```
<p class="serioes">Sehr geehrter Herr Wichtig:</p>
<p class="serioes">Hallo Welt - Hello World!</p>
<p class="spass">tralala, t&uuml;del&uuml;&uuml;</p>
```

```
*.serioes{font-family:arial,sans-serif;}
*.spass{font-family:fantasy;
         font-size:2em;
         border:2px solid blue;}
/* Dicke Art Farbe */
```

Sehr geehrter Herr Wichtig:

Hallo Welt - Hello World!

tralala, tüdefüü

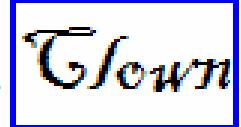
Elemente div und span

- div und span sind HTML-Elemente ohne eigene Bedeutung!
 - Im Gegensatz dazu haben „echte“ Elemente wie h1 oder p eine Bedeutung.
- Das ist nur sinnvoll, wenn
 - man diese mit mit id oder class zusätzlich formatiert.
- Unterschied:
 - div ist ein Blockelement, d.h. es produziert vorher und nachher einen Zeilenumbruch
 - span ist ein inline-Element, d.h. es produziert nichts.

```
<div class="serioes">
    <p>Sehr geehrter Herr Wichtig:</p>
    <p>Hallo Welt - Hello World!</p>
</div>
<p>Ich habe heute einen
    <span class="spass">Clown</span>
gefrühstückt.
</p>
```

Sehr geehrter Herr Wichtig:

Hallo Welt - Hello World!

Ich habe heute einen  gefrühstückt.

Anwendung von div

- div wird inzwischen sehr häufig verwendet
 - Ganz eigene Formate ohne Altlasten
 - Zusätzliche Formate, Rahmen, etc.
 - Positionierung ganzer HTML-Bereiche!
- Positionierung
 - Früher wurden HTML-Bereiche in Tabellen eingewickelt, ganz früher in Frames.
 - Heute werden diese in div-Container eingewickelt und mit Positionierungsangaben versehen
 - Dies erfolgt mit der CSS-Eigenschaft „position“ und einer Option absolute, relative, static

Weiteres

- CSS ist inzwischen sehr mächtig.
 - Es gibt ganze Bücher dazu, in denen man z.B. findet, wie man Aufklappmenüs mit CSS baut.
- Zustandsabhängige „Pseudo-“formate:
 - `a:hover{color:red;}` – Link, über dem sich die Maus befindet
 - `p:first-letter{font-size:2em;}` – Erster Buchstabe eines Absatzes
 - `p:before{content:"-->";}` – Text vor jedem Absatz
- Formatierungssyntax findet man z.B. kostenlos bei

de.selfhtml.org/css/index.htm

Positionierung und Anzeige von Elementen

- ↳ [Allgemeines zur Positionierung und Anzeige von Elementen](#)
- ↳ [position \(Positionsart\)](#)
- ↳ [top \(Startposition von oben\)](#)
- ↳ [left \(Startposition von links\)](#)
- ↳ [bottom \(Startposition von unten\)](#)
- ↳ [right \(Startposition von rechts\)](#)
- ↳ [width \(Breite\)](#)
- ↳ [min-width \(Mindestbreite\)](#)
- ↳ [max-width \(Maximalbreite\)](#)
- ↳ [height \(Höhe\)](#)
- ↳ [min-height \(Mindesthöhe\)](#)
- ↳ [max-height \(Maximalhöhe\)](#)
- ↳ [overflow \(Elementbereich mit über großem Inhalt\)](#)
- ↳ [direction \(Richtung\)](#)
- ↳ [float \(Textumfluss\)](#)
- ↳ [clear \(Fortsetzung bei Textumfluss\)](#)
- ↳ [z-index \(Schichtposition bei Überlappung\)](#)
- ↳ [display \(Anzeigeart bzw. Nichtanzeige ohne Platzhalter\)](#)
- ↳ [visibility \(Anzeige bzw. Nichtanzeige mit Platzhalter\)](#)
- ↳ [clip \(Anzeigebereich eingrenzen\)](#)

CSS-Eigenschaften

Sinnvolle Vorkenntnisse zum Verständnis:  [HTML](#)

Dieser Abschnitt behandelt die einzelnen CSS-Eigenheiten. Der Abschnitt besteht aus folgenden Seiten:

- [Schriftformatierung](#)
- [Schriftformatierung mit Schriftartendaten](#)
- [Ausrichtung und Absatzkontrolle](#)
- [Außenrand und Abstand](#)
- [Innenabstand](#)
- [Rahmen](#)
- [Hintergrundfarben und -bilder](#)
- [Listenformatierung](#)
- [Tabellenformatierung](#)
- [Pseudoelemente und Pseudoklassen](#)
- [Positionierung und Anzeige von Elementen](#)
- [Layouts für Printmedien](#)
- [Sound-Kontrolle für Sprachausgabe](#)
- [Anzeigefenster](#)
- [Filter \(nur Microsoft\)](#)

Software Engineering 2

Thema 7: Java Beans und Sessions

**Fachhochschule Ludwigshafen
University of Applied Sciences**

HTTP hat kein Gedächtnis

- Grundsätzlich ist HTTP ein zustandsloses Protokoll.
 - Jeder HTTP-Request ist völlig unabhängig von allen anderen.
 - Für Web-Anwendungen wie Online-Shops o.ä. ist das ungeeignet.
- Wir brauchen ein Verfahren
 - wie ein Client wiedererkannt werden kann bzw.
 - wie wir dem Client ein Gedächtnis beibringen
- Die technischen Möglichkeiten sind
 - Cookies
 - URL-Rewriting

`http://www.roeckle.de/index.php?sessionid=7392401`

 - Versteckte Felder

`<input type="hidden" name="session" value="7392401" />`

HTTP und Sessions

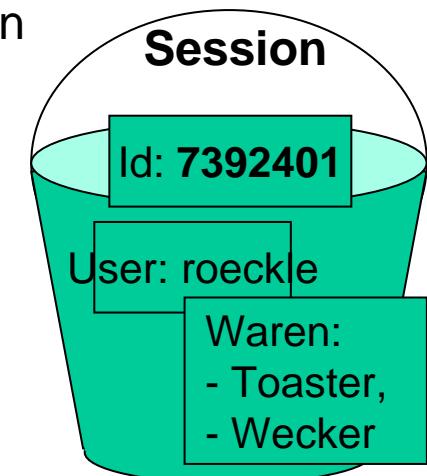
- Eine Session bezeichnet (semantisch / abstrakt):
 - Alle zusammengehörenden Seitenaufrufe eines Benutzers.



- Die technische Implementierung einer Session
 - geht mit unterschiedlichen Mechanismen (vgl. letzte Folie).

JSP und Sessions

- In JSP
 - brauchen wir uns nicht von Hand mit den genannten Session-Mechanismen rumzuschlagen.
 - Das System stellt uns ein Objekt `session` bereit.
 - Die konkrete Implementierung (Cookie, URL-Rewriting, Hidden Field) übernimmt der Servlet-Container (Tomcat).
 - Diese ist für uns transparent: Wir müssen gar nicht wissen, wie er das genau macht.
- Das `session`-Objekt ist vordefiniert und dient
 - Technisch: Der dauerhaften Speicherung von Daten über mehrere Requests hinweg.
 - Logisch: Dem Transport von Informationen zwischen Webseiten
- Das `session`-Objekt funktioniert
 - so ähnlich wie eine Collection in Java.
 - Es kann Objekte (nicht nur Strings) aufnehmen und zurückgeben.



JSP und vordefinierte Objekte

- Wir haben inzwischen folgende vordefinierten Objekte in JSP kennengelernt:

Name	Klasse / Interface	Methoden
out	javax.servlet.jsp. JspWriter	print(), println(), etc.
request	javax.servlet.http. HttpServletRequest	getParameter(String), getParameterNames(String), etc.
response	javax.servlet.http. HttpServletResponse	sendRedirect(), setContentType(), etc.
session	javax.servlet.http. HttpSession	setAttribute(String, Object), Object getAttribute(String), etc.

- Diese werden vom System bereitgestellt.
- Wir können einfach darauf zugreifen.
- Wir brauchen sie nicht erst mit `new ...()` erzeugen

Session und Objekte

- In der session können wir
 - beliebige Objekte (nicht nur Strings) speichern und zwar
 - wieder als name/value-Paar aber
 - dieses Mal in der Form String → Object
- Die Befehle dazu sind

Methode	Beschreibung
setAttribute(String name, Object obj)	Speichert obj unter dem Namen name in der session
Object getAttribute(String name)	Liest aus der session das Objekt, das dort unter dem Namen name gespeichert ist.

- Beispiel:

```
UserBean ub = new UserBean();
session.setAttribute("user", ub);
UserBean ub1 = (UserBean) session.getAttribute("user");
```

Session und Objekte

- Erklärung:

- `getAttribute(String name)` hat den Rückgabetyp `Object`.

```
<% session.getAttribute %>
```

- `getAttribute(String arg0) Object - HttpSession`
- `getAttributeNames() Enumeration - HttpSession`

- Wenn also ein Objekt aus der `session` gelesen wird, dann erhalten wir dieses Objekt zunächst unter dem Datentyp `Object`.
 - Wenn wir wissen, dass es sich dabei um eine `UserBean` handelt, dann wollen wir das Objekt in Java auch als `UserBean` – also in einer Variable vom Typ `UserBean` speichern.
 - Damit das passt, müssen wir das zurückgegebene Objekt zuerst casten.

```
UserBean ub1 = (UserBean) session.getAttribute("user");
```

Session vs. Request

Request	Session
Ist nur für einen einzigen Aufruf gültig	Ist über mehrere (viele / alle) Seitenaufrufe hinweg gültig
Enthält Parameter aus HTML-Formularen	Enthält Objekte aus Java-Programmen
nur Lesen: <pre>String request. getParameter(<name>)</pre> Geschrieben wird vom System (automatisch)	Schreiben und Lesen: <pre>session. setAttribute(<name>, <Object>) Object session. getAttribute(<name>)</pre>
Enthält nur Strings	Enthält allgemeine Objekte, also String, int oder andere Objekte
Kann unter einem Namen auch mehrere Werte enthalten <pre>String request. getParameterValues(<name>)</pre>	Kann unter einem Namen nur einen Wert (nur ein Objekt) enthalten.

Session - Sicherheit

- Session-Ids werden vom Server erzeugt.
 - Dafür muss ein sicheres Zufallsverfahren genutzt werden, damit ein Angreifer die Session-Id nicht erraten kann.
- Sessions dürfen nicht ewig dauern, sonst
 - läuft der Speicher des Servers voll und
 - Angreifer hätten mehr Zeit, eine Session-Id zu erraten
- Wann ist eine Session zu Ende?
 - Wenn der Benutzer sich abmeldet. Dafür gibt es die Methode `session.invalidate()`
 - Wenn ein Benutzer den Browser einfach abschaltet, dann bekommt das der Server erstmal gar nicht mit, deshalb muss eine Maximalzeit definiert sein. Dafür gibt es die Methode `session.setMaxInactiveInterval(int)`
- Anmerkung:
 - Eine Default-Timeout-Dauer kann auch im Server (Tomcat) festgelegt werden.

Session – Weitere Methoden

- Zusammenfassung der Methoden von session:

- void setAttribute(String, Object)
- Object getAttribute(String)
- void invalidate()
- void setMaxInactiveInterval(int)

} bereits besprochen

-
- void removeAttribute(String)
 - long getCreationTime()
 - String getId()
 - long getLastAccessedTime()
 - int getMaxInactiveInterval()
 - boolean isNew()

} vgl. java.sun.com/j2ee/5/docs/api/
→ HttpSession

➔➔➔ Übung JSP 4: Urlaubsziele freestyle

Sessions und MVC

- Grundsätzlich kann man
 - in der `session` auch einfache Daten transportieren wie `int`,
`String`, `StringBuffer`, etc.
- Nach MVC
 - sollten unsere Daten sowieso in Java Beans gegliedert sein,
 - deshalb bietet es sich an, die kompletten Beans in der `session` zu transportieren.
- Erinnerung: In unserer Beispielanwendung O4S Portal
 - haben wir es geschafft, dass ein User in die Datenbank geschrieben (registriert) wird.
 - Wir haben aber keine Rückmeldung (Erfolg oder Misserfolg) bekommen und
 - wenn es schief gegangen ist, waren im Formular trotzdem alle Werte weg.

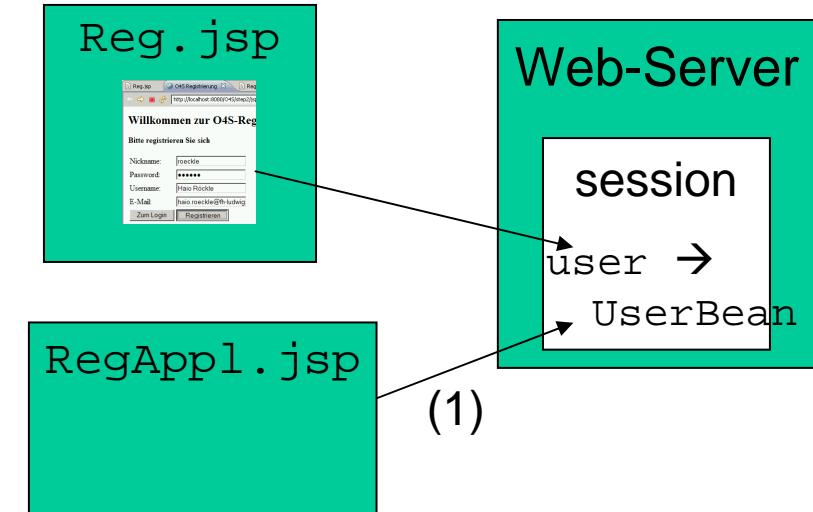
O4S Portal

- Aktion: Erweitern Sie Ihre Registrierungsanwendung, so dass
 - die Formulardaten nach einem Klick nicht einfach weg sind.
- Beobachtung:
 - Bei dem Schritt von `RegAppl.jsp` (zurück) nach `Reg.jsp` wird kein Button geklickt, also auch kein Request ausgelöst.
→ Die Daten müssen über die `session` transportiert werden.
 - Die Daten stehen ja schon in einer `UserBean`!
- Lösungsansatz:
 - In `RegAppl.jsp` wird die gesamte `UserBean` in die Session geschrieben.
 - In `Reg.jsp` wird sie wieder ausgelesen
 - Beim Seitenaufbau werden die Werte aus der `UserBean` gelesen.

O4S Portal

- Schritt 1: Wir erweitern unsere RegAppl.jsp, so dass
 - vor dem response.sendRedirect()
 - die UserBean in die session geschrieben wird
- Ansatz:
 - Wir verwenden den Namen "user"
 - Wir schreiben den Code bevor die insert-Aktion ausgeführt wird.

```
de.fh_lu.o4s.beans.UserBean user
    = new de.fh_lu.o4s.beans.UserBean();
...
user.setEmail(email);
// UserBean in die session schreiben
session.setAttribute("user", user);
// Aktion ausführen und auswerten
int rc = user.insertUpdateUser();
if (rc == 0 || rc == 1)
```



O4S Portal

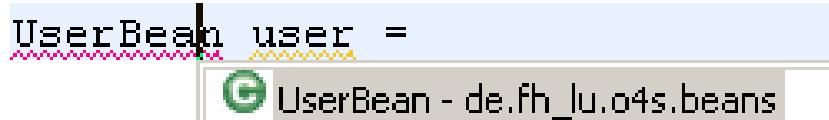
- Schritt 2: Wir erweitern unsere Reg.jsp, so dass
 - ganz am Anfang
 - die UserBean aus der session in eine Variable gelesen wird.

```
<body>
<%>
de.fh_lu.o4s.beans.UserBean user =
    (de.fh_lu.o4s.beans.UserBean) session.getAttribute("user");
%>
<h2>Willkommen zur O4S-Registrierung</h2>
<h4>Bitte registrieren Sie sich</h4>
```

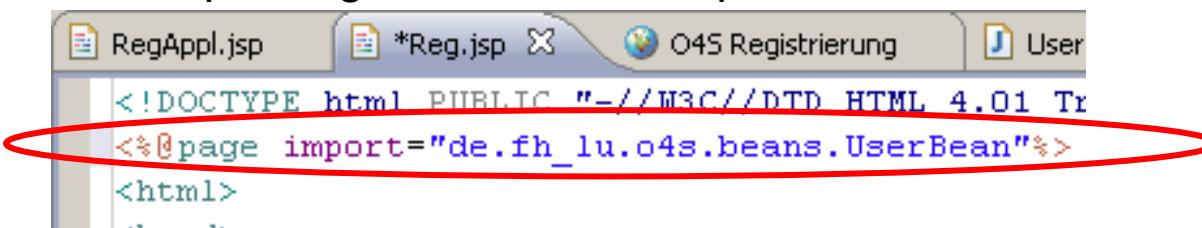
- Beobachtung:
 - Jetzt steht Code in der View, was nach dem MVC Prinzip eigentlich nicht sein darf → das lösen wir später.

O4S Portal

- Schritt 2, Forts.: Die volle Qualifizierung der UserBean (Package-Angabe) macht den Code schwerer lesbar
 - Wir brauchen eine import-Funktionalität
- Ansatz mit Eclipse Auto-Vervollständigung:
 - Wir geben nur UserBean ein (ohne Package) und tippen Ctrl-<blank> bzw. Strg-<Leertaste>



- Eclipse schlägt die möglichen Klassen vor, wir wählen aus.
→ Eclipse ergänzt oben die import-Zeile



- Ergebnis:
 - Im Code dürfen wir die Package-Angabe jetzt weglassen:

```
UserBean user = (UserBean) session.getAttribute("user"); 15
```

O4S Portal

- Schritt 2, Forts.: Was passiert beim ersten Aufruf der Reg.jsp?
 - Da steht noch keine UserBean in der session.
→ `getAttribute("user")` liefert null zurück.
 - Es ist besser, wenn man sich drauf verlassen kann, dass ein Objekt nicht null ist!
→ Deshalb erzeugen wir in diesem Fall selbst eine UserBean und speichern diese auch gleich in der session.

```
UserBean user = (UserBean) session.getAttribute("user");
if (user == null) {
    user = new UserBean();
    session.setAttribute("user", user);
}
```

O4S Portal

- Schritt 3: Wir erweitern unsere Reg.jsp, so dass
 - in den Formularfeldern die User-Informationen eingetragen werden, wenn welche vorhanden sind.
- Erinnerung:
 - Bei der Definition der Formularfelder im HTML-Code wird ein value angegeben:
`<input type="text" name="userid" value="" />`
- Ansatz:
 - Dort kann ein Wert aus der UserBean eingetragen werden:
`<input type="text" name="userid" value="<% user.getUserid() %>" />`
- Anmerkung:
 - Anstatt `<% user.getUserid() %>` hätten wir hier auch `<% out.print(user.getUserid()); %>` schreiben können

O4S Portal

- Schritt 3, Forts.:
 - Dieselbe Ergänzung machen wir für die Felder Username und E-Mail.

```
<input type="text" name="username"  
       value="<% user.getUsername() %>" />  
  
<input type="text" name="email"  
       value="<% user.getEmail() %>" />
```
 - Für das Passwort-Feld ist es nicht sinnvoll, weil man nicht lesen kann was drinsteht (könnte falsch sein).

O4S Portal

- Schritt 4, Ausprobieren
 - Idealerweise vorher Server neu starten,
 - ggfs. auch Server löschen und Workspace refreshen (schadet nie)



The image shows two screenshots of a web browser displaying the O4S registration process. A large green arrow points from the left screenshot to the right one.

Screenshot 1 (Left): Step 2

Willkommen zur O4S-Reg

Bitte registrieren Sie sich

Nickname:	roeckle
Password:	*****
Username:	Haio Röckle
E-Mail:	haio.roeckle@fh-ludwig

Zum Login Registrieren

Screenshot 2 (Right): Step 4

Willkommen zur O4S-Registrierung

Bitte registrieren Sie sich

Nickname:	roeckle
Password:	*****
Username:	Haio Röckle
E-Mail:	haio.roeckle@fh-ludwig

Zum Login Registrieren

- Beobachtung:
 - Scheint funktioniert zu haben, aber
 - eine Erfolgs-/Misserfolgsmeldung gibt es immer noch keine

O4S Portal

- Beobachtung
 - Wir haben in der `Reg.jsp` ganz am Anfang sichergestellt, dass eine `UserBean` existiert, sowohl in der Session als auch in unserer Variable `user` → das ist gut und konsistent.
 - In der `RegAppl.jsp` wird die `UserBean` allerdings eher ad hoc angelegt → weniger gut und konsistent
- Beans lieber am Anfang anlegen, denn dann
 - sieht man sofort, welche Daten in der JSP eine Rolle spielen,
 - kann man sie später nicht mehr vergessen,
 - können Sie nicht `null` sein
 - achtet man in der gesamten Seite eher darauf, was mit der Bean passiert.

O4S Portal

- Aktion: Passen Sie Ihre RegAppl.jsp so an, dass
 - auch hier die UserBean am Anfang angelegt wird.
- Ansatz:
 - Die folgenden Zeilen nach oben ziehen

```
de.fh_lu.o4s.beans.UserBean user = new de.  
user.setUserid(userid);  
user.setPassword(password);  
user.setUsername(username);  
user.setEmail(email);  
// UserBean in die session schreiben  
session.setAttribute("user", user);  
// Aktion ausföhren und auswerten  
int rc = user.insertUpdateUser();
```

- Außerdem
 - die UserBean importieren,
 - bedenken, was passiert, wenn in der session schon eine UserBean gespeichert ist
 - bedenken, wie die Sessiondaten mit den Requestdaten zusammenhängen.

O4S Portal

- Schritt 1: Wir können die Zeilen zum Einlesen der UserBean (und ggfs. zum neu anlegen) aus der Reg.jsp übernehmen

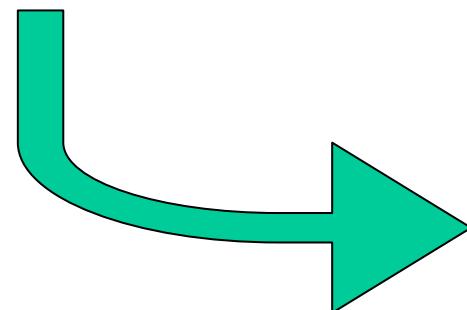
```
<%@page import="de.fh_lu.o4s.beans.UserBean"%>
...
// Beans / Session pflegen
UserBean user = (UserBean) session.getAttribute("user");
if (user == null) {
    user = new UserBean();
    session.setAttribute("user", user);
}
```

- Damit können wir uns darauf verlassen,
 - immer genau eine gültige UserBean (\neq null) zu haben.
- Allgemein:
 - Anwendungsdaten werden in einer Anwendungsbean gespeichert.
 - Evtl. kann man auch mehrere Beans gebrauchen.

O4S Portal

- Schritt 2: Sessiondaten vs. Requestdaten
 - Sessiondaten sind alt, Requestdaten sind neu
 - es ist völlig in Ordnung, die Werte in der session mit den Werten aus dem request zu überschreiben

```
// Felder einlesen
String userid  = request.getParameter("userid");
String password = request.getParameter("password");
String username = request.getParameter("username");
String email    = request.getParameter("email");
```



```
if (register.equals("Registrieren")){
    //Button "Registrieren" wurde geklickt
    user.setUserId(userid);
    user.setPassword(password);
    user.setUsername(username);
    user.setEmail(email);
    // Aktion ausführen und anwerten
```

O4S Portal

- Anmerkungen
- Ein neuer `request` mit neuen Daten aktualisiert nun auch die `session`.
- Das `UserBean`-Objekt ist immer noch dasselbe Objekt.
 - Wir haben nur seine Attribute geändert.
 - Das `UserBean`-Objekt selbst ist und bleibt in der `session` verankert (solange wir daran nichts ändern)
 - wir brauchen die Bean nicht neu in die `session` zu schreiben...
 - ... es würde aber auch nichts schaden.

O4S Portal

- Um Erfolgs-/Misserfolgsmeldungen ausgeben zu können, beachten wir das Folgende:
 - Der Erfolg / Misserfolg entsteht auf der `RegAppl.jsp` kurz vor der Weiterleitung
 - Die `Reg.jsp` ist nur eine (dumme) View, die gar nicht weiß, was in der Anwendung passiert ist.
→ Idealerweise setzen wir die Meldung bereits in der `RegAppl.jsp` und transportieren Sie dann zu der Seite, wo sie angezeigt werden soll (`Reg.jsp`)
- Der Transport von Daten
 - erfolgt in der Session
 - mit Hilfe von Java Beans
→ wir brauchen eine Message-Bean (class `MsgBean`)

O4S Portal

- Aufbau der Messages in der Reg.jsp:

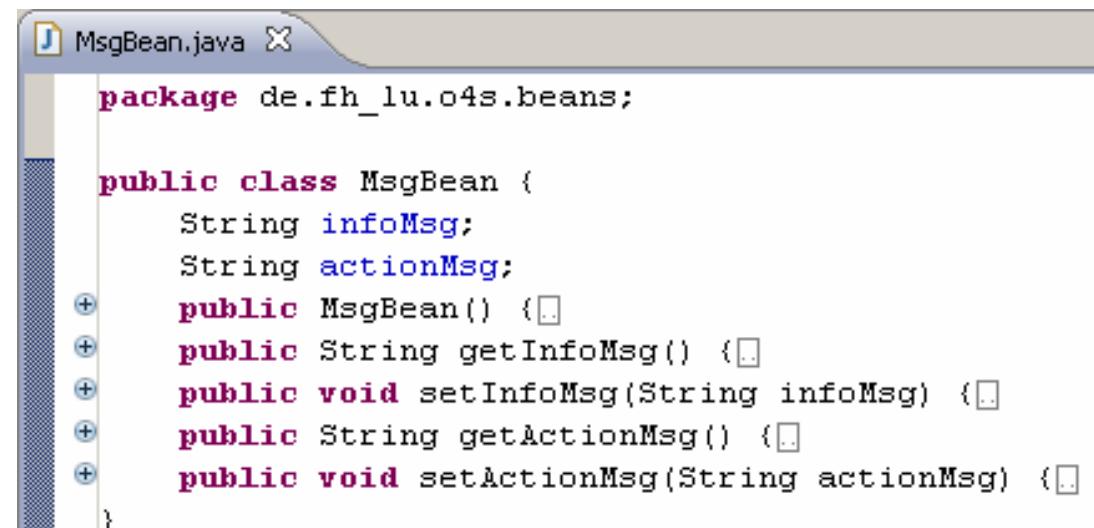
```
<h2>Willkommen zur O4S-Registrierung</h2>
<h4>Bitte registrieren Sie sich</h4>
```

- Bei uns besteht eine Meldung aus zwei Zeilen:
 - Eine Informationszeile, in der dem Benutzer etwas mitgeteilt wird, z.B. der aktuelle Stand oder der Erfolg der letzten Aktion
 - Eine Aktionszeile, in der der Benutzer aufgefordert wird, etwas zu tun.
- Entsprechend muss auch die MsgBean aufgebaut sein.



MsgBean

- Aktion: Legen Sie eine Java Bean namens `MsgBean` an:
 - Im Package `de.fh_lu.o4s.beans`.
 - Mit zwei String-Attributten: `infoMsg` und `actionMsg` (sonst keine Daten)
- Durchführung:
 - Klasse `MsgBean` mit den genannten Attributen anlegen
 - Da `MsgBean` eine Java Bean sein soll, brauchen wir noch einen Konstruktor ohne Parameter und `get-` und `set-` Methoden für beide Attribute.



```
J MsgBean.java X
package de.fh_lu.o4s.beans;

public class MsgBean {
    String infoMsg;
    String actionMsg;
    public MsgBean() { }
    public String getInfoMsg() { }
    public void setInfoMsg(String infoMsg) { }
    public String getActionMsg() { }
    public void setActionMsg(String actionMsg) { }
}
```

- Da wir die Message
 - in RegAppl.jsp erzeugen und in Reg.jsp lesen wollen,
 - müssen wir die MsgBean in beide Seiten einbinden.
- Aktion: Sorgen Sie – analog zur UserBean – dafür, dass
 - in die session unter dem Namen "msg" eine MsgBean (ein Objekt vom Typ MsgBean) eingebunden wird,
 - die in Reg.jsp unter RegAppl.jsp jeweils unter einer Variable msg verfügbar sein soll.
- Schritt 1 – Reg.jsp:
 - Wir schauen nach wie wir das bei der UserBean gemacht haben:

```
<%@page import="de.fh_lu.o4s.beans.UserBean"%>
...
UserBean user = (UserBean) session.getAttribute("user");
if (user == null) {
    user = new UserBean();
    session.setAttribute("user", user);
}
```

- Schritt 2: Wir kopieren den UserBean-Code und
 - ersetzen UserBean durch MsgBean,
 - ersetzen "user" durch "msg",
 - ersetzen user durch msg und
 - fügen oben eine import-Zeile für die MsgBean hinzu: Analog zur UserBean, z.B. wie früher mit Eclipse-AutoVervollständigen.

```
<%@page import="de.fh_lu.o4s.beans.UserBean"%>
<%@page import="de.fh_lu.o4s.beans.MsgBean"%>
...
UserBean user = (UserBean) session.getAttribute("user");
if (user == null){
    user = new UserBean();
    session.setAttribute("user", user);
}
MsgBean msg = (MsgBean) session.getAttribute("msg");
if (msg == null){
    msg = new MsgBean();
    session.setAttribute("msg", msg);
}
```

- Schritt 3: Wir tun dasselbe in RegAppl.jsp.

O4S Portal

- Aktion: Ergänzen Sie Reg.jsp, RegAppl.jsp und MsgBean,
 - so dass Sie für den Fall einer frisch gestarteten Anwendung und
 - für jeden möglichen Ausgang der RegAppl.jsp
 - jeweils eine gute Meldung erzeugen und in der Reg.jsp anzeigen.
- Schritt 1: Wir identifizieren die folgenden Fälle:

Fall	Meldung (infoMsg / actionMsg)
ganz neue Bean	Willkommen zur O4S Registrierung Bitte registrieren Sie sich
Schreiben erfolgreich	Sie wurden erfolgreich registriert Bitte melden Sie sich am O4S Portal an
Administrator geschrieben	Sie wurden als Administrator registriert Bitte melden Sie sich am O4S Portal an
Datenbankfehler	Datenbankfehler Bitte wenden Sie sich an Ihren Systembetreuer
sonstiger Fehler	Sonstiger Fehler Bitte wenden Sie sich an Ihren Systembetreuer

O4S Portal

- Schritt 2: Für jeden der identifizierten Fälle legen wir in MsgBean die entsprechende Methode an:

```
public void setNewBean() {
    infoMsg = "Willkommen zur O4S Registrierung";
    actionMsg = "Bitte registrieren Sie sich";
}

public void setRegSuccess() {
    infoMsg = "Sie wurden erfolgreich registriert";
    actionMsg = "Bitte melden Sie sich am O4S Portal an";
}

public void setRegAdmin() {
    infoMsg = "Sie wurden als Administrator registriert";
    actionMsg = "Bitte melden Sie sich am O4S Portal an";
}

public void setDBkaputt() {
    infoMsg = "Datenbankfehler";
    actionMsg = "Bitte wenden Sie sich an Ihren Systembetreuer";
}

public void setAnyError() {
    infoMsg = "Sonstiger Fehler";
    actionMsg = "Bitte wenden Sie sich an Ihren Systembetreuer";
}
```

O4S Portal

- Schritt 3: Wir ergänzen die RegAppl.jsp, so dass wir
 - der MsgBean für jeden der Fälle die richtige Message zuweisen

```
// Aktion ausführen und auswerten
int rc = user.insertUpdateUser();
if (rc == 0 || rc == 1){
    // Schreiben erfolgreich
    msg.setRegSuccess();
} else if (rc == 2){
    // Administrator geschrieben
    msg.setRegAdmin();
} else if (rc == 98){
    // Datenbankfehler
    msg.setDBkaputt();
} else{
    // sonstiger Fehler
    msg.setAnyError();
}
response.sendRedirect("./Reg.jsp");
```

O4S Portal

- Schritt 4: Wir ergänzen den Standardkonstruktor von MsgBean,
 - so dass auch eine neu erzeugte MsgBean die richtige Message hat:

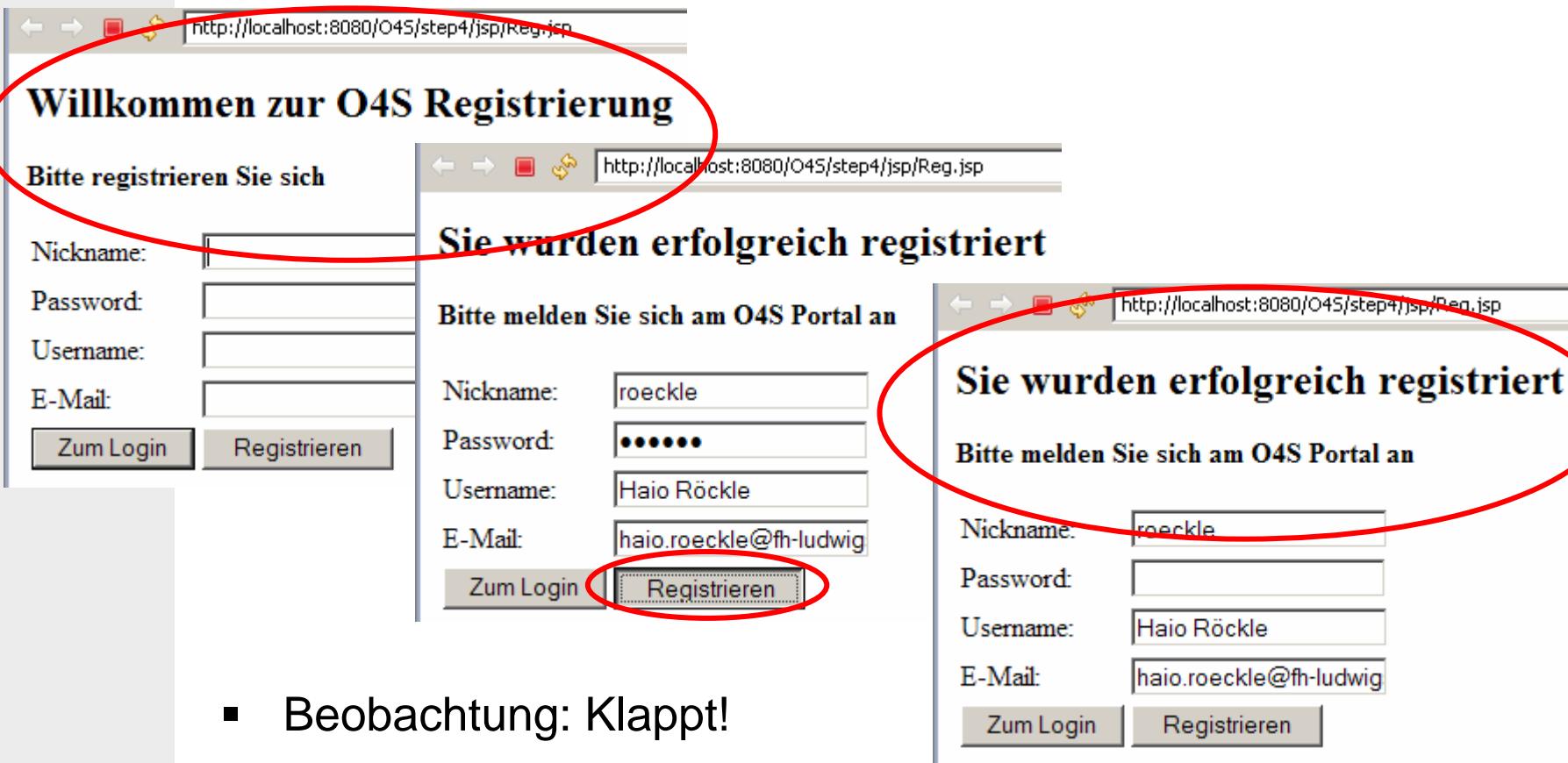
```
public MsgBean() {  
    super();  
    this.setNewBean();  
}
```
- Beobachtung: Bisher steht in der Reg.jsp noch statisch drin:

```
<h2>Willkommen zur O4S-Registrierung</h2>  
<h4>Bitte registrieren Sie sich</h4>
```
- Schritt 5: Wir passen die Reg.jsp an, so dass dynamisch
 - die Meldung aus der MsgBean angezeigt wird.

```
<h2><%= msg.getInfoMsg() %></h2>  
<h4><%= msg.getActionMsg() %></h4>
```
- Beobachtung:
 - Das geht genau wie bei den value-Attributen der input-Felder.

O4S Portal

- Schritt 6: Testen
 - Server löschen, Workspace resetten, Reg.jsp neu starten (unter Tomcat)



The figure consists of three screenshots of a web application interface. The first screenshot shows a registration form titled "Willkommen zur O4S Registrierung". It contains fields for Nickname, Password, Username, and E-Mail, along with "Zum Login" and "Registrieren" buttons. A red oval highlights the entire top section of the first screenshot. The second screenshot shows a success message "Sie wurden erfolgreich registriert" and a login form below it. The third screenshot shows a login form with the same fields and buttons as the first, but the "Registrieren" button is highlighted with a red oval.

Willkommen zur O4S Registrierung

Bitte registrieren Sie sich

Nickname:

Password:

Username:

E-Mail:

Zum Login Registrieren

Sie wurden erfolgreich registriert

Bitte melden Sie sich am O4S Portal an

Nickname: roeckle

Password:

Username: Haio Röckle

E-Mail: haio.roeckle@fh-ludwig

Zum Login Registrieren

Sie wurden erfolgreich registriert

Bitte melden Sie sich am O4S Portal an

Nickname: roeckle

Password:

Username: Haio Röckle

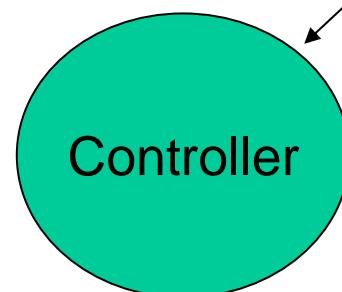
E-Mail: haio.roeckle@fh-ludwig

Zum Login Registrieren

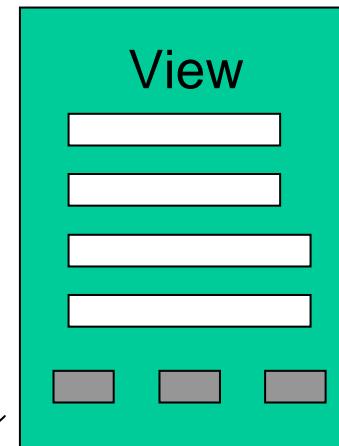
- Beobachtung: Klappt!

Rückschau

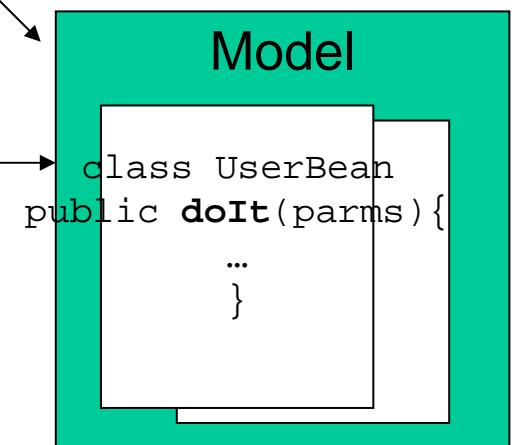
Die View
über gibt die
Eingaben
(Daten und
Aktionen) an
den Controller



Der Controller
instanziert die
Modelklassen, ruft die
richtigen Funktionen
auf, über gibt die
Parameter und steuert
den Screenflow



Die View holt die
Ergebnisse aus
dem Model:
`<% = msg.getInfoMsg() %>`



→→→ Übung JSP 5: Urlaubsziele MVC

Veranstaltung „Web-Anwendungen“

HS Ludwigshafen, Prof. Dr. H. Röckle,
Teil 2: Server-seitige Webanwendungen mit JSP

1	Vorbemerkungen	4
2	Webserver und http	5
2.1	Theorie	5
2.1.1	Begriffe.....	5
2.1.2	http-Übersicht.....	6
2.1.3	http und TCP/IP.....	7
2.2	Praxis.....	7
2.2.1	Beispiel: Der „Apache-Webserver“	7
2.2.2	Wireshark	10
2.2.3	Übertragung von Parametern im Request.....	10
2.2.4	http-Header ansehen	12
3	Wiederholung von Datenbank-Grundlagen.....	14
3.1	Theorie	14
3.1.1	PostgreSQL	14
3.1.2	Datenbankzugriff aus Applikationen und Objekten.....	14
3.1.3	Die Tabelle „user“	14
3.2	Praxis.....	15
3.2.1	Einrichtung JDBC-Treiber für PostgreSQL und JDBCAccess-Framework	15
3.2.2	Anlegen der Tabelle „user“ mit PGAdmin.....	16
3.2.3	AppInstallAccountTable	16
4	Java Server Pages – Hintergründe und Voraarbeiten.....	20
4.1	Grundlagen: Dynamische Webseiten	20
4.2	Praxis: TomCat in Eclipse einrichten.....	21
5	Java Server Pages – Getting Started.....	23
5.1	Grundlagen	23
5.1.1	Statische und dynamische Teile einer Webseite	23
5.1.2	Kontrollstrukturen	26
5.1.3	Ergänzungen.....	27
5.1.4	Verbindung zwischen JSP und http.....	28
5.2	Praktische Anwendung: RegView.jsp und RegAppl.jsp	30
5.2.1	RegView.jsp und Parameter	30
5.2.2	RegAppl.jsp und Auswertung der Request-Parameter.....	31
5.2.3	Auswertung der angefragten Aktion	32
5.2.4	Weiterleitung auf sichtbare JSP	33
6	MVC und Java Beans	34
6.1	Theorie	34
6.1.1	Schichtenmodelle	34
6.1.2	MVC – Model-View-Controller.....	35
6.1.3	MVC für Webanwendungen	35
6.1.4	Java-Beans.....	36
6.1.5	Ausprägung von MVC in dieser Veranstaltung	36
6.2	Praxis: UserBean und Datenbankzugriffe	36
6.2.1	Einstieg.....	37

6.2.2	Datenbank-Connection	37
6.2.3	Speichern eines Users mit insertUserNoCheck()	38
6.2.4	Testen der insertUserNoCheck()	40
6.2.5	Die Methode checkUserExists().....	41
6.2.6	Die Hilfsmethode prepareAttributesForDB().....	42
6.2.7	Speichern eines Users mit insertUserIfNotExists()	43
6.2.8	Testen der insertUserIfNotExists().....	44
6.3	Praxis: Benutzerregistrierung in der RegAppl.jsp.....	44
6.3.1	Bisheriger Stand	44
6.3.2	Registrieraktion nach MVC.....	45
6.3.3	Test und Reflektion	46
7	Sessions	47
7.1	Theorie	47
7.2	Praxis: Wiederanzeige der eingegebenen Benutzerdaten.....	47
7.2.1	Übersicht	47
7.2.2	Datenspeicherung in der RegAppl.jsp.....	48
7.2.3	Daten lesen in der RegView.jsp	48
7.2.4	Daten anzeigen in der RegView.jsp	49
7.2.5	Code verschönern in der RegAppl.jsp.....	49
7.2.6	Testen	51
7.3	Praxis: Einführung einer Messaging-Infrastruktur.....	51
7.3.1	Vorüberlegungen und MVC	51
7.3.2	Die MessageBean	52
7.3.3	Erste Meldung	52
7.3.4	Weitere Meldungen	53
7.3.5	Anpassungen in der RegAppl.jsp	54
7.3.6	Reflektion und Ausblick.....	55
8	JSP-Tags, -Direktiven, -Scopes und -Objekte.....	56
8.1	Theorie	56
8.1.1	JSP-Direktiven	56
8.1.2	JSP-Scopes	56
8.1.3	JSP-Objekte.....	57
8.1.4	JSP-Tags.....	57
8.2	Praxis: Nutzung von JSP-Tags.....	59
8.2.1	Nutzung von jsp:useBean	59
8.2.2	Nutzung von jsp:getProperty.....	60
8.2.3	Oberflächen-Kosmetik	61
9	JSP-Architektur und -Deklarationen	63
9.1	Theorie	63
9.1.1	JSP als Java Klasse.....	63
9.1.2	Deployment und Ausführung der JSP im TomCat.....	64
9.1.3	JSP-Deklarationen.....	64
9.1.4	Beispiel Zähler	64
9.1.5	Zusammenfassung	65
9.2	Praxis: denullify().....	65
9.2.1	Die Methode	65
9.2.2	Einsatz der Methode.....	66
9.2.3	Reflektion und Erweiterung	66
10	Anmeldeseite und Portal	67
10.1	Theorie: Übersicht.....	67
10.1.1	Benutzersicht / Oberflächen	67

10.1.2	Aufbau in Komponenten	67
10.2	Praxis: Login- und Portal-Komponente	69
10.2.1	Login-Komponente	69
10.2.2	Messages in den weiteren Zweigen der RegAppl.jsp.....	73
10.2.3	Messages und Sprünge zwischen zwei Apps	74
10.2.4	Portal-Komponente	75
10.2.5	Testen	78
10.2.6	Beheben der Authentisierungslücken	79
11	Anhang	82
11.1	Anhang 1: Kochrezept zur Entwicklung einer Webanwendung	82
11.2	Anhang 2: Systemeinrichtung	82

1 Vorbemerkungen

Dieses Skript ist nur ein Teil der Vorlesungsunterlagen der Veranstaltung Anwendungssysteme 2: Web-Anwendungen im Bachelorstudiengang Wirtschaftsinformatik der Hochschule Ludwigshafen. Insbesondere sind Einordnung, Lernziele und Literatur im ersten Teil des Skripts zu finden.

Trotzdem möchten wir auch an dieser Stelle noch einige Vorbemerkungen machen:

- Es geht in diesem Skript um die Server-seitige Softwareentwicklung mit Java Server Pages (JSP)
- Lernziel dieser Veranstaltung ist es,
 - Grundideen und Lösungsansätze für eine gute, modulare, wartbare Architektur darzustellen, die im Wesentlichen auf dem MVC-Paradigma aufbaut,
 - Einfache Webanwendungen unter Eclipse mit JSP auf der Basis von TomCat mit der genannten MVC-basierten Architektur selbst entwickeln zu können.
- In der Praxis werden meistens Frameworks verwendet, die viele der hier gelösten Aufgaben vor dem Entwickler verbergen. Trotzdem sind wir der Ansicht, dass die Kenntnis der Aufgaben und Lösungen auch bei der Nutzung von Frameworks von großem Nutzen ist.

Zur Benutzung innerhalb dieses JSP Teils benötigen wir (oder müssen wir uns selbst bauen bzw. besorgen):

- Als Vorlage
 - Eine html-Datei RegView.html mit Überschriften und Formular
 - Eine Java-Klasse UserBeanEmpty als Vorlage
- Zum Einstieg in die Datenbankfunktionalität
 - Das JDBCAccess-Framework mit PostgreSQLAccess und Postgres-JDBC-Treiber
 - Die Java-Klasse AppInstallAccountTable
- Zum Einbinden
 - Das Logo der HS Lu

Weitere Informationen zum Aufbau des Systems, das wir nutzen werden, finden sich im Anhang dieses Skripts.

2 Webserver und http

2.1 Theorie

2.1.1 Begriffe

Aus der Veranstaltung „Rechnernetze“ sollten bereits die Grundbegriffe „Webserver“ und „http“ bekannt sein:

- Ein **Webserver** ist ein (Server-)Dienst, der zum Zugriff durch Webclients bereitsteht.
- **http** ist das Protokoll, also die Sprache, die zwischen Webclients und Webserver genutzt wird.
- Eine (statische) **Webseite** ist eine HTML-Seite, die auf einem Webserver zum Abruf zur Verfügung gestellt wird.
- Eine **Website** (engl. „site“ = Ort, Stelle) oder **Webpräsenz** fasst mehrere zusammengehörige Webseiten zusammen.
- Ein Webserver kann auch als **http-Server** oder **Internetserver** oder – wenn der Zugriff räumlich eingeschränkt ist – als **Intranetserver** bezeichnet werden.

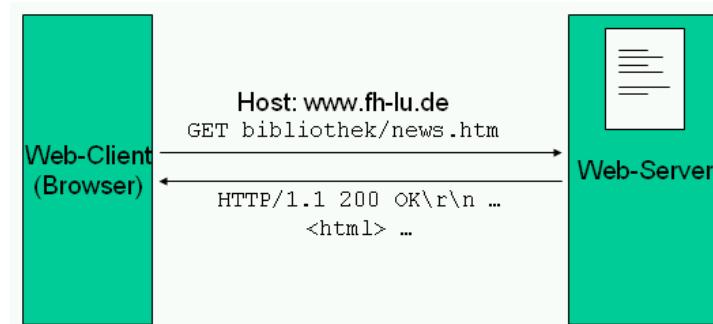
Das http-Protokoll ist vom Prinzip her sehr einfach:

- Der Client sendet eine Anfrage, engl. „**Request**“, mit der er eine html-Seite anfordert.
- Der Server sendet eine Antwort, engl. „**Response**“, in der die angeforderte html-Seite enthalten ist.



Anmerkungen:

- Ein Webserver ist damit eine Software, die auf Anfrage HTML-Seiten zurückliefert. Diese Software kann als Dienst auf jedem Rechner installiert werden, z.B. für Übungszwecke auf Ihren Laptops oder Heim-PCs.
- Für viel genutzte Internetpräsenzen wird natürlich exclusive Hardware verwendet, oft viele Server für eine Webpräsenz. In dem Fall wird der Begriff Webserver nicht nur für die Software sondern auch für die Hardware verwendet.
- Bei den meisten http-Requests handelt es sich um „GET“- oder „POST“-Requests, vgl. 0.



- Die Seiten, die ein Webserver ausliefert, sind in der Regel in einem speziellen Verzeichnisbaum des Webservers gespeichert, vgl. 2.2.1.

Sie sollten im Laufe Ihres Studiums irgendwann einmal die Installation eines eigenen Webservers durchgeführt haben, z.B. auf Ihrem Laptop oder Heim-PC. Die Vorstellung, dass ein Laptop ein Server sein könnte, ist eventuell zunächst verwirrend aber nicht schwer zu verstehen, wenn der Begriff „Server“ als „Server-Software“ interpretiert wird.

In 2.2.1 werden wir deshalb die Installation eines Webservers, in unserem Fall des Apache Webservers, Schritt für Schritt durchgehen, auch wenn dieser für diese Veranstaltung im Anschluss nicht mehr gebraucht wird.

2.1.2 http-Übersicht

http, das hypertext transfer protocol, baut auf den Netzwerkprotokollen TCP und IP auf und beschreibt, wie Requests und Responses zwischen Webclient und Webserver konkret aussehen. Seit 2015 liegt es in Version http 2.0 vor.

Bei Requests und Responses handelt es sich um http-Netzwerkpakete, die immer ähnlich aufgebaut sind, nämlich aus 3 Blöcken:



Die Unterschiede liegen innerhalb der einzelnen Blöcke. Einige davon beschreiben wir im Folgenden. Header-Zeilen enthalten im Wesentlichen technische Informationen, die wir in 2.2.1 betrachten.

Requests

Bei Requests enthält die „erste http-Zeile“ die Anfragemethode und den Pfad der angefragten Datei. Relevante Anfragemethoden sind für uns GET oder POST, die beide dem Abruf von Webseiten dienen, sich aber in Details unterscheiden.

Weitere Anfragemethoden sind

- OPTIONS – technische Informationen des Servers anfordern
- HEAD – Wie GET aber ohne Body
- PUT – Datei hochladen
- DELETE – Datei löschen
- TRACE – Liefert die Anfrage zurück (zum Debuggen)
- CONNECT – für SSL

GET-Request

Bei einem GET-Request sieht die erste Zeile des http-Pakets etwa so aus:

- GET benutzer/bearbeiten.html?userid=schmidt&passwort=geheim&... HTTP/1.1

Parameter werden in der GET-Zeile mit übertragen, dafür ist der Body des GET-Requests in der Regel leer. Da die Länge der ersten http-Zeile beschränkt ist, eignet sich GET nur, wenn die Gesamtlänge der Parameter nicht zu lang ist.

POST-Request

Bei einem POST-Request stehen in der ersten Zeile keine Parameter:

- POST benutzer/bearbeiten.html HTTP/1.1

Parameter werden dann im Body des POST-Requests übertragen, im Wesentlichen ohne Längenbeschränkung:

```
userid=schmidt  
passwort=geheim  
...=...
```

Response

Die erste Zeile einer http-Response enthält einen Statuscode und eine Statusmeldung, z.B.

- HTTP/1.1 200 OK

Statuscode und Statusmeldung enthalten dieselbe Information, z.B. bedeutet der Statuscode 200 immer „OK“. Weitere Statuscodes sind

- 100 – 199: Informationsmeldungen
- 200 – 299: Erfolgsmeldungen
- 300 – 399: Die Anfrage wurde weitergeleitet
- 400 – 499: Client-Fehler, z.B.
 - 404: Dokument nicht vorhanden
 - 403: Keine Berechtigung für das angeforderte Dokument
- 500 – 599: Server-Fehler

Der Body der Response enthält in der Regel die angeforderte html-Seite.

Zusammenfassung

	GET-Request	POST-Request	Response
erste http-Zeile	URL der abgerufenen Seite und evtl. Parameter	URL der abgerufenen Seite	Statuscode und Statusmeldung
http-Header	Vgl. 2.2.1	Vgl. 2.2.1	Vgl. 2.2.1
http-Body	leer	Parameter, falls vorhanden	html-Text der abgerufenen Seite

2.1.3 http und TCP/IP

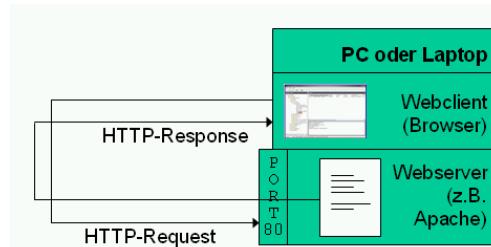
http-Pakete werden über das Netzwerkprotokoll TCP/IP übertragen. Da die Größe von TCP/IP-Netzwerkpaketen beschränkt ist, kann es sein, dass ein http-Paket größer ist als ein TCP/IP-Paket, z.B: wenn in einer http-Response eine große html-Seite übertragen wird.

In dem Fall wird das http-Paket auf mehrere TCP/IP-Pakete aufgeteilt. Nur das erste TCP/IP-Paket enthält dann den http-Header. Dieser http-Header kann deshalb nicht mehr geändert werden, wenn das erste TCP/IP-Paket eines http-Pakets versendet wurde.

2.2 Praxis

2.2.1 Beispiel: Der „Apache-Webserver“

Sie sollten im Laufe Ihres Studiums irgendwann einmal einen eigenen Webserver installiert haben, z.B. auf Ihrem Laptop oder Heim-PC. Die Vorstellung, dass ein Laptop ein Server sein könnte, ist eventuell zunächst verwirrend aber nicht schwer zu verstehen, wenn der Begriff „Server“ als „Server-Software“ interpretiert wird:



Wir gehen nun die Installation eines Webservers anhand des Apache-Webservers durch.

Die Apache Software Foundation

Die Apache Software Foundation ist eine Plattform und Community, für eine große Zahl von Open Source Projekten (<http://apache.org/>).

Das Apache http-Server Projekt

Eines der wichtigsten Projekte ist das Apache http-Serverprojekt (<http://httpd.apache.org/>). httpd steht für http-daemon, wobei daemon in der UNIX / Linux-Welt für Dienst oder Service steht.

Herunterladen

Über die genannte Webseite lassen sich Quellcodes des Apache httpd-Servers herunterladen. Aufgrund des großen Downloadaufkommens sind die Quellen vielfach gespiegelt. Installationspakete müssen von Seiten Dritter heruntergeladen werden, die auf der Apache-Seite verlinkt sind.

Für Windows waren dies zur Entstehungszeit dieses Manuskripts z.B. die Seiten <http://www.apachehaus.com/cgi-bin/download.plx> oder <http://www.apachelounge.com/download/>. Für Apache 2.2 standen noch vollständige msi-Installer zur Verfügung,

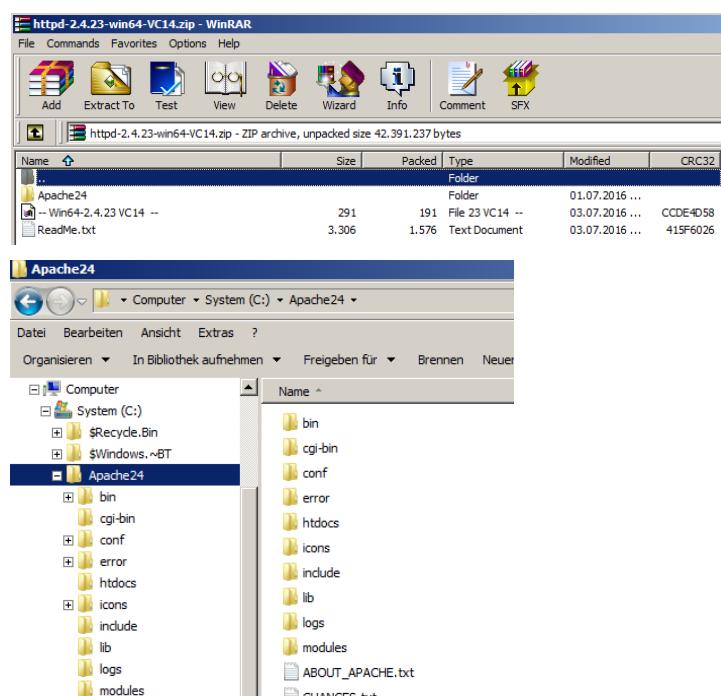
- für Apache 2.4 gibt es nur noch ZIP-Dateien,
- außerdem wird auch die Visual C++ Runtime benötigt.

Beide können auf den genannten Seiten heruntergeladen werden, die Dateinamen sind z.B. httpd-2.4.23-win64-VC14.zip bzw. vc_redist.x64.exe.

Installation

Zunächst wird die Visual C++ Runtime mit der Installationsdatei installiert. Hierzu sind keine spezifischen Konfigurationen nötig.

Anschließend wird die ZIP-Datei geöffnet und das Verzeichnis Apache24 wird nach C:\Apache24 entpackt.



Grundsätzlich wäre natürlich auch ein anderes Verzeichnis für das Entpacken denkbar, der Verzeichnisname sollte aber keine Leerzeichen enthalten, weil dies gelegentlich zu unerwartetem Verhalten führt.

Ausführung

Der Server wird ausgeführt, indem die folgende Datei gestartet wird:

- C:\Apache24\bin\httpd.exe

Eventuell müssen Firewall Freigaben gesetzt oder bestätigt werden. Da der Server im Hintergrund auf Requests wartet, besitzt er keine Benutzeroberfläche. Der Server ist also nur im Prozessmanager sichtbar.

Unter Windows kann der Server auch als Dienst gestartet bzw. installiert werden. Dafür muss in einer Eingabeaufforderung mit Administratorberechtigungen der folgende Befehl ausgeführt werden:

- C:\Apache24\bin\httpd.exe -k install

In diesem Fall ist der httpd auch unter den Windows-Diensten sichtbar und kann mit dem Servicemanager beendet und gestartet werden.

Um die Installation als Dienst rückgängig zu machen, wird verwendet:

- C:\Apache24\bin\httpd.exe -k uninstall

Wenn httpd als Dienst installiert ist, kann dieser auch mit dem Apache Service Monitor gestartet und beendet werden. Der Apache Service Monitor selbst wird gestartet mit

- C:\Apache24\bin\ApacheMonitor.exe

Anzuzeigende Dateien

Der Apache Webserver liefert in den Grundeinstellungen nur Dateien aus einem bestimmten Verzeichnis und dessen Unterverzeichnissen aus. In der Standardeinstellung ist dies das Verzeichnis

- C:\Apache24\bin\htdocs

Für eine benutzerdefinierte Installation könnte dies geändert werden, z.B. zu

- D:\Daten\web\public

Es kann auch das html-Verzeichnis unseres Eclipse-Workspaces angegeben werden. Wie die Änderung durchgeführt wird, wird unten im Abschnitt „Konfiguration“ beschrieben.

Test

Für den Test wird ein Webclient verwendet, also in Browser. In der Adresszeile ist anzugeben, dass auf den eigenen Rechner zugegriffen werden soll. Dazu dient die Host-Angabe „localhost“ oder gleichbedeutend „127.0.0.1“.

Wenn keine Datei angegeben wird, sucht der Webserver standardmäßig nach einer Datei index.html und liefert diese zurück. Die meisten Standardinstallationen besitzen eine Default-Datei mit dem Namen index.html, die mindestens die Wörter „it works“ anzeigt.

Konfiguration

Der Apache Webserver wird gesteuert mit Hilfe der Konfigurationsdatei

- C:\Apache24\conf\httpd.conf

Dies ist eine reine Textdatei, die per Editor geändert werden kann. Zeilen, die mit # beginnen, sind Kommentarzeilen. **Nach jeder Änderung muss der Server neu gestartet werden, damit die Änderung wirksam wird.**

Typische Anpassung ist u.a. die Änderung des Verzeichnisses der anzuzeigenden Daten:

```
# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this directory, but
# symbolic links and aliases may be used to point to other locations.
#
DocumentRoot "${SRVROOT}/htdocs"
<Directory "${SRVROOT}/htdocs">
```

Dies kann geändert werden, z.B. zu

```
# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this directory, but
# symbolic links and aliases may be used to point to other locations.
#
# DocumentRoot "${SRVROOT}/htdocs"
```

```
# <Directory "${SRVROOT}/htdocs">
DocumentRoot "D:\Daten\web\public"
<Directory "D:\Daten\web\public">
```

Der Eintrag muss zweimal geändert werden. Der DocumentRoot-Eintrag zeigt auf das Verzeichnis, der Directory-Eintrag vergibt die nötigen Zugriffsrechte.

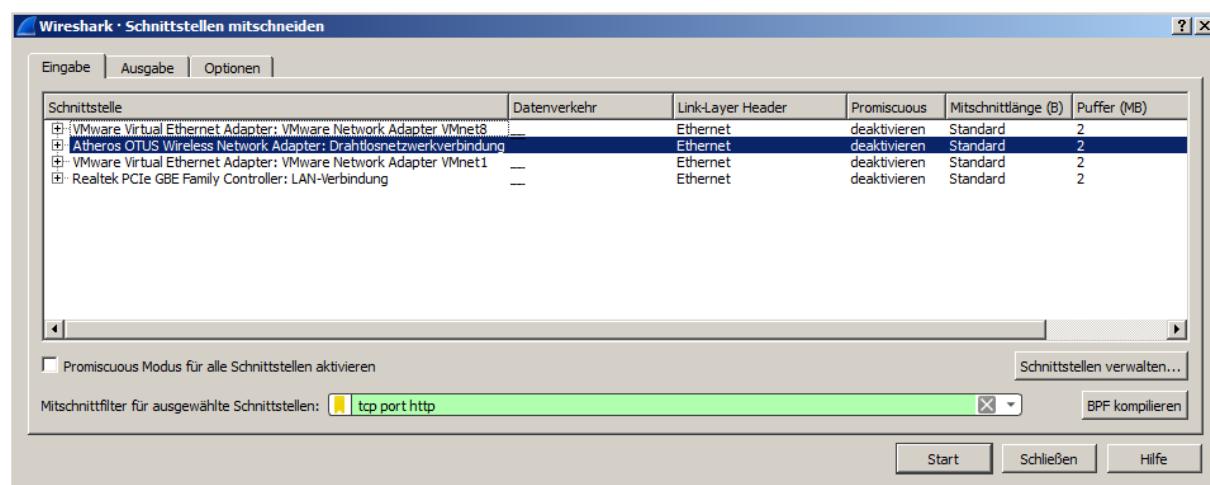
Weitere Konfigurationen sind z.B. der Netzwerkport, auf den der Server hört (standardmäßig Port 80), Detaillierte Zugriffsrechte, Module zur Erweiterung der Serverfunktionalität, z.B. zur Ausführung von PHP-Programmen.

2.2.2 Wireshark

Um uns die http-Header einiger http-Pakete anzusehen, installieren wir Wireshark. Wireshark ist ein Open Source Netzwerküberwachungstool, das von <http://www.wireshark.org> heruntergeladen werden kann. Bei der Installation wird der sogenannte WinPCap-Treiber mit installiert.

Wireshark kann Netzwerkpakete an Netzwerkschnittstellen mitlesen, aufzeichnen und die Aufzeichnung speichern, filtern, analysieren, etc.

Einstieg ist z.B. der Menüpunkt Aufzeichnen -> Optionen und in Version 2.0.5 das folgende Fenster:



Es ist auszuwählen, welche Netzwerkschnittstelle mitgeschnitten werden soll. Bei WLAN-Schnittstellen unter Windows muss meistens der Promicuous Mode abgeschalten werden, weil es sonst Fehler gibt, weil der WLAN Adapter dies meistens nicht unterstützt.

Da im Netz sehr viele Pakete in unterschiedlichen Protokollen übertragen werden, ist es sinnvoll, diese bereits beim Mitlesen zu filtern. Das kann über einen „Mitschnittfilter“ erfolgen.

Zum Test könnte jetzt auf „Start“ geklickt werden, anschließend eine Webseite aufgerufen werden und dann auf „Stop“ geklickt werden. Die übertragenen Netzwerkpakete werden mitgelesen und angezeigt. Wenn auf eines der Pakete geklickt wird, wird der Inhalt des Pakets angezeigt.

2.2.3 Übertragung von Parametern im Request

Aufbauend auf dem html-Formular zur Benutzerregistrierung demonstrieren wir mittels Wireshark die Übertragung von Parametern bei GET- und POST-Requests. Wir rufen die Seite RegView.html auf und tragen sinnvolle Werte ein:

```

<form action=".\\jsp\\RegAppl.jsp" method="get">
  <table>
    <tr>
      <td>Nickname:</td>
      <td><input type="text" name="userid" value="" /></td>
    </tr>
    <tr>
      <td>Passwort:</td>
      <td><input type="password" name="password" value="" /></td>
    </tr>
    <tr>
      <td>Username:</td>
      <td><input type="text" name="username" value="" /></td>
    </tr>
    <tr>
      <td>E-Mail:</td>
      <td><input type="text" name="email" value="" /></td>
    </tr>
    <tr>
      <td><input type="reset" name="reset" value="Zurücksetzen" /></td>
      <td><input type="submit" name="register" value="Registrieren" /></td>
    </tr>
  </table>
</form>

```

Nickname:	<input type="text"/>
Passwort:	<input type="password"/>
Username:	<input type="text"/>
E-Mail:	<input type="text"/>
<input type="button" value="Zurücksetzen"/>	<input type="button" value="Registrieren"/>

Nickname:	Heiner
Passwort:	*****
Username:	Brand
E-Mail:	hb@abc.de
<input type="button" value="Zurücksetzen"/>	<input type="button" value="Registrieren"/>

Bei Klick auf den Button „Registrieren“ werden die folgenden name/value Paare an RegAppl.jsp gesendet:

- userid=Heiner
- password=geheim
- username=Brand
- email=hb@abc.de
- register=Registrieren

Bei einem GET-Request erfolgt das in der ersten http-Zeile in der Form

<http://192.168.179.39/jsp/RegAppl.jsp?userid=Heiner&password=geheim&username=Brand&email=hb@abc.de®ister=Registrieren>

Um dies mittels Wireshark ansehen zu können, müssen wir den Aufruf über eine Netzwerkschnittstelle leiten. Dazu können wir

- Die Dateien RegView.html und RegAppl.jsp in das Verzeichnis der anzugebenden Daten unseres Apache Webservers kopieren.
- Unseren Apache Webserver starten
- Die Firewall so konfigurieren, dass sie remote Zugriff auf den Apache Webserver erlaubt.
- Mit einer Kommandozeile und dem Befehl ipconfig die IP-Adresse unseres Rechners ermitteln

```
C:\Users\roeckle>ipconfig
Windows-IP-Konfiguration

Drahtlos-LAN-Adapter Drahtlosnetzwerkverbindung:
  Verbindungsspezifisches DNS-Suffix: fritz.box
  Verbindungslokale IPv6-Adresse . : fe80::91f8:8a7c:d4f3:e6e9%12
  IPv4-Adresse . . . . . : 192.168.179.39
  Subnetzmaske . . . . . : 255.255.255.0
  Standardgateway . . . . . : 192.168.179.1
```

- Mit einem anderen Rechner per Browser auf die RegView.html zugreifen, die von unserem Apache Webserver ausgeliefert wird, z.B. per <http://192.168.179.39/RegView.html>
- Geeignete Daten eintragen
- Das Capturing von Wireshark einschalten
- Datenabruft durchführen
- Capturing von Wireshark ausschalten
- Request ansehen in Wireshark:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.179.21	192.168.179.39	TCP	66	56524 + 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.010805	192.168.179.39	192.168.179.21	TCP	66	80 + 56524 [SYN, ACK] Seq=1 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 \$
3	0.018171	192.168.179.21	192.168.179.39	TCP	54	56524 + 80 [ACK] Seq=1 Ack=1 Win=65700 Len=0
4	0.011465	192.168.179.21	192.168.179.39	HTTP	504	GET /jsp/RegAppl.jsp?userid=Heiner&password=geheim&username=Brand&email=hb@abc.de®ister=Registrieren HTTP/1.1
5	0.015573	192.168.179.39	192.168.179.21	HTTP	498	HTTP/1.1 404 Not Found (text/html)
6	0.216049	192.168.179.21	192.168.179.39	TCP	54	56524 + 80 [ACK] Seq=451 Ack=445 Win=65256 Len=0

```
Frame 4: 504 bytes on wire (4032 bits), 504 bytes captured (4032 bits) on interface 0
Ethernet II, Src: IntelCor_19:50:fa (7c:7a:91:19:50:fa), Dst: AvmGmbh_02:c8:22 (bc:05:43:02:c8:22)
Internet Protocol Version 4, Src: 192.168.179.21, Dst: 192.168.179.39
Transmission Control Protocol, Src Port: 56524 (56524), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 450
HyperText Transfer Protocol
  GET /jsp/RegAppl.jsp?userid=Heiner&password=geheim&username=Brand&email=hb@abc.de&register=Registrieren HTTP/1.1\r\n
  Host: 192.168.179.39\r\n
  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
  Accept-Language: de,en-US;q=0.7,en;q=0.3\r\n
  Accept-Encoding: gzip, deflate\r\n
  DNT: 1\r\n
  Referer: http://192.168.179.39/RegView.html\r\n
  Connection: keep-alive\r\n
```

Anschließend testen wir die Parameter-Übertragung bei einem POST-Request

- Wir ändern die RegView.html, so dass die <form> als POST-Request abgeschickt wird:

```
<form action="..\\jsp\\RegAppl.jsp" method="post">
```
- Wir kopieren die RegView.html in das Verzeichnis der anzuzeigenden Daten unseres Apache Webservers
- Wir stoppen unseren Apache Webserver-Dienst und starten ihn neu
- Wir greifen wie oben vom remote Rechner auf die RegView.html unseres Apache Webservers zu, indem wir die IP-Adresse angeben. Achtung: Die Seite kann gecached sein. Es ist darauf zu achten, dass sie tatsächlich neu abgerufen wird. Ggf. kann mit „Quelltext ansehen“ geprüft werden, dass jetzt tatsächlich „post“ in der Datei steht.
- Wir tragen wieder Daten ein,
- starten das Capturing unseres Wireshark,
- schicken die Seite ab,
- stoppen das Capturing unseres Wireshark
- und sehen uns den Request an:

No.	Time	Source	Destination	Protocol	Length	Info
71	73.779956	192.168.179.21	192.168.179.39	TCP	54	56597 → 80 [ACK] Seq=430 Ack=1068 Win=64632 Len=0
72	77.884531	192.168.179.21	192.168.179.39	HTTP	573	POST /jsp/RegAppl.jsp HTTP/1.1 (application/x-www-form-u
73	77.887874	192.168.179.39	192.168.179.21	HTTP	497	HTTP/1.1 404 Not Found (text/html)
74	78.084137	192.168.179.21	192.168.179.39	TCP	54	56597 → 80 [ACK] Seq=949 Ack=1511 Win=65700 Len=0
75	83.423391	192.168.179.39	192.168.179.21	TCP	54	80 → 56597 [FIN, ACK] Seq=1511 Ack=949 Win=65024 Len=0
76	83.423440	192.168.179.21	192.168.179.39	TCP	54	56597 → 80 [ACK] Seq=949 Ack=1512 Win=65700 Len=0
77	83.423904	192.168.179.21	192.168.179.39	TCP	54	56597 → 80 [FIN, ACK] Seq=949 Ack=1512 Win=65700 Len=0

⋮

Ethernet II, Src: IntelCor_19:50:fa (7c:7a:91:19:50:fa), Dst: AvmGmbh_02:c8:22 (bc:05:43:02:c8:22)
Internet Protocol Version 4, Src: 192.168.179.21, Dst: 192.168.179.39
Transmission Control Protocol, Src Port: 56597 (56597), Dst Port: 80 (80), Seq: 430, Ack: 1068, Len: 519

HTTP/1.1 404 Not Found

Host: 192.168.179.39\r\nUser-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:47.0) Gecko/20100101 Firefox/47.0\r\n

⋮

Response in frame: 73

HTML Form URL Encoded: application/x-www-form-urlencoded

Form item: "userid" = "Heiner"

Key: userid

Value: Heiner

Form item: "password" = "geheim"

Form item: "username" = "Brand"

Form item: "email" = "hb@abc.de"

01b0 61 70 70 6c 69 63 61 74 69 6f 6e 2f 78 2d 77 77 application/x-ww
01c0 77 2d 66 6f 72 6d 2d 75 72 6c 65 6e 63 6f 64 65 w-form-u rlencode
01d0 64 0d 0a 43 6f 6e 74 65 6e 74 2d 4c 65 6e 67 74 d..Content-Lengt
01e0 68 3a 20 38 34 0d 0a 0d 0a 75 73 65 72 69 64 3d h: 84... .userid=

01f0 48 65 69 6e 65 72 26 70 61 73 73 77 6f 72 64 3d Heiner&password=

0200 67 65 68 65 69 6d 26 75 73 65 72 6e 61 6d 65 3d geheim&username=

0210 42 72 61 6e 64 26 65 6d 61 69 6c 3d 68 62 25 34 Brand&email=hb%4

0220 30 61 62 63 2e 64 65 26 72 65 67 69 73 74 65 72 0abc.de®ister=

0230 3d 52 65 67 69 73 74 72 69 65 72 65 6e Registrieren

Zum Schluss könnten wir die Firewall wieder so konfigurieren, dass der remote Zugriff blockiert wird.

2.2.4 http-Header ansehen

Mit Wireshark können auch die http-Header in den mitgelesenen Netzwerkpaketen angesehen werden.

Die folgende Abbildung zeigt den Request zum Abruf der Seite der Hochschule Ludwigshafen. In der ersten http-Zeile „GET / HTTP/1.1“ besteht die Pfadangabe der abgerufenen Datei nur aus „/“.

No.	Time	Source	Destination	Protocol	Length	Info
10	2.181581	192.168.179.39	143.93.200.17	TCP	66	49302 → 80 [ACK] Seq=1 Ack=
11	2.182086	192.168.179.39	91.10.91.242	TCP	74	49303 → 80 [SYN] Seq=0 Win=
12	2.183528	192.168.179.39	143.93.200.17	HTTP	395	[GET / HTTP/1.1]
13	2.325943	192.168.179.39	143.93.200.17	TCP	74	49304 → 80 [SYN] Seq=0 Win=
14	2.361602	143.93.200.17	192.168.179.39	TCP	74	80 → 49304 [SYN, ACK] Seq=0
15	2.361722	192.168.179.39	143.93.200.17	TCP	66	49304 → 80 [ACK] Seq=1 Ack=
16	2.362387	192.168.179.39	143.93.200.17	HTTP	399	GET / HTTP/1.1
17	2.402507	143.93.200.17	192.168.179.39	TCP	66	80 → 49304 [ACK] Seq=1 Ack=
18	2.432342	192.168.179.39	143.93.200.17	TCP	66	[TCP ACKed unseen segment]

Frame 12: 395 bytes on wire (3160 bits), 395 bytes captured (3160 bits) on interface 0
 Ethernet II, Src: AvmGmbH_02:c8:22 (bc:05:43:02:c8:22), Dst: AvmGmbH_66:47:b1 (bc:05:43:66:47:b1)
 Internet Protocol Version 4, Src: 192.168.179.39, Dst: 143.93.200.17
 Transmission Control Protocol, Src Port: 49302 (49302), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 329
Hypertext Transfer Protocol
 GET / HTTP/1.1
 Host: hs-lu.de
 User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:48.0) Gecko/20100101 Firefox/48.0
 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
 Accept-Language: de,en-US;q=0.7,en;q=0.3
 Accept-Encoding: gzip, deflate
 DNT: 1
 Connection: keep-alive
 Upgrade-Insecure-Requests: 1
 Full request URI: http://hs-lu.de/
 [HTTP request 1/1]
 [Response in frame: 20]

Die folgende Abbildung zeigt eine Response, die die Homepage der HS Ludwigshafen zurückgibt:

No.	Time	Source	Destination	Protocol	Length	Info
16	2.362387	192.168.179.39	143.93.200.17	HTTP	399	GET / HTTP/1.1
20	2.458655	143.93.200.17	192.168.179.39	HTTP	623	HTTP/1.1 301 Moved Permanently (text/html)
50	2.586074	143.93.200.17	192.168.179.39	HTTP	1364	[HTTP/1.1 200 OK (text/html)]
53	2.599178	192.168.179.39	143.93.200.17	HTTP	552	GET /typo3temp/scriptmerger/compressed/head-706dfb36f...
59	2.607294	192.168.179.39	143.93.200.17	HTTP	536	GET /typo3temp/scriptmerger/compressed/head-728551bfb...
60	2.607877	192.168.179.39	143.93.200.17	HTTP	552	GET /typo3temp/scriptmerger/compressed/head-f743dea47...

Frame 50: 1364 bytes on wire (10912 bits), 1364 bytes captured (10912 bits) on interface 0
 Ethernet II, Src: AvmGmbH_66:47:b1 (bc:05:43:66:47:b1), Dst: AvmGmbH_02:c8:22 (bc:05:43:02:c8:22)
 Internet Protocol Version 4, Src: 143.93.200.17, Dst: 192.168.179.39
 Transmission Control Protocol, Src Port: 80 (80), Dst Port: 49304 (49304), Seq: 24481, Ack: 334, Len: 1298
 [18 Reassembled TCP Segments (25778 bytes): #22(1440), #23(1440), #25(1440), #26(1440), #28(1440), #29(1440), #31(1440), #32(1440),
Hypertext Transfer Protocol
 HTTP/1.1 200 OK
 Date: Tue, 30 Aug 2016 07:40:41 GMT
 Server: Apache/2.4.7 (Ubuntu)
 X-Powered-By: PHP/5.5.9-1ubuntu4.16
 Set-Cookie: contrast=0; expires=Sat, 29-Oct-2016 07:40:41 GMT; Max-Age=5184000
 Set-Cookie: contrastAge=1472542841; expires=Sat, 29-Oct-2016 07:40:41 GMT; Max-Age=5184000
 Set-Cookie: fontSize=0; expires=Sat, 29-Oct-2016 07:40:41 GMT; Max-Age=5184000
 Set-Cookie: fontSizeAge=1472542841; expires=Sat, 29-Oct-2016 07:40:41 GMT; Max-Age=5184000
 Set-Cookie: layout=0; expires=Sat, 29-Oct-2016 07:40:41 GMT; Max-Age=5184000
 Set-Cookie: layoutAge=1472542841; expires=Sat, 29-Oct-2016 07:40:41 GMT; Max-Age=5184000
 Cache-Control: private
 Vary: Accept-Encoding
 Content-Encoding: gzip
 Content-Length: 24952
 Keep-Alive: timeout=5, max=100
 Connection: Keep-Alive
 Content-Type: text/html; charset=utf-8
 [HTTP response 1/7]
 [Time since request: 0.223687000 seconds]
 [Request in frame: 16]
 [Next request in frame: 53]
 [Next response in frame: 106]
 Content-encoded entity body (gzip): 24952 bytes -> 161978 bytes
Line-based text data: text/html
<!DOCTYPE html>
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7" lang="de" xmlns="http://www.w3.org/1999/xhtml"> <![endif]-->\n<!--[if IE 7]> <html class="no-js lt-ie9 lt-ie8" lang="de" xmlns="http://www.w3.org/1999/xhtml"> <![endif]-->\n<!--[if IE 8]> <html class="no-js lt-ie9" lang="de" xmlns="http://www.w3.org/1999/xhtml"> <![endif]-->\n<!--[if gt IE 8]> <!--><html class="no-js" lang="de" xmlns="http://www.w3.org/1999/xhtml"> <!-->!-->\n<head>\n</head>\n<meta charset="utf-8">\n<!-- \n<tKonzept, Design und TYPO3 Entwicklung: Brain Appeal GmbH, Mannheim (www.brain-appeal.com)\n<\n<tThis website is powered by TYPO3 - inspiring people to share!\n<tTYPO3 is a free open source Content Management Framework initially created by Kasper Skaarhoj and licensed under GNU/GPL.\n<tTYPO3 is copyright 1998-2016 of Kasper Skaarhoj. Extensions are copyright of their respective owners.\n<tInformation and contribution at http://typo3.org/\n-->\n<\n<base href="http://www.hs-lu.de/">\n<\n<\n<meta name="generator" content="TYPO3 CMS">\n<meta name="description" content="Hochschule Ludwigshafen am Rhein">\n<meta name="author" content="Hochschule Ludwigshafen am Rhein">\n

Außer den http-Headern ist hier auch erkennbar, dass im Body der Response die HTML-Seite übertragen wird.

3 Wiederholung von Datenbank-Grundlagen

3.1 Theorie

3.1.1 PostgreSQL

Aus der Veranstaltung „Datenbanken“ sollte die Datenbank PostgreSQL inclusive Installation, Benutzung, SQL und JDBC-Zugriff bekannt sein.

3.1.2 Datenbankzugriff aus Applikationen und Objekten

Um aus einer Java-Applikation über einen JDBC-Treiber auf die Datenbank zuzugreifen, ist ein Objekt der Klasse `java.sql.Connection`, also ein JDBC-Connection-Objekt erforderlich.

Üblicherweise wird die Erzeugung eines solchen Connection-Objekts möglichst zentral durchgeführt, damit der entsprechende Code nicht mehrmals geschrieben bzw. kopiert werden muss.

In Applikationen ist es deshalb sinnvoll, die Erzeugung des Connection-Objekts bereits in der `main()`-Methode durchzuführen bzw. aufzurufen. Falls die Connection in einem Attribut gespeichert werden soll, ist zu beachten, dass die `main()`-Methode static ist, das Connection-Attribut aber evtl. nicht static sein soll.

In Objekten, die mehrere Datenbank-Methoden besitzen, ist es sinnvoll, die Erzeugung des Connection-Objekts im Konstruktor durchzuführen bzw. aufzurufen und die Connection in einem Attribut zu speichern.

3.1.3 Die Tabelle „user“

Für die Praxisbeispiele dieser Veranstaltung verwenden wir eine Tabelle „user“. JedeR TeilnehmerIn der Veranstaltung legt ihre/seine eigene Tabelle an und verwendet als Schema „s“ + <Matrikelnummer>. EinE StudentIn mit Matrikelnummer 654321 würde demnach das Schema s654321 verwenden. Da diese Matrikelnummer an der HS Lu auf absehbare Zeit noch nicht vorkommt, kann diese vom Dozenten verwendet werden.

Die Tabelle s<MatrNr>.user hat die folgende Gestalt

Attributname	Datentyp	NOT NULL	Schlüssel	Default	Bemerkung
userid	char(16)	X	X		
password	char(32)	X			
active	char(1)	X		,Y'	
admin	char(1)	X		,N'	
username	varchar(256)				
email	varchar(256)				

Zur Erinnerung: Bei dem Datentyp `char(32)` speichert die Datenbank immer 32 Zeichen und füllt ggfs. mit Leerzeichen auf, verschenkt als Speicherplatz. Bei dem Datentyp `varchar(256)` werden nur die benutzten Zeichen gespeichert, allerdings ist auch eine Angabe erforderlich, wie lang der Inhalt tatsächlich ist, es entsteht also zusätzlicher Verwaltungsaufwand. Für eine effiziente (zeilenweise) Speicherung sollen zuerst alle Attribute fester Länge angegeben werden und erst danach die Attribute variabler Länge.

3.2 Praxis

3.2.1 Einrichtung JDBC-Treiber für PostgreSQL und JDBCAccess-Framework

JDBC-Treiber für PostgreSQL

Zum Zugriff auf die PostgreSQL Datenbank mittels JDBC wird ein JDBC-Treiber benötigt. Es handelt sich dabei um ein JAR-File, das für Java 1.8 zur Entstehungszeit dieses Skripts den Dateinamen postgresql-9.4.1208.jar trägt.

Dieses JAR-File wird durch den Postgres-Stackbuilder standardmäßig in einem Verzeichnis pgJDBC installiert. Zur Nutzung unter Eclipse muss es ins Projekt eingebunden werden und zwar einerseits in den Java Build Path und andererseits in die Deployment Assembly:

- Herunterladen der genannten Datei und Ablegen in einem Verzeichnis auf dem lokalen Rechner.
- Rechtsklick auf das Projekt -> Properties -> Java Build Path -> Libraries -> (wenn die Datei noch nicht da ist) Add External JARs... -> Auswahl der genannten Datei
- Rechtsklick auf das Projekt -> Properties -> Deployment Assembly -> (wenn die Datei noch nicht da ist) Add... -> Java Build Path Entries -> Auswahl der genannten Datei

JDBC-Access Framework

Zum Zugriff auf PostgreSQL verwenden wir das JDBC-Framework, das bereits in der Veranstaltung Datenbanken benutzt wurde. Es besteht aus den drei Dateien

- JDBCAccess.java
- PostgreSQLAccess.java
- NoConnectionException.java,

die in einem Package de.hs_lu.bw.jdbc abgelegt werden und im OLAT zur Verfügung stehen sollten.

In der Klasse PostgreSQLAccess ist die Methode setDBParms() folgendermaßen anzupassen:

- dbDrivername = "org.postgresql.Driver"; generell für PostgreSQL
- dbURL = "jdbc:postgresql://143.93.200.243:5432/BWUEBDB"; für die Übungsdatenbank auf dem Server der HS Lu
- dbURL = "jdbc:postgresql://localhost:5432/dbName"; oder
dbURL = "jdbc:postgresql://127.0.0.1:5432/dbName"; für lokalen Datenbankzugriff
- dbUser und dbPassword wie in der Veranstaltung „Datenbanken“
- dbSchema = "s6xxxxx"; anstatt 6xxxxx ist nach dem kleinen ,s' die Matrikelnummer einzusetzen

PostgreSQLAccess besitzt eine main()-Methode, ist also ausführbar. Zum Test kann diese Klasse ausgeführt werden, um zu sehen, ob

- der Treiber geladen wurde
- eine Verbindung hergestellt wurde.

Im Fall eines Fehlschlags wird ein Stack-Trace der entstehenden Exception in der Konsole ausgegeben und die Exception in eine NoConnectionException umgewandelt.

Nutzung des JDBC-Access Framework

Das JDBCAccess-Framework kann genutzt werden, um eine JDBC-Connection zu erzeugen. Die Zeile `Connection dbConn = new PostgreSQLAccess().getConnection()` erzeugt ein JDBC-Connection-Objekt, also ein Objekt der Klasse `java.sql.Connection`.

Im Fehlerfall wird eine `NoConnectionException` geworfen.

3.2.2 Anlegen der Tabelle „user“ mit PGAdmin

Zunächst verwenden wir pgAdmin, um die Tabelle s<MatrNr>.user anzulegen und wieder zu löschen. Die funktionierenden SQL-Statements verwenden wir dann im nächsten Kapitel. Wir finden pgAdmin in unserer PostgreSQL-Installation unter

- \bin\pgAdmin3.exe.

In pgAdmin verwenden wir den SQL-Editor mit dem folgenden Statement:

```
create table s654321.user(
    userid      char(16)      not null primary key ,
    password    char(32)      not null           ,
    active      char(1)       not null default 'Y' ,
    admin       char(1)       not null default 'N' ,
    username   varchar(256)  not null           ,
    email       varchar(256)  not null           )
```

Für den Fall, dass sich ein Fehler einschleicht, ist es auch sinnvoll, das gegenteilige Statement vorrätig zu haben: `drop table s654321.user`

Nach mehrmaligem Experimentieren mit diesen beiden Statements sollten wir uns darauf verlassen, dass der SQL-Code funktioniert. Danach können wir diesen auch aus JDBC heraus verwenden.

3.2.3 AppInstallAccountTable

Wir verwenden eine Java-Anwendung AppInstallAccountTable, um die Account-Tabelle anzulegen, zu füllen und ggfs. wieder zu löschen. Sie finden die Anwendung im OLAT.

Als Wiederholung der JDBC-Techniken werden wir auch eine Methode bereitstellen, um die Daten der Tabelle account zu lesen und in der Konsole anzuzeigen.

Methode main()

Um die Methoden zum anlegen, füllen und löschen in nicht-statischer Form entwickeln zu können, müssen wir zunächst in der main-Methode ein Anwendungsobjekt erzeugen.

Um nicht für jede Methode der Anwendung eine eigene Datenbankverbindung öffnen zu müssen, benötigen wir außerdem ein Attribut vom Typ `java.sql.Connection`, dem wir ebenfalls in der main-Methode bereits eine offene Datenbankverbindung zuweisen.

Schließlich rufen wir die erste nicht-statische Methode der Anwendung auf, die die weitere Ausführung steuert:

```
public class AppInstallAccountTable {

    public Connection dbConn = null;

    public static void main(String[] args) throws SQLException {
        AppInstallAccountTable myApp = new AppInstallAccountTable();
        myApp.dbConn = new PostgreSQLAccess().getConnection();
        myApp.installAccountTable();
    }

    public void installAccountTable() throws SQLException {
        System.out.println("installAccountTable() gestartet");
//        this.dropAccountTable();
//        this.createAccountTable();
//        this.insertFirstUser();
//        this.insertSecondUser();
    }
}
```

Methode installAccountTable()

Die Methode `installAccountTable()` ruft die weiteren Methoden auf. Die Aufrufe können einzeln aus- und einkommentiert werden, so dass diese je nach Zustand der Datenbank selektiv ausgeführt werden können.

Beim ersten Aufruf der `insert`-Methode sollte z.B. die `drop`-Methode auskommentiert bleiben. Sobald die Tabelle vorhanden ist, muss für einen erneuten `insert` zunächst der `drop` ausgeführt werden.

Methode createAccountTable()

Die Methode createAccountTable() erzeugt die Tabelle account analog zu 3.2.2.

Es wird dasselbe SQL-Statement verwendet, das in Eclipse bzw. Java zunächst in einem String zu speichern ist. Da der String relativ lang ist, ist es aus Lesbarkeitsgründen sinnvoll, diesen in mehreren Zeilen zuzuweisen. **Achtung:** Ein häufiger Fehler besteht darin, dass im SQL-String Leerzeichen erforderlich sind, die durch die mehrzeilige Zuweisung verloren gehen.

Der String sollte für spätere Fehlersuche unbedingt in der Konsole ausgegeben werden. Insbesondere fehlende Leerzeichen sind dort sehr schnell erkennbar.

```
public void createAccountTable() throws SQLException{
    String sql = "create table account (" +
        "userid CHAR(16) NOT NULL PRIMARY KEY," +
        "password CHAR(32) NOT NULL ,," +
        "active CHAR(1) NOT NULL DEFAULT 'Y','" +
        "admin CHAR(1) NOT NULL DEFAULT 'N','" +
        "username VARCHAR(256) ,," +
        "email VARCHAR(256) );";
    System.out.println(sql);
    Statement myStat = dbConn.createStatement();
    myStat.executeUpdate(sql);
    System.out.println("Tabelle account angelegt");
}
```

Bei SQL-Methoden kann fast immer eine SQLException auftreten. In unserem Beispiel werfen wir diese, so dass Eclipse im Fehlerfall letztlich einen Stack-Trace ausgeben wird.

Da das insert-Statement keine Parameter enthält, ist es nicht erforderlich, ein PreparedStatement zu verwenden. Da auch keine Daten anzuzeigen sind, ist der JDBC-Teil der Methode sehr einfach.

Methode dropAccountTable()

Die Methode dropAccountTable() löscht die Tabelle account. Sie ist analog zur create-Methode aufgebaut. Die auskommentierte Zeile zeigt die Kurzform des JDBC-Aufrufs.

```
public void dropAccountTable() throws SQLException {
    String sql = "drop table account";
    System.out.println(sql);
//    dbConn.createStatement().executeUpdate(sql);
//    Statement myStat = dbConn.createStatement();
//    myStat.executeUpdate(sql);
    System.out.println("Tabelle account gedroppt");
}
```

Methode insertFirstUser()

Die Methode insertFirstUser() demonstriert die Syntax eines statischen insert-Befehls:

```
public void insertFirstUser() throws SQLException{
    String sql = "INSERT INTO account " +
        "(USERID, PASSWORD, ACTIVE, ADMIN, USERNAME, EMAIL) " +
        "VALUES " +
        "('testuser1', 'geheim', 'Y', 'N', 'Testuser 1', 'testus1@test.de')";
    System.out.println(sql);
    Statement myStat = dbConn.createStatement();
    myStat.executeUpdate(sql);
    System.out.println("Testuser1 eingefügt");
}
```

Methode insertsecondUser()

Die Methode insertsecondUser() demonstriert die Syntax eines insert-Befehls mit PreparedStatement:

```

public void insertSecondUser() throws SQLException{
    String sql = "INSERT INTO account " +
        "(USERID, PASSWORD, ACTIVE, ADMIN, USERNAME, EMAIL) " +
        "VALUES (?, ?, ?, ?, ?, ?)";
    System.out.println(sql);
    PreparedStatement myStat = dbConn.prepareStatement(sql);
    myStat.setString(1, "testuser2");
    myStat.setString(2, "geheim");
    myStat.setString(3, "Y");
    myStat.setString(4, "N");
    myStat.setString(5, "Testuser 2");
    myStat.setString(6, "testus2@test.de");
    myStat.executeUpdate();
    System.out.println("Testuser2 eingefügt");
}

```

Methode insertMoreUsers()

Die Methode insertMoreUser() demonstriert, wie mit einem einzelnen PreparedStatement-Objekt mehrere verschiedene Datensätze nacheinander verarbeitet werden können:

```

public void insertMoreUsers() throws SQLException{
    String sql = "INSERT INTO account " +
        "(USERID, PASSWORD, ACTIVE, ADMIN, USERNAME, EMAIL) " +
        "VALUES (?, ?, ?, ?, ?, ?)";
    System.out.println(sql);
    PreparedStatement myStat = dbConn.prepareStatement(sql);
    myStat.setString(1, "testuser3");
    myStat.setString(2, "geheim");
    myStat.setString(3, "Y");
    myStat.setString(4, "N");
    myStat.setString(5, "Testuser 3");
    myStat.setString(6, "testus3@test.de");
    myStat.executeUpdate();
    System.out.println("Testuser eingefügt");
    myStat.setString(1, "testuser4");
    myStat.setString(2, "geheim");
    myStat.setString(3, "Y");
    myStat.setString(4, "N");
    myStat.setString(5, "Testuser 4");
    myStat.setString(6, "testus4@test.de");
    myStat.executeUpdate();
    System.out.println("Testuser eingefügt");
}

```

Statische vs. dynamische Datenübergabe

Die Übergabe der Daten in die PreparedStatements erfolgte in dieser Anwendung statisch bzw. hart codiert, d.h. die Daten wurden direkt als Konstanten in die entsprechenden Felder geschrieben.

In der Praxis werden die Daten meistens aus Variablen stammen. Dies ändert die Syntax jedoch kaum. Wir demonstrieren dies an dieser Stelle nicht, weil hier nur Grundlagen wiederholt werden sollen. Die Datenübergabe über Variablen wird Inhalt anderer Kapitel sein.

Methode showAccountData()

Die Methode showAccountData() beendet unsere Wiederholung von JDBC. Einige Details sind erkennbar:

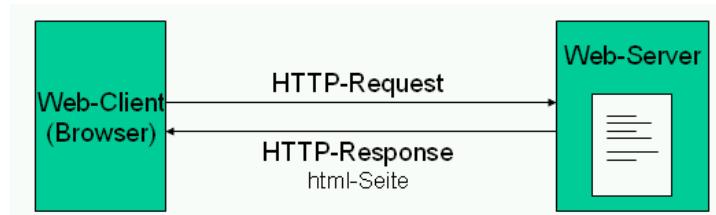
- Im SELECT-Statement wurden alle Feldnamen angegeben (anstatt nur SELECT * FROM account), um Klarheit zu schaffen, welche Daten zu erwarten sind und in welcher Reihenfolge.
- Anstatt executeUpdate(sql) wurde executeQuery(sql) verwendet. Entsprechend liefert executeQuery ein ResultSet zurück.
- In einem ResultSet dbRes ist immer eine Zeile „aktuell“. Mit dbRes.next() wird beim ersten Mal die erste Zeile zur aktuellen Zeile gemacht, danach jeweils auf die nächste Zeile weitergeschalten.
- next() liefert true zurück, wenn eine Zeile gefunden wurde, andernfalls false. Deshalb kann eine while-Schleife genutzt werden, um den Durchlauf zu steuern.
- Mit getString(„USERID“) wird in der aktuellen Zeile das Feld USERID gelesen, weil USERID ein String-(Character-)Attribut ist. Es gibt auch Methoden getInt(), etc., die hier aber nicht zum Einsatz kommen.

```
public void showAccountData() throws SQLException{
    String sql = "SELECT USERID, PASSWORD, ACTIVE, ADMIN, USERNAME, EMAIL " +
        "FROM account ";
    System.out.println(sql);
    ResultSet dbRes = dbConn.createStatement().executeQuery(sql);
    while (dbRes.next()){
        String userid   = dbRes.getString("USERID");
        String password = dbRes.getString("PASSWORD");
        String active   = dbRes.getString("ACTIVE");
        String admin    = dbRes.getString("ADMIN");
        String username = dbRes.getString("USERNAME");
        String email    = dbRes.getString("EMAIL");
        String outString = userid + ", ";
        outString += password + ", ";
        outString += active + ", ";
        outString += admin + ", ";
        outString += username + ", ";
        outString += email;
        System.out.println(outString);
    }
}
```

4 Java Server Pages – Hintergründe und Vorarbeiten

4.1 Grundlagen: Dynamische Webseiten

In 2.1.1 haben wir statische Webseiten (html-Seiten) betrachtet, die auf einem Webserver gespeichert sind und durch einen http-Request abgerufen werden können. Die http-Response enthält dann im Body die html-Seite. Wir haben dies mit der folgenden Abbildung veranschaulicht:



Als Erweiterung können dynamische Webseiten betrachtet werden, die ebenfalls durch einen http-Request angefordert werden. Wiederum besteht die Antwort aus einer html-Seite, die in der http-Response zurückgegeben wird. Allerdings wird die html-Seite im dynamischen Fall durch ein Programm auf dem Webserver erst erzeugt:



Anmerkungen:

- Das Programm, das die html-Seite erzeugt, kann als html-Generator bezeichnet werden, der auf dem Webserver lauffähig sein muss. Man spricht deshalb von **Server-seitiger Entwicklung**.
- Server-seitige html-Generatoren können z.B. mit PHP oder mit Java Server Pages entwickelt werden.
 - PHP ist eine Skriptsprache ähnlich zu JavaScript. Sie kann z.B. auf einem Apache-Webserver ausgeführt werden, wenn dieser durch ein PHP-Modul erweitert wird.
 - Java Server Pages (JSP) ist ein Framework, das auf der Sprache Java aufbaut und das nicht auf dem Standard Apache Webserver lauffähig ist.
 - PHP und JSP unterscheiden sich von der Art der Benutzung kaum, insbesondere können in beiden Sprachen statische und dynamische Anteile innerhalb einer Webseite gemischt werden. Außerdem erlauben beide Sprachen ähnliche Softwarearchitekturen.

➔ Wir werden in dieser Veranstaltung JSPs verwenden, damit wir unsere bereits gelernten Kenntnisse der Sprache Java weiterverwenden können.

Weitere Anmerkungen:

- Im http-Request können Parameter an den Webserver übertragen werden, vgl. 2.2.3. Bei statischen Seiten haben diese keinen Einfluss auf die zurückgegebene Webseite, bei dynamischen Webseiten werden aufgrund unterschiedlicher Parameter in der Regel auch unterschiedliche Ergebnisseiten erzeugt.
- Die Daten, die für die Erzeugung der Ergebnisseiten benötigt werden, werden meistens auf der Server-Seite aus einer Datenbank gelesen, z.B. Produktinformationen eines Onlineshops.
- html-Generator und Webserver müssen außerdem in der Lage sein, die erzeugte html-Seite in den Body der http-Response zu schreiben.

4.2 Praxis: TomCat in Eclipse einrichten

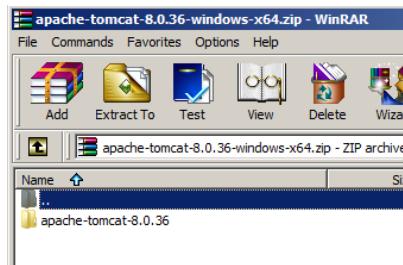
Da JSPs nicht im „normalen“ Apache Webserver lauffähig sind, wird ein JSP-fähiger Webserver benötigt. Wir verwenden hier den Webserver TomCat, bei dem es sich ebenfalls um ein Apache-Projekt handelt. Zur Entstehungszeit dieses Manuskripts verwenden wir die Version TomCat 8.0.

TomCat kann in Eclipse eingebunden werden, vgl. 0, was die Entwicklung stark vereinfacht. In Produktivumgebungen wird TomCat in der Regel ohne Eclipse deployed. Dies behandeln wir in dieser Veranstaltung nicht.

TomCat herunterladen

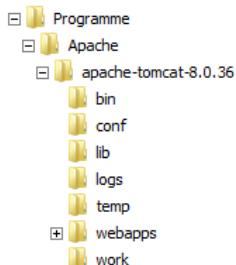
Ausgangspunkt für Tomcat ist die Webseite tomcat.apache.org. Dort sind Download-Seiten verlinkt. Im Zusammenhang mit Eclipse Mars sollte Tomcat 8.0 verwendet werden. Zur Entstehungszeit dieses Manuskripts ist die Version Tomcat 8.0.36 aktuell.

Es sollte eine ZIP-Datei und kein Windows Service Installer heruntergeladen werden. In unserem Fall die Datei apache-tomcat-8.0.36-windows-x64.zip



TomCat entpacken

Bei apache-tomcat-8.0.36-windows-x64.zip handelt es sich um ein Archiv, das beim Entpacken ein Verzeichnis apache-tomcat-8.0.36 erzeugt. Als Ort für das Entpacken bietet sich C:\Programme\Apache an, so dass folgende Dateistruktur entsteht. Eine weitergehende Installation von Tomcat ist nicht nötig und auch nicht sinnvoll:



Der Tomcat-Webserver wird damit bereitgestellt aber noch nicht gestartet. Zum Start von Tomcat wird eine Server-Konfiguration in Form der XML-Datei server.xml benötigt, die während der Entwicklung von Eclipse erzeugt wird. Das Starten und Stoppen des Tomcat-Service wird damit von Eclipse verwaltet.

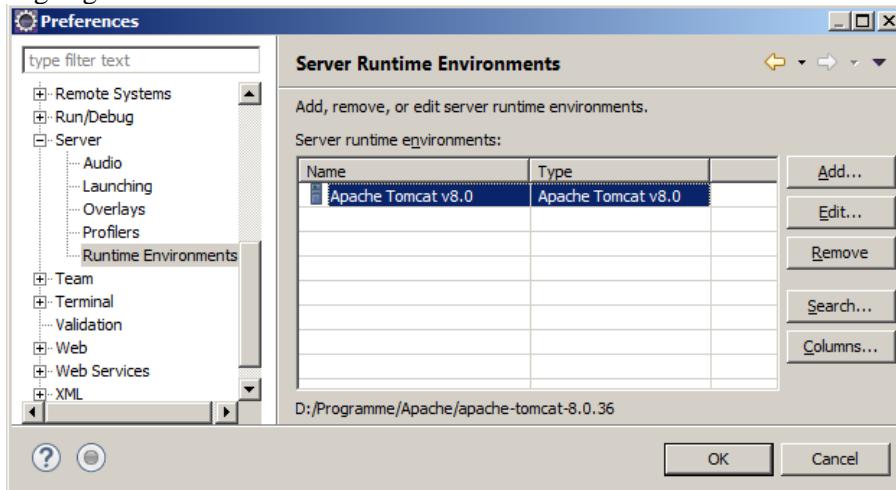
Damit dies möglich ist, darf Tomcat nicht außerhalb von Eclipse gestartet werden. Deshalb ist auch von der Installation durch den Windows Installer abzuraten, weil dieser den Tomcat als Dienst installiert und sofort startet.

TomCat in Eclipse einbinden

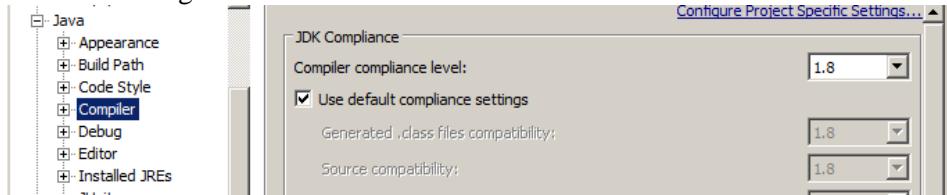
Innerhalb von Eclipse sind verschiedene Orte zu prüfen bzw. einzurichten:

- Window -> Preferences -> Server -> Runtime Environments
 - hier ist Tomcat einzutragen mit Add... und unter Angabe des „Tomcat Installation Directory“, in unserem Fall C:\Programme\Apache\apache-tomcat-8.0.36. Falls doppelte Einträge bestehen sollten diese bereinigt werden, z.B. indem alle Einträge gelöscht werden und genau einer neu

angelegt wird.



- Für die Veranstaltung wird ein Java Development Kit (JDK) benötigt. Ein einfaches JRE (Java Runtime Environment) reicht zwar für die ersten Schritte aber nicht für den Datenbankzugriff. Dieses muss auch unter Window -> Preferences -> Java -> Installed JREs verankert werden.
- Window -> Preferences -> Java -> Compiler: Hier empfehlen wir derzeit 1.8, speziell im Zusammenhang mit Tomcat 8.0



- Für die JSP-Entwicklung ist in Eclipse mindestens ein Dynamic Web Project erforderlich. Beim Anlegen dieses Projekts kann als „Target Runtime“ Apache Tomcat v8.0 angegeben werden. Falls dies nicht angegeben wurde, kann dies später in den Projekteigenschaften nachgeholt oder korrigiert werden. Genauer: Rechtsklick auf das Projekt -> Properties -> Targeted Runtimes. Hier sollte Apache Tomcat v8.0 zu finden sein. Andernfalls könnte er hier angelegt werden, aber nur, wenn er bereits in den Window -> Preferences angelegt wurde (s.o.). Dies sollte also bereits vorher erfolgen.
- Unter <Projekt, Rechtsklick> -> Properties -> Java Compiler sollte das Compiler Compliance Level ebenfalls auf 1.8 stehen.
- <Projekt, Rechtsklick> -> Properties -> Java Build Path -> Libraries
 - Hier sollte die JRE System Library geprüft werden. Diese sollte mindestens Java 1.8 unterstützen. Meistens ist „Workspace Default JRE“ sinnvoll. Dieser Punkt ist generell auch im Zusammenhang mit Window -> Preferences -> Java -> Installed JREs zu sehen (s.o.)
 - Bei den Libraries sollte auch Apache Tomcat v8.0 stehen, damit in Eclipse die Code-Vervollständigung in Bezug auf Server-Objekte wie request und response (vgl. 5.1.4) funktioniert. Andernfalls sollte diese hinzugefügt werden mit Add Library ... -> Server Runtime -> Apache Tomcat v8.0. Damit das funktioniert muss der Tomcat bereits in den Window -> Preferences eingetragen sein (s.o.)

5 Java Server Pages – Getting Started

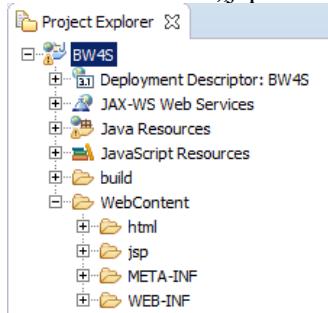
5.1 Grundlagen

5.1.1 Statische und dynamische Teile einer Webseite

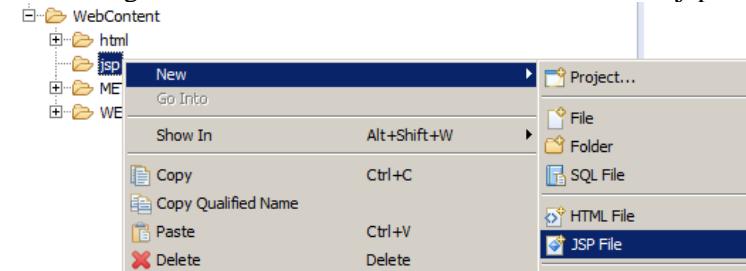
Das erste Beispiel für JSP ist an vielen Stellen im Netz zu finden. Es handelt sich um eine Datums- und Zeitanzeige.

JSP-Beispiel für Einsteiger: new java.util.Date()

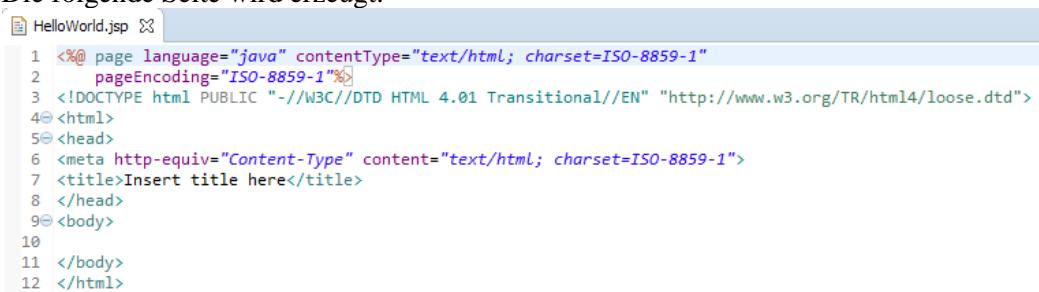
- Im Verzeichnis WebContent unseres Dynamic Web Projects BW4S legen wir ein Unterverzeichnis „jsp“ an:



- Wir erzeugen eine JSP mit Rechtsklick auf WebContent\jsp -> New -> JSP File



- Falls JSP-File hier nicht angezeigt wird, könnte dies an der Eclipse-Perspective liegen (oben rechts im Eclipse-Hauptfenster), diese sollte auf Java EE gesetzt sein.
- Wir geben unserer neuen JSP den Dateinamen HelloWorld.jsp -> Finish.
- Die folgende Seite wird erzeugt:



```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2     pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4<html>
5<head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>Insert title here</title>
8 </head>
9<body>
10
11 </body>
12 </html>
```

- Wir beobachten
 - a) die Sprachangabe <%@page language="java" ... %>
 - b) dass der Rest der Seite reines html ist
 - c) dass der Body der Seite (zu Beginn noch) leer ist.

- Wir tragen in den Body einen Teil statisches html und einen Teil dynamisches JSP ein, in diesem Fall ein Java-Scriptlet:

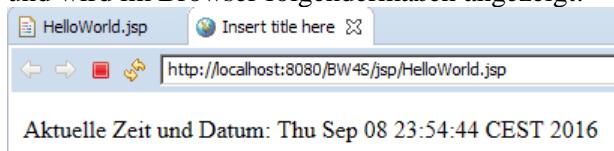
```
<body>
<p>Aktuelle Zeit und Datum:</p>
<% java.util.Date myDate = new java.util.Date();
   String myDateString = myDate.toString();
   out.print(myDateString);
%>
</p>
</body>
```

- Der statische Teil, in diesem Fall das <p>-Tag mit dem zugehörigen Text landen direkt in der html-Ergebnisseite.
- Durch <% ... %> wird ein JSP-Scriptlet erzeugt, das den dynamischen Teil enthält. Es handelt sich um Java Code, der auf dem Server ausgeführt wird:
 - Es wird ein Objekt namens myDate vom Typ java.util.Date erzeugt. Per Default enthält dieses Objekt das aktuelle Datum und die aktuelle Uhrzeit zum Zeitpunkt der Erzeugung.
 - Durch `toString()` wird myDate in ein String-Objekt umgewandelt, das nun ebenfalls Datum und Uhrzeit enthält.
 - Durch `out.print(myDateString)` wird der String in die html-Ergebnisseite geschrieben, also in den Body der Response, da dieser das html-Ergebnis enthält und an den Client transportiert.
 - out ist dabei ein in JSP eingebautes Objekt, das gerade den Body der http-Response repräsentiert.

- Auf dem Client kommt der html-Quelltext der Ergebnisseite deshalb folgendermaßen an:

```
<body>
<p>Aktuelle Zeit und Datum:</p>
Thu Sep 08 23:54:44 CEST 2016
</p>
</body>
```

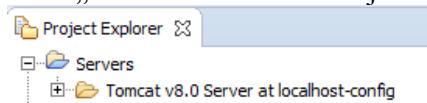
- und wird im Browser folgendermaßen angezeigt:



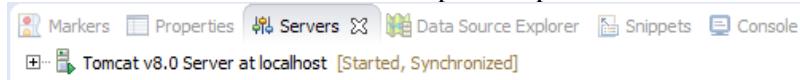
Ausführung der JSP

Damit die Seite so ankommen kann, muss diese aber auf dem Tomcat-Server ausgeführt werden:

- Rechtsklick auf die Datei HelloWorld.jsp -> Run As -> Run on Server -> Tomcat v8.0 Server -> Finish
Falls Tomcat hier nicht angezeigt wird, müssen die Schritte aus 4.2 noch einmal durchlaufen werden.
- Im Hintergrund generiert Eclipse nun eine passende Tomcat-Konfiguration und speichert diese unter „Servers“ sowohl im Project Explorer



als auch im unteren Bereich des Eclipse-Hauptfensters:



- Außerdem wird Tomcat mit dieser Konfiguration gestartet. Die Meldungen während des Starts landen in der Eclipse-Java-Konsole:

```
Tomcat v8.0 Server at localhost [Apache Tomcat] D:\Programme\Java\bin\javaw.exe (08.09.2016, 23:54:40)
Sep 08, 2016 11:54:43 PM org.apache.coyote.AbstractProtocol start
INFORMATION: Starting ProtocolHandler ["http-nio-8080"]
Sep 08, 2016 11:54:43 PM org.apache.coyote.AbstractProtocol start
INFORMATION: Starting ProtocolHandler ["ajp-nio-8009"]
Sep 08, 2016 11:54:43 PM org.apache.catalina.startup.Catalina start
INFORMATION: Server startup in 1403 ms
```

- Auf dem Tomcat-Server wird jetzt die JSP ausgeführt:

- Alle statischen html-Teile werden direkt in die html-Ergebnisseite (den Body der http-Response) geschrieben.
- Das JSP-Scriptlet wird ausgeführt. Von drei Befehlen schreibt nur einer in die html-Ergebnisseite (den Body der http-Response).
- Nachdem die JSP ausgeführt wurde, wird die http-Response an den Client geschickt. Wie bei statischen html-Seiten liest dieser nun den html-Text aus dem Body der Response und gibt diesen an den Browser, der die html-Seite geeignet anzeigt.

Abkürzung für die JSP

Dieses Beispiel wird im Internet meistens knapper dargestellt. Wir kürzen das Beispiel schrittweise:

```
<% java.util.Date myDate = new java.util.Date();
   String myDateString = myDate.toString();
   out.print(myDateString);
%>
```

- Wir erinnern uns, dass print(...) oder println(...) Befehle in Java ihren Argumenten automatisch eine toString()-Methode schicken, so dass print(...) auf beliebige Objekte angewendet werden kann. Unser Beispiel kann deshalb reduziert werden zu:

```
<% java.util.Date myDate = new java.util.Date();
   out.print(myDate);
%>
```

und durch zusammenfassen weiter zu

```
<% out.print(new java.util.Date()); %>
```

- Wir beobachten, dass der gesamte Java Code nun nur noch aus einem einzigen out.print(...)-Befehl besteht, der einen bestimmten Wert an die html-Ergebnisseite (den Body der http-Response) schicken soll.
- Für diese Situation, die oft auftaucht, bietet JSP eine Abkürzung, nämlich die sogenannte JSP-Expression <%= ... %>, bei der wir nur noch den Ausdruck angeben müssen. Tomcat ergänzt dies dann automatisch um den out.print(...)-Befehl. Die Kurzform unseres Beispiels lautet also

```
<body>
<p>Aktuelle Zeit und Datum:
<%= new java.util.Date() %>
</p>
</body>
```

Wir sparen also den out.print(...)-Befehl samt Klammern und auch das Semikolon muss nicht mehr gesetzt werden.

- Wir führen diese Form aus wie oben und erhalten dasselbe Ergebnis wie in der ausführlichen Form.

Zusammenfassung

In einer JSP können statisches html und Java-Programmfragmente gemischt werden.

- Die statischen html-Anteile werden direkt in die Ergebnisseite geschrieben. Da die Ergebnisseite im Body der http-Response an den Client übertragen wird, bedeutet dies, dass statische html-Anteile direkt in den Body der Response geschrieben werden.
- Dies erfolgt über das in JSP eingebaute Objekt out, das für den Body der http-Response steht.
- Die Java Fragmente (die dynamischen Anteile) haben nur dann Einfluss auf die Ergebnisseite, wenn dies ausdrücklich in Form eines Java-Ausgabebefehls angewiesen wird...
- oder durch if-Anweisungen oder Schleifen Einfluss auf die statischen Anteile hat
- Mit <%= ... %> wird ein JSP-Scriptlet definiert, das beliebigen Java Code enthalten kann. Wenn out.print(...); Befehle enthalten sind, wird in die html-Ergebnisseite (den Body der http-Response) geschrieben.
- Mit <%= ... %> wird eine JSP-Expression definiert. Diese darf nur einen Ausdruck enthalten (ohne Semikolon am Ende) und der Wert des Ausdrucks wird direkt in die html-Ergebnisseite (den Body der http-Response) geschrieben. Dies ist in der Regel eine Abkürzung für ein JSP-Scriptlet, das nur einen out.print(...); Befehl enthält.

5.1.2 Kontrollstrukturen

Innerhalb einer JSP können mehrere JSP-Scriptlets und JSP-Expressions eingesetzt werden. Das Zusammenspiel zwischen statischen und dynamischen Teilen der Seite kann außerdem durch Java-Kontrollstrukturen gesteuert werden. Wir beleuchten dies anhand einiger Beispiele.

Lokale Variablen

- Wir erweitern unsere HelloWorld.jsp um den folgenden Code:

The screenshot shows the code editor on the left and the browser output on the right. The code editor contains the following JSP code:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Hello World</title>
</head>
<body>
<% String hello = "Hello World!"; %>
<p>Aktuelle Zeit und Datum:<br/>
<%= hello + " es ist jetzt " + new java.util.Date() %>
</p>
</body>
</html>
```

The browser output shows the rendered HTML with the current date and time inserted after the greeting message.

- Beobachtungen:

- Wir haben den <title> angepasst, das hat allerdings nichts mit JSP zu tun. Anpassungen im statischen Teil sind erlaubt.
- Wir haben ein JSP-Scriptlet eingefügt, in dem eine (lokale) String-Variable hello definiert wird.
- Wir haben in einem anderen JSP-Fragment, genauer: In unserer JSP-Expression, auf hello zugegriffen.
- Die Ausführung zeigt, dass der Wert von hello zwischen den unterschiedlichen JSP-Fragmenten der einen Seite erhalten bleibt.

- Anmerkungen zur Ausführung:

- Die JSP wird wiederum durch Run As -> Run on Server -> Tomcat 8.0 gestartet.
- Falls die Seite noch vom letzten Mal offen war, reicht eine Aktualisierung der Seite, z.B. mit F5. Allerdings dauert es einige Sekunden, bevor die Änderung der JSP bei einer Aktualisierung der Seite im Browser wirksam wird. Dies hat technische Gründe, die mit der Architektur des JSP-Frameworks und der Funktionsweise des JSP-Servers (bei uns Tomcat) zusammenhängen.

for-Schleifen

- Schritt 1: Wir erweitern unsere HelloWorld.jsp um den folgenden Code:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
<% int index = 1;
while (index <= 10){
    out.print("Wir betrachten die nächste Zahl: ");
    if (index % 2 == 0){
        out.println("Die " + index + " ist gerade<br>");
    }else{
        out.println("Die " + index + " ist ungerade<br>");
    }
    index++;
}
%>
```

Wir betrachten die nächste Zahl: Die 1 ist ungerade
Wir betrachten die nächste Zahl: Die 2 ist gerade
Wir betrachten die nächste Zahl: Die 3 ist ungerade
Wir betrachten die nächste Zahl: Die 4 ist gerade
Wir betrachten die nächste Zahl: Die 5 ist ungerade
Wir betrachten die nächste Zahl: Die 6 ist gerade
Wir betrachten die nächste Zahl: Die 7 ist ungerade
Wir betrachten die nächste Zahl: Die 8 ist gerade
Wir betrachten die nächste Zahl: Die 9 ist ungerade
Wir betrachten die nächste Zahl: Die 10 ist gerade

- Da wir hier nur Java (in Form eines JSP-Scriptlets) geschrieben haben, ist das Ergebnis vorhersehbar.
- Beobachtung: Der erste out.print(...) erzeugt nur konstanten, also statischen, Output.

- Schritt 2: Den konstanten Output können wir auch als normalen html-Text schreiben, ohne dass sich die Ausgabe ändert:

```
<% int index = 1;
while (index <= 10){ %>
    Wir betrachten die nächste Zahl:
    <% if (index % 2 == 0){
        out.print("Die " + index + " ist gerade<br>");
    }else{
        out.print("Die " + index + " ist ungerade<br>");
    }
    index++;
}
%>
```

- Beobachtung: Wir müssen dafür den konstanten Teil aus der Java-Klammer <% ... %> herausnehmen, indem wir diese vor dem konstanten Teil schließen und danach wieder öffnen. Dadurch sparen wir das `out.print(...)`, da statisches html automatisch in die html-Ergebnisseite geschrieben wird. Die while-Schleife bleibt gültig über die Grenzen der Java-Klammern.

- Schritt 3: Ebenso können wir die konstanten Anteile aus den anderen `out.print(...)`-Befehlen herausnehmen:

```
%> herausnehmen:  
%> int index = 1;  
%> while (index <= 10){ %>  
    Wir betrachten die nächste Zahl:  
    %> if (index % 2 == 0){ %>  
        Die <% out.print(index); %> ist gerade<br>  
    %> }else{ %>  
        Die <% out.print(index); %> ist ungerade<br>  
    %> }  
    index++;  
}  
%>
```

- Beobachtung: Auch die if-Bedingung bleibt gültig über die Grenzen der Java-Klammern hinweg. Der Code wird allerdings nicht unbedingt lesbarer.

- Schritt 4: Da sich die `out.print(...)`-Befehle jetzt nur noch auf einen einzelnen Wert, speziell den index, beziehen, können diese mit JSP-Expressions abgekürzt werden:

```
index, schicken, können diese mit JSTL  
% int index = 1;  
    while (index <= 10){ %>  
        Wir betrachten die nächste Zahl:  
        <% if (index % 2 == 0){ %>  
            Die <%= index %> ist gerade<br>  
        <% }else{ %>  
            Die <%= index %> ist ungerade<br>  
        <% }  
        index++;  
    }  
%>
```

Zusammenfassung:

- Java-Kontrollstrukturen bleiben über mehrere JSP-Scriptlets oder JSP-Expressions hinweg gültig.
 - Es ist Geschmackssache, ob statische Teile konsequent aus den Java-Klammern herausgenommen werden oder nicht.

for-Schleifen und Tabellen

Analog zum letzten Abschnitt erzeugt der folgende Code aus einem Array eine Tabelle:

```
<% String[] freundInnen = {"Martin", "Gaby", "Peter", "Carola"}; %>
<table class="border1">
<% for (String freundIn : freundInnen){ %>
    <tr><td class="border1"><%= freundIn %></td></tr>
<% } %>
</table>
```

Martin
Gaby
Peter
Carola

```
3
4 @CHARSET "ISO-8859-1";
5 *.border1 {
6   border: solid black 1px;
7   border-collapse: collapse;
8 }
```

Anmerkung: Code-Fragmente wie diese werden im weiteren Verlauf der Veranstaltung kaum vorkommen, da wir mit einer guten Architektur wesentlich besser lesbaren Code entwickeln werden.

5.1.3 Ergänzungen

Troubleshooting

Bei der Ausführung von JSPs mit Tomcat muss Eclipse im Hintergrund ziemlich viel arbeiten. Speziell bei häufigen Änderungen kann Eclipse dabei durcheinanderkommen. Für den Fall, dass die JSP korrekt ist (mehrmals durchgelesen und geprüft), haben sich die folgenden Tipps bewährt:

- Wenn Eclipse einen Fehler anzeigt, der nicht nachvollziehbar ist:
Code in die Zwischenablage kopieren -> löschen -> wieder einfügen -> Datei speichern
Oft ist der Fehler dann weg.
- Wenn nicht nachvollziehbar ist, warum die JSP nicht funktioniert:
Tomcat-Konfiguration im unteren Fenster löschen (dann ist sie auch links im Project Explorer weg) -> Projekt mit „Refresh“ aktualisieren -> Server neu anlegen lassen.
- Manchmal passen Java Version, Tomcat Version, Project Facet, o.ä. nicht zusammen → ausprobieren oder den Dozenten fragen
- Wenn der Tomcat-Dienst schon außerhalb von Eclipse gestartet wurde, kann Eclipse diesen nicht auch starten. Lösung: Dienst außerhalb von Eclipse zunächst beenden, dann innerhalb von Eclipse weiterarbeiten.

Kommentare in JSPs

Mit <%-- ... --%> kann ein sogenannter JSP-Kommentar in den statischen Teil einer JSP eingefügt werden. Da bereits der Server erkennt, dass es sich um einen Kommentar handelt, wird dieser nicht an den Client ausgeliefert. Im html-Quellcode ist der Kommentar also nicht sichtbar.

Innerhalb von JSP-Scriptlets sind außerdem normale Java-Kommentare //... oder /* ... */ möglich, die ebenfalls nicht an den Client ausgeliefert werden.

Im Gegensatz dazu stellt <!-- ... --> einen html-Kommentar dar, der vom Server als html an den Client geliefert wird, also im html-Quellcode erscheint, allerdings vom Browser nicht an der Oberfläche angezeigt wird.

Sichtweisen

Man kann eine JSP betrachten als

- HTML-Seite mit dynamischen (Java-)Inhalten: Das bietet sich an, wenn nur wenig Java-Code enthalten ist.
- Java-Methode, die HTML-Output erzeugt, wobei HTML-Inhalt entweder über out.print(" "); ausgegeben oder einfach in die Datei geschrieben wird. Das bietet sich an, wenn if-Statements oder Schleifen auf mehrere JSP-Scriptlets verteilt sind.

5.1.4 Verbindung zwischen JSP und http

Da mit JSPs Ergebnisse eines Webservers, also eines http-Servers erzeugt werden, muss JSP eng mit http verzahnt werden. Dies bezieht sich insbesondere auf die drei Punkte:

- Weitergabe der html-Ergebnisseite an den Body der http-Response über das eingebaute Objekt out (vgl. Kap. 4 und 5.1.1)
- lesen / auswerten von Parametern aus dem http-Request (s.u.)
- Weiterleitung auf eine andere Webseite, z.B. durch einen Response-Header (s.u.)

Eingebaute Objekte request und response

Wir haben bereits das in JSP eingebaute Objekt out kennengelernt. Wir konnten dieses benutzen, ohne es vorher deklarieren zu müssen wie es in Java eigentlich üblich sein müsste. Dies liegt daran, dass es vom JSP-Framework zur Verfügung gestellt wird.

Zwei weitere eingebaute Objekte, die benutzt werden können, ohne dass sie vorher deklariert werden müssen, sind

- request – enthält alle Informationen des http-Requests, z.B. die übertragenen Parameter, aber auch die http-Header, vgl. 2.2.3 und 2.2.4. Aus dem request-Objekt wird in der Regel nur gelesen.
- response – enthält alle Informationen der http-Response, die an den Client geschickt werden. Insbesondere können über das response-Objekt Header der http-Response gesetzt werden.

Auswertung von Request-Parametern

Request-Parameter kommen in der Regel aus html-Formularen und bestehen aus einem name und einem value. Sie können per GET oder per POST übertragen werden, vgl. 2.2.3.

Für jedes input-Tag, das einen name hat, wird ein name/value-Paar übertragen. Ausnahme sind submit-Buttons, von denen nur derjenige übertragen wird, der tatsächlich geklickt wurde.

Wenn der name bekannt ist, kann der value mit dem Befehl `request.getParameter(<name>)` gelesen werden. Wird z.B. der Parameter `tool=screwdriver` übertragen, dann kann mit

- `String tool = request.getParameter(„tool“)`

der Wert des Parameters in die Variable `tool` gelesen werden. Der Wert der Variable `tool` ist demnach anschließend ebenfalls „screwdriver“

Anmerkungen:

- Der Befehl `getParameter(...)` wird also an das fest eingebaute Objekt `request` geschickt.
- Der name entspricht dem name-Attribut aus dem input-Tag des Formulars, das den Request ausgelöst hat.
- Üblicherweise soll die Java-Variable, in der der value gespeichert wird, ebenso heißen wie der name des input-Tags bzw. des übertragenen Parameters.
- Da lokale Java-Variablen mit Kleinbuchstaben beginnen sollen, gilt dies auch für die input-Tags der html-Formulare.
- Wird ein Name verwendet, der gar nicht übertragen wird, dann ist der value null.
- Meistens sind die verwendeten Namen bekannt. Wenn nicht können diese mit `request.getParameterNames()` gelesen werden, vgl. Übungsaufgabe

Weiterleitung auf eine andere JSP

Zur Weiterleitung gibt es zwei verschiedene Techniken:

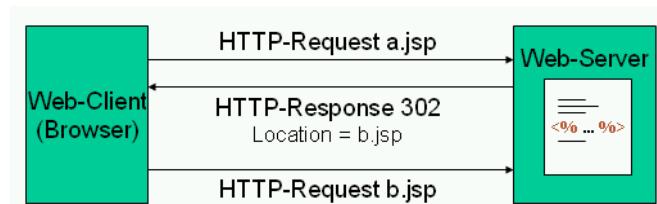
- Weiterleitung über den Client mit `response.sendRedirect(...)`
- Direkte Weiterleitung innerhalb des Servers mit `forward(...)`

Wir werden im Folgenden beide Varianten darstellen. Im weiteren Verlauf der Veranstaltung werden wir aber nur die erste davon verwenden.

Weiterleitung mit `response.sendRedirect(...)`

Bei Weiterleitung mit `response.sendRedirect(...)` sendet der Server an den Client eine http-Response, die nicht den Code 200 OK enthält, vgl. 2.1.2, sondern den Code 302 REDIRECT. Außerdem enthält die Response eine Header-Zeile `Location=...`, in der angegeben wird, welche Seite anstatt der aktuellen Seite abgerufen werden soll.

Der Client reagiert auf diese http-Response, indem er tatsächlich die angegebene Seite aufruft und diese anzeigt.

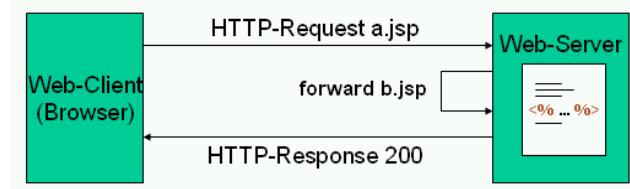


Der Befehl, um eine solche Weiterleitung auszuführen lautet `response.sendRedirect(<ziel>)`, d.h. der Befehl `sendRedirect(...)` wird an das fest eingebaute Objekt `response` geschickt.

Details und Java-Code betrachten wir in der praktischen Anwendung.

Weiterleitung mit forward(...)

Eine Weiterleitung mit forward(...) findet innerhalb des Servers statt. Der Client merkt gar nicht, dass er seine Response von einer anderen Seite erhält, als der, an die er seinen Request gerichtet hatte.



Zur Implementierung wird z.B. der folgende Java-Code verwendet:

```
<%
String target = "/jsp/RegView.jsp";
RequestDispatcher dispatcher = this.getServletContext().getRequestDispatcher(target);
dispatcher.forward(request, response);
%>
```

Anmerkungen:

- In der URL-Zeile des Client-Browsers steht nach dem forward() immer noch die Originalseite, z.B. a.jsp. Diese Technik kann also verwendet werden, um die tatsächliche Zielseite zu verschleiern.
- Ein forward() wird auch schneller sein als eine Weiterleitung mit response.sendRedirect(), kann aber nur auf demselben Server stattfinden.
- Im Gegensatz zu forward() kann response.sendRedirect() auf eine beliebige Seite im Netz weiterleiten.

5.2 Praktische Anwendung: RegView.jsp und RegAppl.jsp

5.2.1 RegView.jsp und Parameter

RegView.jsp

Im html-Teil dieser Veranstaltung hatten wir eine Datei RegView.html entwickelt, die das folgende Formular enthält:

```
<form action="..\jsp\RegAppl.jsp" method="get">
  <table>
    <tr>
      <td>Nickname:</td>
      <td><input type="text" name="userid" value="" /></td>
    </tr>
    <tr>
      <td>Passwort:</td>
      <td><input type="password" name="password" value="" /></td>
    </tr>
    <tr>
      <td>Username:</td>
      <td><input type="text" name="username" value="" /></td>
    </tr>
    <tr>
      <td>E-Mail:</td>
      <td><input type="text" name="email" value="" /></td>
    </tr>
    <tr>
      <td><input type="submit" name="zumLogin" value="Zum Login" /></td>
      <td><input type="submit" name="register" value="Registrieren" /></td>
    </tr>
  </table>
</form>
```

Nach dem Abschicken dieses Formulars wird die RegAppl.jsp aufgerufen. Wenn aber die RegView.html nicht schon auf dem Tomcat gestartet wird, dann wird auch die RegAppl.jsp ohne Tomcat gestartet, so dass enthaltene JSP-Fragmente nicht ausgeführt werden.

Um hier nicht durcheinanderzukommen, legen wir eine JSP-Seite RegView.jsp an und kopieren das Formular aus der RegView.html in die RegView.jsp. Weitere Inhalte wie title, CSS, etc. kopieren wir ebenfalls.

Übertragene Parameter

Beim Aufruf der RegAppl.jsp werden die Parameter aus der RegView.jsp übertragen. Im Detail handelt es sich um

- Die Textfelder userid, password, username, email mit dem jeweils eingetragenen value.
- Ein name/value-Paar zumlogin=Zum Login oder register=Registrieren, je nachdem, welcher dieser Buttons zur Auslösung des Formulars geklickt wurde.

Da nur einer der Buttons geklickt werden kann, kommt der andere Button als Parameter nicht vor.

5.2.2 RegAppl.jsp und Auswertung der Request-Parameter

RegAppl.jsp

Zur Auswertung und Verarbeitung der Parameter, die von der RegView.jsp abgeschickt werden (vgl. 5.2.1), legen wir eine JSP-Seite RegAppl.jsp an.

Die RegAppl.jsp unterscheidet sich von der RegView.jsp ganz wesentlich:

- Sinn der RegView.jsp ist es, angezeigt zu werden und eine Schnittstelle zum Benutzer zu liefern. Aus diesem Grund enthält die RegView.jsp viel html aber (fast) kein Java.
- Sinn der RegAppl.jsp ist es, die Parameter einzulesen, zu verarbeiten und weitere Aktionen anzustoßen. Die RegAppl.jsp soll nicht angezeigt werden, sondern zum Schluss an eine anzeigbare Seite weiterleiten. Deshalb enthält die RegAppl.jsp kein html, sondern nur Java.

Auswertung der Request-Parameter

Wie in 5.1.4 dargestellt, lesen wir die übertragenen Parameter mit dem Befehl request.getParameter(...) ein und speichern diese in lokalen Java-Variablen desselben Namens:

```
<body>
<%
String userid = request.getParameter("userid");
String password = request.getParameter("password");
String username = request.getParameter("username");
String email = request.getParameter("email");
String zumlogin = request.getParameter("zumlogin");
String register = request.getParameter("register");
%>
</body>
```

Um testen zu können, dass die Parameter-Übergabe tatsächlich so funktioniert, schreiben wir die Java-Variablen anschließend mit out.println(...) in die html-Ergebnisseite und mit System.out.println(...) in die Konsole:

```
out.println("userid: " + userid + "<br>");
out.println("password: " + password + "<br>");
out.println("username: " + username + "<br>");
out.println("email: " + email + "<br>");
out.println("zumlogin: " + zumlogin + "<br>");
out.println("register: " + register + "<br>");

System.out.println("userid: " + userid);
System.out.println("password: " + password);
System.out.println("username: " + username);
System.out.println("email: " + email);
System.out.println("zumlogin: " + zumlogin);
System.out.println("register: " + register);
%>
```

Wir testen die Parameter-Übergabe, indem wir die RegView.jsp aufrufen (Run As -> Run On Server -> Tomcat 8.0), Werte eingeben, einen der beiden Buttons klicken (z.B. Registrieren) und uns das Ergebnis sowohl auf der RegAppl.jsp als auch in der Konsole ansehen:

The screenshot shows a registration form on the left and its corresponding output on the right.

RegView.jsp:

- Nickname: sabine
- Passwort: *****
- Username: Sabine Mayer
- E-Mail: sm@abc.de
- Zum Login**
- Registrieren**

Output (Browser):

```
userid: sabine
password: geheim
username: Sabine Mayer
email: sm@abc.de
zumlogin: null
register: Registrieren
```

Output (Java Console):

```
Tomcat v8.0 Server at localhost [Apache Tomcat] D:\Programme\Java\bin\javaw.exe (09.09.2016, 13:37):
userid: sabine
password: geheim
username: Sabine Mayer
email: sm@abc.de
zumlogin: null
register: Registrieren
```

Beobachtungen:

- Die Ausgabe mit `out.print(...)` zeigt die eingelesenen Parameter auf der Webseite `RegAppl.jsp` an. Für eine zeilenweise Darstellung auf der Webseite werden ausdrücklich html-Tags `
` benötigt.
- Die Ausgabe mit `System.out.print(...)` zeigt die eingelesenen Parameter in der Java-Konsole an.
- Wir haben im Beispiel den Button `Registrieren` geklickt. Die Auswertung des nicht vorhandenen Buttons `zumlogin` führt tatsächlich zum Wert `null`.

Da die `RegAppl.jsp` keine Anzeige haben soll, löschen wir die `out.print(...)`-Befehle wieder aus dem Code. Die Anzeige in der Konsole reicht.

null-Prüfung

Wir werden im nächsten Abschnitt prüfen, welcher Button geklickt wurde. Hierfür benötigen wir if-Abfragen der Art `if (zumlogin.equals("Zum Login"))`. Falls `zumlogin == null` ist, läuft diese Abfrage auf eine `NullPointerException`.

Um dies zu verhindern, sorgen wir dafür, dass alle Variablen, die aus Buttons erzeugt wurden, nicht null sein dürfen. Wir ergänzen den Code zum Einlesen der Parameter wie folgt (nur die unteren beiden Zeilen sind neu):

```
String zumlogin = request.getParameter("zumlogin");
String register = request.getParameter("register");
if (zumlogin == null) zumlogin = "";
if (register == null) register = "";
```

Studierende früherer Semester haben darauf hingewiesen, dass dies auch kürzer codiert werden könnte in der Form `String zumlogin = "" + request.getParameter("zumlogin");`

Das ist auch grundsätzlich korrekt, allerdings macht die oben stehende explizite Form deutlicher, worum es geht.

null-Prüfungen werden uns in der Veranstaltung noch öfter begegnen, z.B. beim einlesen von (String-)Arrays.

5.2.3 Auswertung der angefragten Aktion

Anhand des geklickten Buttons kann entschieden werden, ob der Benutzer / die Benutzerin sich tatsächlich registrieren möchte oder doch eher zum Login springen möchte. Je nach dem geklickten Button ist eine andere Aktion auszuführen. Dafür programmieren wir zunächst eine folgende Verzweigung bzw. „Aktionsweiche“.

Wie in 5.2.2 zu sehen enthält die Java-Variable `register` den Wert „Registrieren“, wenn der entsprechende Button geklickt wurde. Dies entspricht dem name und dem value des entsprechenden input-Tags in der `RegView.jsp`. Analog enthält die Java-Variable `zumlogin` den Wert „Zum Login“, wenn dieser Button geklickt wurde.

Die entsprechenden Bedingungen sind demnach

- `if (zumlogin.equals("Zum Login"))`

- if (register.equals("Registrieren"))

Für die Aktionsweiche ergibt sich also der folgende Code:

```
if (zumlogin.equals("Zum Login")){
}else if (register.equals("Registrieren")){
}else{
}
```

Beobachtungen:

- Wir haben nicht nur Aktionszweige für zumlogin und register eingerichtet, sondern auch noch für den Fall, dass keiner der beiden Buttons geklickt wurde. Dieser Fall kann z.B. eintreten, wenn von einer anderen Anwendung aus auf die RegAppl.jsp gesprungen wird oder wenn Hacker versuchen, die RegAppl.jsp ohne Parameter oder mit falschen Parametern aufzurufen. Wir wollen auf jeden Fall vermeiden, dass die RegAppl.jsp angezeigt wird, da diese dafür nicht vorgesehen ist. Deshalb müssen wir auch für den unvorhergesehenen Fall, dass kein Button geklickt wurde, vorsorgen.
- Wir haben die Aktionsweiche so programmiert, dass immer genau einer (also mindestens einer, aber auch höchstens einer) der Zweige durchlaufen wird.

5.2.4 Weiterleitung auf sichtbare JSP

Nach Ausführung der RegAppl.jsp soll wieder eine sichtbare Seite angezeigt werden. Dies erfolgt durch Weiterleitung mit response.sendRedirect(...), vgl. 5.1.4.

Anmerkungen:

- Es ist zu beachten, dass wir hier zunächst die Ausführung der Aktionen überspringen. Natürlich erfolgt die Weiterleitung erst nach der Aktion.
- response.sendRedirect(...) beendet nicht die Ausführung der Seite wie es etwa der Java-Befehl return tun würde.
- response.sendRedirect(...) soll auf jeden Fall aufgerufen werden.
- response.sendRedirect(...) darf auf einer Seite nur einmal aufgerufen werden.
- Da in der Aktionsweiche genau ein Zweig durchlaufen wird, können die genannten Bedingungen erfüllt werden, indem jeder Zweig der Aktionsweiche mit einer Weiterleitung endet.

Die Weiterleitziele müssen gut überlegt werden, da sich hier die Qualität der Benutzerführung entscheidet. Grundsätzlich sind verschiedene Möglichkeiten denkbar. Wir wählen die Folgenden:

- Nach (erfolgreichem oder fehlgeschlagenem) Registrieren soll zurück auf die RegView.jsp gesprungen werden.
- Bei Zum Login soll auf eine Seite LoginAppl.jsp gesprungen werden, die ihrerseits zurück auf eine LoginView.jsp springen wird.
- Im else-Zweig soll ebenfalls auf die RegView.jsp gesprungen werden.

```
if (zumlogin.equals("Zum Login")){
    // Aktionscode hier

    response.sendRedirect("./LoginAppl.jsp");
}else if (register.equals("Registrieren")){
    // Aktionscode hier

    response.sendRedirect("./RegView.jsp");
}else{
    // Aktionscode hier

    response.sendRedirect("./RegView.jsp");
}
```

Beobachtung: Mit „./...“ wird im aktuellen Verzeichnis nach der entsprechenden Zielseite gesucht. Ggf. könnte das auch weggelassen werden, also z.B. direkt response.sendRedirect(„RegView.jsp“);

6 MVC und Java Beans

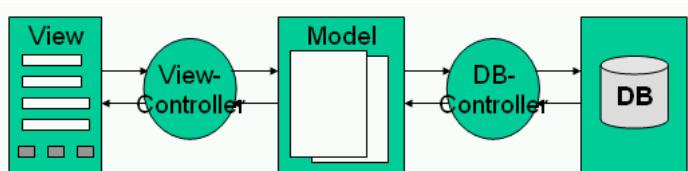
6.1 Theorie

- Große Software-Pakete müssen sauber untergliedert sein, damit sie gepflegt werden können. Diese Untergliederung ist eine der Aufgaben der Software-Architektur.
- Gut aufgebaute Software-Komponenten sollte man in anderen Software-Paketen wiederverwenden können.
- Gute Software-Architektur trennt die Darstellung von der Funktionalität.
- Meistens werden sogenannte Schichtenmodelle verwendet. Die Art und Anzahl der Schichten variiert je nach Art des Modells und Detaillierungsgrad.

6.1.1 Schichtenmodelle

5-Schichten Modell

In Schichtenmodellen werden Softwaresysteme aufgeteilt nach Komponenten je nach der Funktion der einzelnen Komponenten im Gesamtsystem. Eine typische Aufteilung ist die folgende:

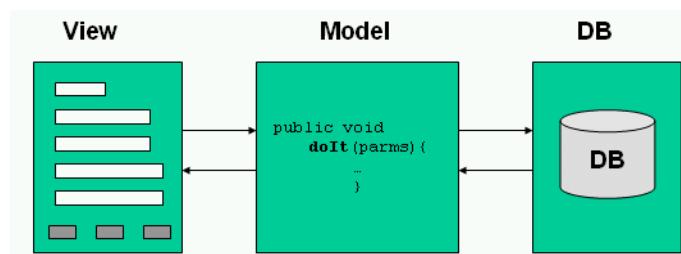


- Eine Darstellungsschicht (View) stellt die Ausgabe dar und bietet die visuelle Schnittstelle zum Benutzer.
- Ein Model enthält die Geschäftslogik und damit die eigentliche Funktionalität der Software, auch wenn diese häufig nur darin besteht, Daten von der View entgegenzunehmen und in Richtung Datenbank weiterzureichen.
- Die Datenbankschicht speichert (persistiert) die verarbeiteten Daten.
- Eine Steuerungsschicht (Controller) zwischen View und Model verbindet die Benutzerschnittstelle mit der Geschäftslogik.
- Eine weitere Steuerungsschicht zwischen Geschäftslogik und Datenbank kann die Speicherung der Daten vereinheitlichen und vereinfachen.

Eine Anforderung an die Schichten ist z.B., dass es möglich sein soll, die View oder die Datenbank auszutauschen, ohne die Geschäftslogik zu verändern oder anzupassen.

3-Schichten Modell

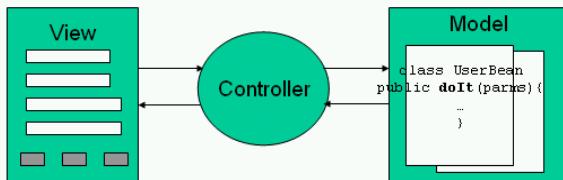
Nur der Vollständigkeit möchten wir erwähnen, dass in früheren Veröffentlichungen die Steuerungsschichten des 5-Schichten Modells meistens nicht dargestellt wurden, so dass von einer 3-Schichten-Architektur geredet wurde:



6.1.2 MVC – Model-View-Controller

Mit Einführung der grafischen Benutzeroberflächen hat sich herausgestellt, dass die Steuerungsschicht zwischen View und Model besondere Bedeutung hat, während die Verbindung zwischen Model und Datenbank eher problemlos machbar ist.

Deshalb wurden die entsprechenden Schichten besonders betrachtet und kurz als MVC bezeichnet (Abkürzung für „Model, View, Controller“). Aufgrund der besonderen Bedeutung stellen wir diese noch einmal ausführlicher dar:



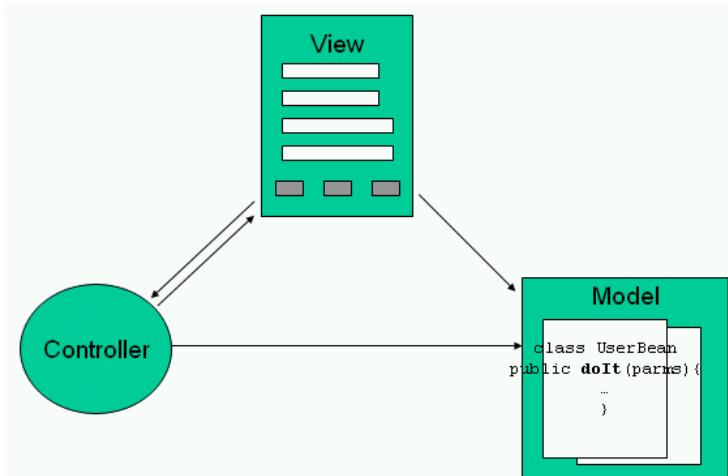
- Das Model enthält die Geschäftslogik. Hier entscheidet sich, welche Daten verarbeitet werden müssen und in welcher Form. Das Model soll die komplette Funktionalität enthalten und bei einer Änderung der View nicht angepasst werden müssen. In Java besteht das Model aus „ganz normalen“ Java-Klassen und stellt Methoden bereit, die vom Controller nur aufgerufen werden müssen. Die Datenbankschicht wird hier nicht separat aufgeführt, sondern als Teil der Geschäftslogik betrachtet.
- Die View enthält nur die Darstellungsschicht, aber keinerlei Geschäftslogik. Sie dient der Interaktion mit dem Benutzer, d.h. dem Benutzer wird eine Oberfläche angezeigt, in der dieser Eingaben vornehmen und Aktionen auslösen kann. Die Verarbeitung dieser Eingaben und Aktionen ist aber nicht Aufgabe der View.
- Der Controller
 - nimmt von der View die Eingaben und die ausgelösten Aktionen entgegen, führt diese aber nicht aus, sondern gibt diese an das Model weiter und löst dort die Durchführung aus.
 - Falls z.B. View und Model in unterschiedlichen Systemen oder mit verschiedenen Techniken laufen, können sich diese auf ihre jeweiligen Systeme konzentrieren und der Controller stellt die Verbindung her.
 - Im klassischen Modell spielt der Controller auch bei der Rückgabe der Ergebnisse vom Model an die View eine Rolle. Bei Webanwendungen wie wir sie betrachten werden, ist dies aber nicht der Fall.

Anmerkungen:

- Bei MVC handelt es sich weniger um eine exakte Trennung von Aufgaben als um ein Grundprinzip, das je nach den Möglichkeiten des verwendeten Systems unterschiedlich ausgeprägt werden kann.
- Die Trennung der Schichten ist nicht immer ganz einfach. Z.B. sind Eingabeprüfungen auf leere Felder, korrekte Datumsformate, etc. oft üblich. Über die Zuordnung solcher Prüfungen zu den verschiedenen Schichten gibt es unterschiedliche Meinungen.
- In manchen älteren MVC-Beschreibungen wird die Benutzerinteraktion dem Controller zugerechnet, bei unseren Webanwendungen sehen wir diese Interaktion bei der View.
- In der Literatur finden sich spezifische Ausprägungen des MVC-Prinzips, z.B. MVVM

6.1.3 MVC für Webanwendungen

Für Webanwendungen mit Java bzw. JSP ist wiederum eine spezifische Ausprägung des MVC-Prinzips erforderlich, insbesondere für die Rückgabe von Ergebnissen vom Model an die View:



- Model und View funktionieren zunächst wie oben beschrieben.
- Der Controller nimmt Eingabe und Aktionen entgegen (wie oben) und gibt diese weiter an das Model. Anschließend wählt der Controller die nächste anzuseigende Webseite aus.
- Wenn eine View angezeigt wird, holt sich diese ihre Daten direkt aus dem Model (das unterscheidet sich vom oben beschriebenen MVC).

6.1.4 Java-Beans

Eine Java-Bean ist eine „ganz normale“ Java-Klasse mit einer wichtigen Zusatzeigenschaft:

1. Eine Java-Bean benötigt einen public-Konstruktor ohne Parameter.

Zwei weitere Eigenschaften sind in der Literatur zu finden, sind für diese Veranstaltung aber nicht von zentraler Bedeutung:

2. Sie besitzt zu jedem Attribut eine Java-Bean get- und set-Methode.
3. Sie ist serialisierbar.

Die erste dieser Eigenschaften benötigen wir, um eine Java-Bean mittels JSP instanziieren zu können. Die zweite kann für unsere Fälle durch andere geeignete get-Methoden ersetzt werden. Die dritte Eigenschaft spielt in dieser Veranstaltung keine Rolle.

6.1.5 Ausprägung von MVC in dieser Veranstaltung

In dieser Veranstaltung werden wir eine Beispiel-Webanwendung entwickeln, die sich am MVC-Prinzip orientiert. Die einzelnen Schichten werden folgendermaßen implementiert:

- Die View wird implementiert durch JSPs (fast) ohne Java-Code. Beispiel: RegView.jsp
- Der Controller wird implementiert durch JSPs ohne Anzeige. Beispiel: RegAppl.jsp
- Das Model wird implementiert durch Java-Beans und ggfs. weitere unterstützende Java-Klassen, die nicht unbedingt Java-Beans sein müssen. Auch die Datenbankzugriffe gehören zur jeweiligen Java-Bean. Beispiel: UserBean.java

6.2 Praxis: UserBean und Datenbankzugriffe

Die UserBean stellt den ersten Teil des Models in unserer Registrierungsanwendung dar. Die UserBean soll einen Benutzer der realen Welt repräsentieren und soll vor Allem in der Lage sein, diesen in der Datenbank zu speichern.

Wir legen die UserBean zunächst als normale Klasse im Package de.hs_lu.bw4s.beans an. Falls das Package noch nicht existiert, legen wir dieses ebenfalls an:



6.2.1 Einstieg

Bei der UserBean handelt es sich um eine einzige Java-Klasse mit denselben Attributen wie in der Tabelle user:

Attribut	Anmerkung
String userid	
String password	
String active	Besagt, ob der User sich am Portal anmelden darf. Ist im Standardfall „Y“
String admin	Besagt, ob der User ein Administrator ist. Ist im Standardfall „N“
String username	
String email	

Damit die UserBean auch wirklich eine Bean wird, erhält sie zunächst die folgenden Standardmethoden (die Anforderung der Serialisierbarkeit verfolgen wir hier nicht explizit):

Methode	Anmerkung
public UserBean(){...}	Konstruktor der UserBean
public get...() und set...(...) Methoden	Können von Eclipse automatisch erzeugt werden

Im OLAT finden Sie die Vorlage UserBeanEmpty, die die genannten Attribute und Methoden (und außerdem eine `toString()`-Methode) enthält. Sie können diese in Ihr Eclipse-Projekt kopieren, diese mit Refactor -> Rename in „UserBean“ umbenennen und im Folgenden darauf aufsetzen.

6.2.2 Datenbank-Connection

Zur Speicherung des Users in der Datenbank benötigen wir eine Datenbankverbindung (Connection). Wir erzeugen diese mit unserem JDBC-Framework (vgl. 3.2.1). Trotzdem bleiben mehrere Möglichkeiten zur Implementierung in der UserBean:

- Jede Datenbankmethode könnte ihre eigene Verbindung aufbauen. Dadurch wären die einzelnen Methoden voneinander unabhängig – was gut wäre – das System müsste aber mehrere Datenbankverbindungen verwalten – was bei einem Benutzer nicht schlimm ist, bei vielen Benutzern aber auf die Performance schlagen kann.
- Wir erzeugen eine Datenbankverbindung im Konstruktor der UserBean und speichern diese als zusätzliches Attribut. Dann muss nur eine Verbindung pro UserBean verwaltet werden, und die einzelnen Methoden können einfach darauf zugreifen. Nachteil könnte sein, dass die Verbindung evtl. wegen Timeout geschlossen wird, wenn das Objekt zu lange ohne Datenbankzugriff lebt.
- Wir erzeugen nur eine Datenbankverbindung für alle UserBean-Objekte gemeinsam. Dies wird in der Praxis wohl der effektivste Weg sein, erfordert aber eine Konfiguration des Frameworks, was nicht Thema dieser Veranstaltung ist.

Wir verwenden die zweite Möglichkeit und implementieren ein Attribut dbConn vom Typ `java.sql.Connection` in unserer UserBean:

The screenshot shows a Java code editor with the following code snippet:

```
3 public class UserBean {  
4     String userid;  
5     String password;  
6     String active;  
7     String admin;  
8     String username;  
9     String email;  
10    Connection dbConn;  
11 }
```

A code completion dropdown is open at the cursor position (line 10, character 11), showing suggestions for 'Connection'. The suggestion 'Import 'Connection' (java.sql)' is highlighted. Other suggestions include imports from CORBA and JNDI, and options to create a class or interface.

Achtung: Bei der automatischen Erzeugung des import-Statements muss darauf geachtet werden, dass tatsächlich die `java.sql.Connection` ausgewählt wird. CORBA-Connections können kein JDBC!

Ab hier besitzt das UserBean Objekt eine Variable, die eine Datenbankverbindung speichern kann. Bis jetzt ist diese Variable aber null, d.h. wir haben noch keine Datenbankverbindung. Diese erzeugen wir deshalb im Konstruktor der UserBean:

```

public UserBean() throws NoConnectionException{
    this.dbConn = new PostgreSQLAccess().getConnection();
}

```

Achtung: PostgreSQLAccess muss importiert werden und die NoConnectionException muss geworfen werden.

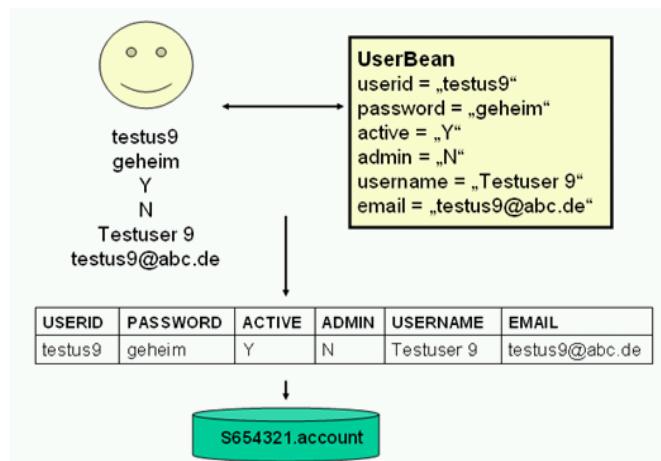
Ab hier besitzt das UserBean Objekt eine Variable dbConn, die ihrerseits eine (im Idealfall funktionierende) Datenbankverbindung enthält. Falls die Datenbankverbindung wegen Timeout geschlossen wird, muss die Anwendung neu gestartet werden, weil wir derzeit keine Funktionalität zur Wiederherstellung der Verbindung eingebaut haben.

6.2.3 Speichern eines Users mit insertUserNoCheck()

Die insert-Methode

Die UserBean steht als elektronisches Modell eines bestimmten Users, nämlich des Users, dessen Werte sie enthält. Wenn z.B. das Attribut userid eines bestimmten UserBean-Objekts den Wert „testus9“ hat, dann steht dieses Objekt stellvertretend für den User mit der Id „testus9“.

Ein insert-Befehl, der an das Objekt gerichtet wird, entspricht also dem Befehl an den User, „sich selbst in der Datenbank zu speichern“. Elektronisch umgesetzt bedeutet das, in der Datenbank einen Datensatz einzufügen, der die Daten enthält, die in den Attributen des UserBean-Objekts gespeichert sind:



Zum Einfügen eines User in die Datenbank benötigen wir einen „insert“-Befehl. Es gibt verschiedene Ergebnis- bzw. Fehlermöglichkeiten:

- Das Einfügen könnte problemlos klappen
- Die Verbindung mit der Datenbank könnte fehlgeschlagen oder zeitlich abgelaufen sein.
- Unser SQL-Statement könnte fehlerhaft sein
- Der User könnte in der Datenbank schon existieren
- Die eingetragenen Felder könnten mit der Datenbank nicht zusammenpassen
- Es könnte ein (unbekannter) Fehler auftreten, den wir nicht erwartet haben

In diesem Abschnitt möchten wir auf die Fehlermöglichkeiten gar nicht eingehen, sondern führen den insert einfach durch und werfen ggfs. eine Exception. Die entsprechende Methode braucht deshalb nicht einmal einen Rückgabewert, wir nennen die Methode wie folgt:

Methode	Anmerkung
void insertUserNoCheck(){...}	

```

public void insertUserNoCheck(){
}

```

Zur Durchführung der Methode brauchen wir dann wie im JDBC üblich:

- Einen SQL-String
- Eine Datenbankverbindung – die wir bereits in 6.2.2 erzeugt haben
- Ein PreparedStatement
- Die Daten, um das PreparedStatement zu vervollständigen
- Einen executeUpdate-Befehl
- Das Exception-Handling

Das insert-Statement

Der SQL-String für das PreparedStatement ist derselbe wie in 3.2.3:

```
public void insertUserNoCheck(){  
    String sql = "INSERT INTO account " +  
        "(USERID, PASSWORD, ACTIVE, ADMIN, USERNAME, EMAIL) " +  
        "VALUES (?, ?, ?, ?, ?, ?);"  
    System.out.println(sql);  
}
```

Datenbankverbindung und PreparedStatement

Die Datenbankverbindung haben wir bereits in 6.2.2 erzeugt und in der Variablen dbConn gespeichert. Hieraus erzeugen wir das PreparedStatement wiederum wie in 3.2.3:

```
public void insertUserNoCheck() throws SQLException{  
    String sql = "INSERT INTO account " +  
        "(USERID, PASSWORD, ACTIVE, ADMIN, USERNAME, EMAIL) " +  
        "VALUES (?, ?, ?, ?, ?, ?);"  
    System.out.println(sql);  
    PreparedStatement myStat = this.dbConn.prepareStatement(sql);  
}
```

Achtung: Die Klasse java.sql.PreparedStatement muss importiert werden und beim Erzeugen des PreparedStatement kann eine SQLException auftreten, die wir werfen.

Setzen der Parameter

Das Setzen der Parameter stellt den wesentlichen Unterschied zu den Methoden aus 3.2.3 dar: Die Parameter werden nicht statisch eingetragen, sondern aus den Attributen des UserBean-Objekts übernommen. Da die UserBean einen User der realen Welt mit dessen Eigenschaften repräsentiert, ist es sinnvoll, diese Eigenschaften aus den Attributen in die Datenbank zu schreiben:

```
public void insertUserNoCheck() throws SQLException{  
    String sql = "INSERT INTO account " +  
        "(USERID, PASSWORD, ACTIVE, ADMIN, USERNAME, EMAIL) " +  
        "VALUES (?, ?, ?, ?, ?, ?);"  
    System.out.println(sql);  
    PreparedStatement myStat = this.dbConn.prepareStatement(sql);  
    myStat.setString(1, this.getUserid());  
    myStat.setString(2, this.getPassword());  
    myStat.setString(3, this.getActive());  
    myStat.setString(4, this.getAdmin());  
    myStat.setString(5, this.getUsername());  
    myStat.setString(6, this.getEmail());  
}
```

Hierbei könnte ebenfalls jeweils eine SQLException auftreten, diese wird aber durch die bereits vorhandene throws-Klausel behandelt.

Ausführung des Statements

Die Ausführung des PreparedStatements erfolgt wieder wie in 3.2.3:

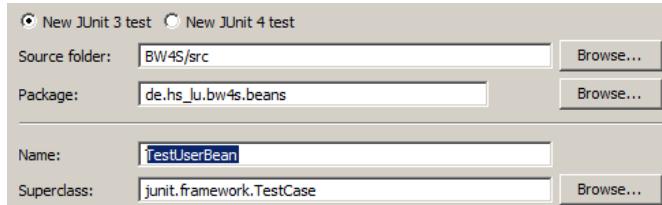
```
    myStat.setString(1, this.getUserid());  
    myStat.setString(6, this.getEmail());  
    myStat.executeUpdate();  
}
```

6.2.4 Testen der insertUserNoCheck()

Die JUnit-Testklasse

Zum Test von Java-Klassen und -Methoden dient üblicherweise das JUnit-Framework. Wir verwenden hier die Version JUnit 3.

Zunächst legen wir im selben Package wie die UserBean eine Testklasse TestUserBean an mit New -> Other... -> Java -> JUnit -> JUnit Testcase



Wir bestätigen, dass die JUnit 3 Library in den Build Path eingebunden werden soll. Ggf. prüfen wir, ob JUnit3 eingebunden ist mit

Rechtsklick auf das Projekt -> Properties -> Java Build Path -> Libraries.

Falls dies nicht der Fall ist, können wir die Library von Hand einbinden mit

Rechtsklick auf das Projekt -> Properties -> Java Build Path -> Libraries -> Add Library -> JUnit -> JUnit 3.

Die setUp()-Methode

Datenbanktests sind nicht ganz einfach, weil für reproduzierbare Tests der Datenbestand innerhalb der Tabellen vergleichbar sein muss. Wir setzen unseren Test auf einer leeren Datenbank-Tabelle account auf. Dafür verwenden wir die folgende Methode in unserer Testklasse TestUserBean:

```
protected void setUp() throws Exception{
    AppInstallAccountTable myApp = new AppInstallAccountTable();
    myApp.dbConn = new PostgreSQLAccess().getConnection();
    try{myApp.dropAccountTable();}catch(Exception e){}
    myApp.createAccountTable();
}
```

Wir verwenden dabei Methoden aus unserer AppInstallAccountTable aus 3.2.3.

Testmethode testInsertUserNoCheck()

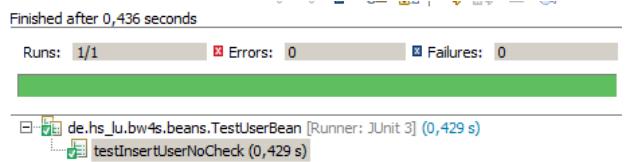
Um unsere insert-Methode zu testen, brauchen wir nun ein UserBean-Objekt mit bekannten Attributwerten, dem wir die insert-Methode schicken können:

```
public void testInsertUserNoCheck() throws Exception{
    UserBean user = new UserBean();
    user.setUserid("testus9");
    user.setPassword("geheim");
    user.setActive("Y");
    user.setAdmin("N");
    user.setUsername("Testuser 9");
    user.setEmail("testus9@abc.de");
    user.insertUserNoCheck();
}
```

Im Prinzip handelt es sich hier nicht um einen JUnit-Test im engeren Sinne, da kein assert-Statement vorhanden ist. Andererseits hat insertUserNoCheck() keinen Rückgabewert, der getestet werden könnte. Wir können hier also nur testen, dass keine Exception auftritt. Für einen schöneren Test brauchen wir die Methode checkUserExists() aus 6.2.5

Ausführung der Testmethode

Eine Ausführung der dargestellten Testmethode funktioniert tatsächlich:



Ein Blick in die Datenbank zeigt, dass der User tatsächlich angelegt wurde:

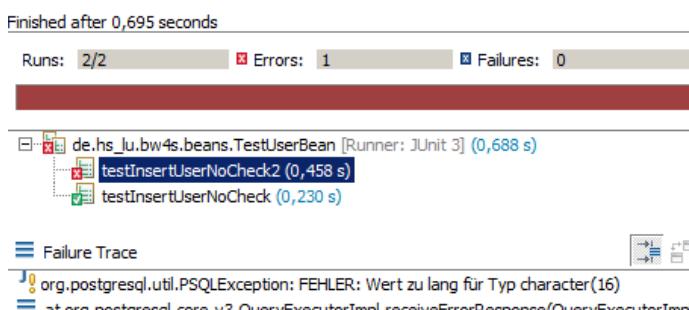
	userid [PK] character(16)	password character(32)	active character(1)	admin character(1)	username character varying(256)	email character varying(256)
1	testus9	geheim	Y	N	Testuser 9	testus9@abc.de

Verschärfter Test

Für einen zweiten Test kopieren wir die oben dargestellte Testmethode und passen diese an. Insbesondere verwenden wir eine userid mit Überlänge, denn in der Tabelle account ist userid nur mit 16 Zeichen definiert:

```
public void testInsertUserNoCheck2() throws Exception{
    UserBean user = new UserBean();
    user.setUserId("testus10testus10testus10");
    user.setPassword("geheim");
    user.setActive("Y");
    user.setAdmin("N");
    user.setUsername("Testuser 10");
    user.setEmail("testus10@abc.de");
    user.insertUserNoCheck();
}
```

Erwartungsgemäß schlägt der Test fehl:



6.2.5 Die Methode checkUserExists()

Ein User (bzw. eine UserBean) sollte in der Lage sein, zu prüfen, ob es ihn in der Datenbank schon gibt, d.h. ob dort ein Datensatz mit seiner userid gespeichert ist. Eine solche Prüfmethode kann z.B. vor dem insert sinnvoll sein, um einen Datenbankfehler zu vermeiden oder nach dem insert, um zu prüfen, ob der Test geklappt hat. Wir verwenden dafür die folgende Methode:

Methode	Anmerkung
boolean checkUserExists(){...}	Rückgabewert ist true, falls der User schon existiert, andernfalls false

Wir suchen zunächst mit einem SQL-Statement in der Datenbank nach einem Datensatz, der dieselbe userid hat wie die UserBean, die die Methode ausführt. Anstelle der Methode executeUpdate() verwenden wir executeQuery() und erhalten zunächst ein ResultSet, das wir weiter untersuchen müssen. Außerdem müssen wir die Klasse java.sql.ResultSet importieren und eine eventuelle SQLException werfen:

```
public boolean checkUserExists() throws SQLException{
    String sql = "SELECT userid FROM account WHERE userid = ?";
    System.out.println(sql);
    PreparedStatement myStat = this.dbConn.prepareStatement(sql);
    myStat.setString(1, this.getUserId());
    ResultSet dbRes = myStat.executeQuery();

    Import 'ResultSet' (java.sql)
    Create class 'ResultSet'
    Create interface 'ResultSet'
    Change to 'Reset' (com.sun.org.glassfish.external.statics)
    Change to 'Reset' (com.sun.org.glassfish.external.statics)
    ...
    import java.sql.Connection;
    import java.sql.PreparedStatement;
    import java.sql.ResultSet;
    import java.sql.SQLException;
    ...
}
```

Da wir gezielt nach einer bestimmten userid gesucht haben, kann das ResultSet genau einen oder gar keinen Eintrag haben, was wir mit next() herausfinden können. Zur Erinnerung: Ein ResultSet besitzt einen internen Pointer, der zu Beginn vor dem ersten Datensatz positioniert ist.

userid	User existiert	User existiert nicht
ResultSet dbRes	Ein Eintrag	Kein Eintrag
dbRes.next()	Findet den Eintrag und liefert true	Findet keinen Eintrag und liefert false
Rückgabewert der Methode	true, weil der User ja existiert	false, weil der User nicht existiert

Wir erkennen, dass der Rückgabewert von dbRes.next() dem Rückgabewert der Methode entspricht. Demnach können wir den Rückgabewert anstatt mit einer if-Schleife direkt als dbRes.next() zurückgeben:

```
public boolean checkUserExists() throws SQLException{
    String sql = "SELECT userid FROM account WHERE userid = ?";
    System.out.println(sql);
    PreparedStatement myStat = this.dbConn.prepareStatement(sql);
    myStat.setString(1, this.getUserid());
    ResultSet dbRes = myStat.executeQuery();
    // if (dbRes.next()){
    //     return true;
    // }else{
    //     return false;
    // }
    return dbRes.next();
}
```

Für die Methode checkUserExists() wären auch mindestens zwei Varianten denkbar:

- Mit dem SQL-Statement „SELECT userid FROM account“.
 - Hierbei entsteht ein ResultSet mit mehr als einem Eintrag, das mit einer while-Schleife durchlaufen und geprüft werden kann.
 - Achtung: Die userids aus ResultSet sind 16 Zeichen lang und sind ggfs. mit Leerzeichen aufgefüllt. Diese Leerzeichen müssen vor dem Stringvergleich mit trim() entfernt werden.
- Mit dem SQL-Statement „SELECT COUNT(*) FROM account WHERE userid = ?“.
 - Hierbei ist zu beachten, dass das ResultSet immer genau eine Ergebnissezeile hat, in der die Anzahl der gefundenen Datensätze steht.
 - Diese Anzahl kann 1 (User wurde gefunden) oder 0 (User wurde nicht gefunden) sein.

Beide Methoden sollten als Übungsaufgabe implementiert werden.

6.2.6 Die Hilfsmethode prepareAttributesForDB()

Im verschärften Test in 6.2.4 war uns aufgefallen, dass die Strings in den Attributen der UserBean nicht länger sein dürfen als die Felder der Datenbank. Wenn wir dies programmtechnisch abfangen möchten, haben wir zwei Möglichkeiten:

- Wir können die Felder passend abschneiden oder
- wir können den insert abbrechen

Wir wählen in diesem Fall die erste Variante und überlegen uns zunächst, dass es noch andere unerwünschte Fälle gibt:

Unerlaubter Fall	Betrifft Felder	Korrektur
Wert = null	Alle	bereits im Kontruktor der UserBean USERID, PASSWORD, USERNAME, EMAIL: auf <leerer String> setzen ACTIVE auf ‚Y‘ setzen ADMIN auf ‚N‘ setzen
Wert zu lang	Alle	Abschneiden: USERID → 16 Zeichen PASSWORD → 32 Zeichen

		USERNAME → 256 Zeichen EMAIL → 256 Zeichen ACTIVE → Auf „N“ setzen ADMIN → Auf „N“ setzen
Wert ungleich „Y“, „N“, „y“, „n“	ACTIVE, ADMIN	ACTIVE, ADMIN: „Y“, „YES“, „J“, „Ja“ auf „Y“ setzen, alles andere auf „N“ setzen

Wir führen die erste Variante durch, indem wir den Konstruktor der UserBean anpassen und außerdem die folgende Methode bereitstellen:

Methode	Anmerkung
void prepareAttributesForDB()	

Da die genannten Anpassungen reines Java sind, sparen wir uns die Erklärungen und zeigen gleich die fertigen Methoden:

```
public UserBean() throws NoConnectionException{
    this.dbConn = new PostgreSQLAccess().getConnection();
    this.userid = "";
    this.password = "";
    this.active = "Y";
    this.admin = "N";
    this.username = "";
    this.email = "";
}

public void prepareAttributesForDB(){
    if (userid.length() > 16) userid = userid.substring(0,16);
    if (password.length() > 32) password = password.substring(0,32);
    if (username.length() > 256) username = username.substring(0,256);
    if (email.length() > 256) email = email.substring(0,256);
    if (active.equalsIgnoreCase("Y"))
        || active.equalsIgnoreCase("YES")
        || active.equalsIgnoreCase("J")
        || active.equalsIgnoreCase("JA")) active = "Y";
    else active = "N";
    if (admin.equalsIgnoreCase("Y"))
        || admin.equalsIgnoreCase("YES")
        || admin.equalsIgnoreCase("J")
        || admin.equalsIgnoreCase("JA")) admin = "Y";
    else admin = "N";
}
```

6.2.7 Speichern eines Users mit insertUserIfNotExists()

In 6.2.3 haben wir ausdrücklich auf Prüfungen vor der Ausführung des SQL-Statements verzichtet. In diesem Abschnitt möchten wir eine „schönere“ insert-Methode implementieren. Schön heißt in diesem Zusammenhang, dass wir vor der Ausführung des insert-Statements

- die Länge der Attribute an die Datenbank anpassen,
- prüfen, ob ein User mit dieser Userid bereits existiert.

Alle anderen möglichen Fehler werden weiterhin in einer Exception gekapselt und geworfen.

Wir verwenden hierfür die folgende Methode:

Methode	Anmerkung
boolean insertUserIfNotExists(){...}	Rückgabewert ist true, wenn der User noch nicht existiert hat und erfolgreich angelegt wurde, andernfalls false.

Der Ablauf der Methode ist der Folgende:

- Im Gegensatz zur insertUserNoCheck() prüfen und ggfs. korrigieren wir zuerst mit prepareAttributesForDB() die Attribute
- Anschließend prüfen wir mit checkUserExists(), ob der User bereits existiert
- Falls er noch nicht existiert, können wir insertUserNoCheck() aufrufen, um die eigentliche Arbeit des Anlegens erledigen zu lassen.
- Den Rückgabewert berücksichtigen wir im entsprechenden Zusammenhang

Für die Implementierung der Methode sind z.B. die folgenden Varianten denkbar:

```
public boolean insertUserIfNotExists() throws SQLException{
    this.prepareStatementForDB();
    if (!this.checkUserExists()){
        this.insertUserNoCheck();
        return true;
    } else return false;
}

public boolean insertUserIfNotExists2() throws SQLException{
    this.prepareStatementForDB();
    boolean userExists = this.checkUserExists();
    if (!userExists) this.insertUserNoCheck();
    return !userExists;
}
```

Wenn die Methode tatsächlich mit einem Booleschen Rückgabewert endet, bedeutet dies auch, dass keine Exception aufgetreten ist, dass der Code also wie erwartet ausgeführt wurde.

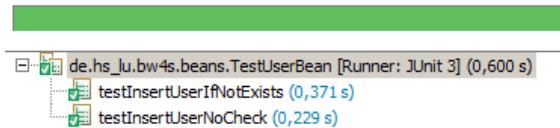
6.2.8 Testen der insertUserIfNotExists()

Zum Test der Methode insertUserIfNotExists() verwenden wir in der Klasse TestUserBean eine Testmethode testInsertUserIfNotExists(), die genau so beginnt wie die testInsertUserNoCheck(), aber am Ende zwei assert-Statements besitzt, um zu testen

- dass insertUserIfNotExists() true zurückgibt und
- dass der User danach tatsächlich existiert

```
public void testInsertUserIfNotExists() throws Exception{
    UserBean user = new UserBean();
    user.setUserId("testus10");
    user.setPassword("geheim");
    user.setActive("Y");
    user.setAdmin("N");
    user.setUsername("Testuser 10");
    user.setEmail("testus10@abc.de");
    assertTrue(user.insertUserIfNotExists());
    assertTrue(user.checkUserExists());
}
```

Die Ausführung der Testklasse liefert



6.3 Praxis: Benutzerregistrierung in der RegAppl.jsp

6.3.1 Bisheriger Stand

In unserer Webanwendung haben wir bisher zwei JSPs:

- RegView.jsp enthält ein Formular, das die Texteingaben für userid, password, username, email und die Buttons zumlogin und register an die RegAppl.jsp schickt.
- RegAppl.jsp enthält nur Java-Code, der die Parameter annimmt, die Buttons auf null checkt, und eine Aktionsweiche implementiert:

```
if (zumlogin.equals("Zum Login")){
    // Aktionscode hier

    response.sendRedirect("./LoginAppl.jsp");
} else if (register.equals("Registrieren")){
    // Aktionscode hier

    response.sendRedirect("./RegView.jsp");
} else{
    // Aktionscode hier

    response.sendRedirect("./RegView.jsp");
}
```

6.3.2 Registrieraktion nach MVC

Übersicht

Nach der MVC-Architektur führt die RegAppl.jsp, die unseren Controller darstellt, keine Aktion selbst aus, sondern delegiert diese an das Model, in unserem Fall die UserBean. Dafür muss die RegAppl.jsp

- Eine UserBean anlegen. Hier kommt der Konstruktor ohne Parameter zum Einsatz
- Der UserBean die notwendigen Daten mitgeben
- Die Aktion der UserBean aufrufen
- Ggf. Rückgabewerte behandeln und weitere Nacharbeiten durchführen

Erzeugen und importieren der UserBean

Beim Erzeugen der UserBean taucht die folgende Fehlermeldung auf:

```
else if (register.equals("Registrieren")){
    UserBean user = new UserBean();
}
UserBean cannot be resolved to a type
Press 'F2' for focus
```

Die Fehlermeldung ist aus Java bekannt und bedeutet, dass wir die Klasse UserBean importieren müssen. Hierfür haben wir die Möglichkeit von Hand an den Anfang der JSP die JSP-Direktive `<%@page import="de.hs_lu.bw4s.beans.UserBean"%>` zu schreiben:

```
UserBean.java *RegAppl.jsp
1 <%@page import="de.hs_lu.bw4s.beans.UserBean"%>
2 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
3     pageEncoding="ISO-8859-1"%>
4 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://
5<html>
6</head>
```

Alternativ haben wir die Möglichkeit, dies von Eclipse per Code-Vervollständigung erledigen zu lassen: Steht der Cursor auf dem Klassennamen, erhalten wir mit Ctrl-<Leertaste> einen Vorschlag zur Vervollständigung des Klassennamens, der außerdem automatisch die oben dargestellte import-Direktive am Seitenanfang einfügt:

```
else if (register.equals("Registrieren")){
    UserBean user = new UserBean();
}
UserBean - de.hs_lu.bw4s.beans
```

Natürlich ist es sinnvoll, zu prüfen, ob das import-Statement tatsächlich angelegt wurde und ob es die richtige Klasse importiert hat.

Eine weitere Alternative wäre gewesen, die UserBean voll zu qualifizieren:
`de.hs_lu.bw4s.beans.UserBean user = new de.hs_lu.bw4s.beans.UserBean();`

Restliche Arbeiten

Die fehlenden Schritte sind unspektakulär und so wie oben dargestellt:

```
if (zumlogin.equals("Zum Login")){
    // Aktionscode hier

    response.sendRedirect("./LoginAppl.jsp");
} else if (register.equals("Registrieren")){
    UserBean user = new UserBean();
    user.setUserid(userid);
    user.setPassword(password);
    user.setUsername(username);
    user.setEmail(email);
    user.insertUserIfNotExists();

    response.sendRedirect("./RegView.jsp");
} else{
    // Aktionscode hier
    response.sendRedirect("./RegView.jsp");
}
```

Eine Auswertung des insert-Befehls führen wir zum jetzigen Zeitpunkt nicht durch.

6.3.3 Test und Reflektion

Test

Um die Funktionalität bis hierhin zu testen, starten wir die RegView im TomCat, geben sinnvolle Werte ein und klicken auf Registrieren:

Nickname: huber
Passwort: *****
Username: Fritz Huber
E-Mail: fritz.huber@abc.de
Zum Login Registrieren

Am Ergebnis können wir bisher noch nicht erkennen, ob das geklappt hat:

Nickname:
Passwort:
Username:
E-Mail:
Zum Login Registrieren

Wir schauen deshalb mit pgAdmin3 direkt in die Tabelle account der Datenbank BWUEBDB:

	userid [PK] character(16)	password character(32)	active character(1)	admin character(1)	username character varying(256)	email character varying(256)
1	huber	geheim	Y	N	Fritz Huber	fh@abc
2	testus9	geheim	Y	N	Testuser 9	testus

Build Path und Deployment Assembly

Falls beim Versuch, auf die Datenbank zuzugreifen, eine FileNotFoundException oder ClassNotFoundException auftritt, fehlt sehr wahrscheinlich der PostgreSQL-JDBC-Treiber im Build Path oder in der Deployment Assembly.

In diesem Fall führen wir die entsprechende Installation / Konfiguration wie im Abschnitt „JDBC-Treiber für PostgreSQL aus 3.2.1 noch einmal durch.

Reflektion

Wir haben jetzt zum ersten Mal sämtliche Schichten des MVC-Architektur erfolgreich durchlaufen.

Zwar sind offensichtlich noch zahlreiche Verbesserungen nötig, dennoch haben wir die grundsätzliche Funktionalität erreicht und damit auch die Machbarkeit nachgewiesen.

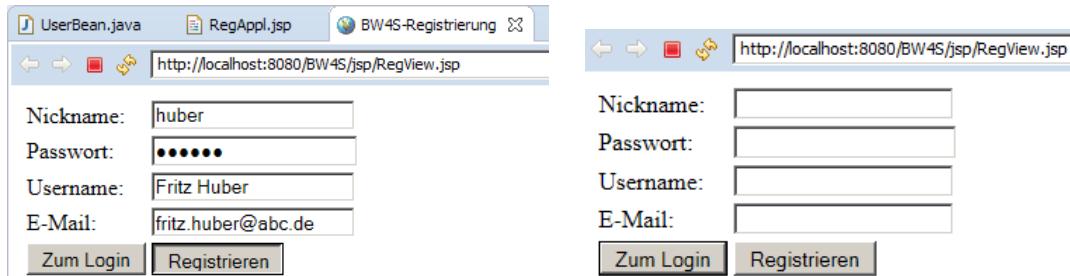
7 Sessions

7.1 Theorie

Die theoretischen Grundlagen zu Sessions finden Sie im Powerpoint-Foliensatz SE2-V07-JSP-5.ppt

7.2 Praxis: Wiederanzeige der eingegebenen Benutzerdaten

Im Test von Abschnitt 6.3.3 haben wir gesehen, dass die Daten in der RegView.jsp nicht mehr angezeigt werden, nachdem einmal die RegAppl.jsp aufgerufen und anschließend per Weiterleitung zur RegView.jsp zurückgekehrt wurde:

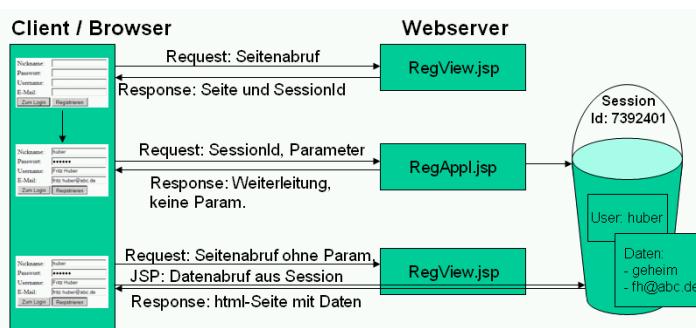


Insbesondere wenn das Anlegen des Users nicht geklappt hat, wäre es wünschenswert, die Daten noch im Formular stehen zu haben, um diese für den nächsten Versuch einfach anpassen zu können.

Ziel dieses Abschnitts wird es deshalb sein, die Daten mittels Session von der RegAppl.jsp zur RegView.jsp zurückzutransportieren und dort anzuzeigen.

7.2.1 Übersicht

In der folgenden Abbildung wird das Grundprinzip verdeutlicht:



- Zu Beginn wird die RegView.jsp leer abgerufen und an den Benutzer ausgeliefert.
- Zeitgleich wird auf dem Server eine bis dahin noch leere Session angelegt und die Session-Id wird mit der html-Seite an den Benutzer übertragen.
- Der Benutzer/die Benutzerin trägt die Daten ein und überträgt diese als Request-Parameter an die RegAppl.jsp.
- Die RegAppl.jsp verarbeitet die Daten und trägt diese u.a. in die Session ein. Durch Weiterleitung (response.sendRedirect(...)) wird wieder die RegView.jsp aufgerufen, allerdings ohne übertragene Daten.
- Die RegView.jsp sollte sich jetzt per JSP die Daten aus der Session holen und diese in ihre eigene Darstellung einbauen.

In diesem Abschnitt sind deshalb zwei Aktionen durchzuführen:

- Speicherung der Daten in der Session durch die RegAppl.jsp.
- Auslesen und der Daten aus der Session und Anzeige durch die RegView.jsp.

Anschließend werden wir den Code der RegAppl.jsp noch anpassen, um diesen mit dem Code in der RegView.jsp zu vereinheitlichen.

7.2.2 Datenspeicherung in der RegAppl.jsp

Grundsätzlich könnten die Parameter einzeln in der Session gespeichert werden, da sie aber bereits in der Anwendung in Form einer Bean gekapselt sind, können sie sehr einfach als ganze Bean gespeichert werden. Hier ist es von Vorteil, dass in der Session nicht nur Strings, sondern beliebige Objekte gespeichert werden können.

Zunächst speichern wir die UserBean in der RegAppl.jsp an der Stelle wo sie entsteht und mit Daten gefüllt wird:

```
else if (register.equals("Registrieren")){
    UserBean user = new UserBean();
    user.setUserId(userid);
    user.setPassword(password);
    user.setUsername(username);
    user.setEmail(email);

    session.setAttribute("user", user);

    user.insertUserIfNotExists();
    response.sendRedirect("./RegView.jsp");
} else{
```

Beobachtungen:

- Zur Speicherung wird ein Name benötigt, unter dem die Bean in der Session später wiederzufinden ist.
- Hier haben wir denselben Namen verwendet, den die Bean auch als Java-Objekt besitzt. In 8.1.4 werden wir herausarbeiten, warum das so sein muss.

7.2.3 Daten lesen in der RegView.jsp

Einfaches Lesen

Um die Bean in der RegView.jsp einzulesen, benötigen wir Java-Code:

- Wir können diesen direkt am Beginn des body einfügen.
- Zunächst lesen wir die UserBean mit session.getAttribute(...) unter dem in 7.2.2 vergebenen Namen in eine Java-Variable desselben Namens.
- Wir erinnern uns, dass session.getAttribute(...) beliebige Objekte zurückgeben kann und wir deshalb casten müssen.
- Außerdem müssen wir die Klasse UserBean importieren, was am einfachsten mit der Auto vervollständigung von Eclipse geht, vgl. 6.3.2:



The screenshot shows a portion of a JSP file in Eclipse. A cursor is at the end of the line: '%> UserBean user = (UserBean) session.getAttribute("user");'. A tooltip from the Java code completion feature is displayed, listing three options: 'UserBean - de.hs_lu.bw4s.beans', 'UserBeanEmpty - de.hs_lu.bw4s.beans', and 'UserBeanEmpty - de.hs_lu.bw4s.beans'. The first option is highlighted with a green circle icon.

null-check

Falls die RegView.jsp vor der RegAppl.jsp aufgerufen wird, wäre die UserBean noch nicht gesetzt. session.getAttribute(...) würde in diesem Fall null an die Variable user zurückgeben. Dies wäre aber schlecht, weil wir einige Zeilen später auf user zugreifen müssen.

Aus diesem Grund fügen wir Code hinzu, der verhindert, dass die Variable user == null bleibt. Stattdessen soll in dem Fall eine neue, leere UserBean erzeugt, der Variable user zugewiesen und in der Session verankert werden:

```

<body>
<%
UserBean user = (UserBean) session.getAttribute("user");
if (user == null){
    user = new UserBean();
    session.setAttribute("user", user);
}
%>
<form action=".\\jsp\\RegAppl.jsp" method="get">
    <table>

```

7.2.4 Daten anzeigen in der RegView.jsp

Die Benutzereingabe für z.B. das input-Feld „userid“ wird als value des Parameters userid übertragen. Umgekehrt kann der value des input-Tags gesetzt werden und wird dann auf der Oberfläche angezeigt.

Wenn wir also den Wert „fritz“ im input-Feld anzeigen möchten, setzen wir value=“fritz“. Da der Wert aber variabel ist und im Attribut userid der UserBean in der Session steht, müssen wir mit JSP auf diese zugreifen:

```

<form action=".\\jsp\\RegAppl.jsp" method="get">
    <table>
        <tr>
            <td>Nickname:</td>
            <td><input type="text" name="userid" value="<% user.getUserid() %>" /></td>
        </tr>
        <tr>

```

Dasselbe führen wir für die Felder username und email durch. Für das Passwort setzen wir keine Anzeige, weil diese an der Oberfläche nicht sichtbar ist und eher für Verwirrung sorgen würde:

```

<body>
<%
UserBean user = (UserBean) session.getAttribute("user");
if (user == null){
    user = new UserBean();
    session.setAttribute("user", user);
}
%>
<form action=".\\jsp\\RegAppl.jsp" method="get">
    <table>
        <tr>
            <td>Nickname:</td>
            <td><input type="text" name="userid" value="<% user.getUserid() %>" /></td>
        </tr>
        <tr>
            <td>Passwort:</td>
            <td><input type="password" name="password" value="" /></td>
        </tr>
        <tr>
            <td>Username:</td>
            <td><input type="text" name="username" value="<% user.getUsername() %>" /></td>
        </tr>
        <tr>
            <td>E-Mail:</td>
            <td><input type="text" name="email" value="<% user.getEmail() %>" /></td>
        </tr>
        <tr>
            <td><input type="submit" name="zumlogin" value="Zum Login" /></td>
            <td><input type="submit" name="register" value="Registrieren" /></td>
        </tr>
    </table>
</form>
</body>

```

Beobachtung: Eclipse erkennt sogar Fehler, wenn wir auf Attribute zugreifen, die es in der UserBean nicht gibt:  `<td><input type="text" name="email" value="<% user.getEmail() %>" /></td>`

7.2.5 Code verschönern in der RegAppl.jsp

UserBean-Leseversuch

Wenn über die RegView.jsp in die RegAppl.jsp gesprungen wird, liegt in der Session schon eine UserBean unter dem Namen „user“ vor und es wäre sinnvoll, diese wiederzuverwenden, anstatt eine neue anzulegen. Deshalb sollte der Code in der RegAppl.jsp ebenfalls einen Leseversuch unternehmen:

```

}else if (register.equals("Registrieren")){
    UserBean user = (UserBean) session.getAttribute("user");
    if (user == null){
        user = new UserBean();
        session.setAttribute("user", user);
    }
    user.setUserid(userid);
    user.setPassword(password);
    user.setUsername(username);
    user.setEmail(email);

//    session.setAttribute("user", user);

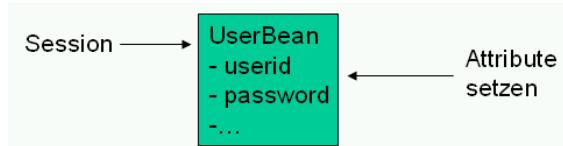
    user.insertUserIfNotExists();
    response.sendRedirect("./RegView.jsp");
}else{

```

Reihenfolge der Befehle

Der wesentliche Unterschied zum vorigen Code (der im Listing noch auskommentiert steht) ist, dass `session.setAttribute("user", user)` jetzt ausgeführt wird, bevor der UserBean Werte zugewiesen werden.

Da das UserBean-Objekt als Ganzes und dessen Verankerung in der Session aber unverändert bleibt und sich durch die Zuweisung der Werte eben nur die Attribute ändern, ist es egal, ob die Bean vor oder nach der Zuweisung in die Session geschrieben wird.



Aus demselben Grund ist es sogar möglich, den Bean-Code ganz an den Anfang der JSP zu setzen und diesen aus der Aktionsweiche zu entfernen:

```

<body>
<%
UserBean user = (UserBean) session.getAttribute("user");
if (user == null){
    user = new UserBean();
    session.setAttribute("user", user);
}

String userid = request.getParameter("userid");
String password = request.getParameter("password");
String username = request.getParameter("username");
String email = request.getParameter("email");
String zumlogin = request.getParameter("zumlogin");
String register = request.getParameter("register");
if (zumlogin == null) zumlogin = "";
if (register == null) register = "";

if (zumlogin.equals("Zum Login")){
    // Aktionscode hier

    response.sendRedirect("./LoginAppl.jsp");
} else if (register.equals("Registrieren")){
    user.setUserid(userid);
    user.setPassword(password);
    user.setUsername(username);
    user.setEmail(email);
    user.insertUserIfNotExists();
    response.sendRedirect("./RegView.jsp");
} else{
    // Aktionscode hier
    response.sendRedirect("./RegView.jsp");
}

```

Dies hat zwei Vorteile:

- Bereits zu Beginn der RegAppl.jsp ist deutlich erkennbar, welche Beans eine Rolle spielen. Da die Beans unser Model darstellen und das Model wiederum die Daten enthält und verarbeitet, wird damit auf den ersten Blick deutlich, welche Datenverarbeitung hier gesteuert wird.
- Der Bean Code ist in der RegView.jsp und der RegAppl.jsp derselbe und wurde damit vereinheitlicht.

Mögliche Seiteneffekte

Da die UserBean über mehrere Aufrufe hinweg dieselbe bleibt, bleiben auch die Attribute jeweils erhalten, solange sie nicht gelöscht oder überschrieben werden.

Bei Aufruf der RegAppl.jsp mit „register“ werden die neuen Werte aus der Eingabe (und damit aus den Request-Parametern) genutzt, um die alten Werte in der Session zu überschreiben. Werte aus Request-Parametern sind also grundsätzlich aktueller als die Werte aus der Session.

Wird die RegAppl.jsp anders aufgerufen als mit „register“, müssen die Attribute der Bean als veraltet betrachtet werden. Bei erneutem Aufruf der RegView.jsp kommen dann diese veralteten Werte wieder an die Oberfläche. Dies müssen wir in Erinnerung behalten, wenn unsere Webanwendung komplexer wird. Eventuell möchten wir dann sogar eine UserBean aus der Session löschen, was mit session.removeAttribute(„user“) möglich ist.

7.2.6 Testen

Der Test der Funktionalität erfolgt durch Aufruf der RegView.jsp mit Tomcat, Eintrag von Werten und Klick auf „Registrieren“. Da die RegAppl.jsp am Ende zurück auf die RegView.jsp weiterleitet, sehen wir dann direkt die eingegebenen Werte wieder mit Ausnahme des Passworts:

The figure consists of three side-by-side screenshots of a web form. Each screenshot shows four input fields: Nickname, Password, Username, and E-Mail, followed by two buttons: 'Zum Login' and 'Registrieren'. In the first screenshot, all fields are empty. In the second screenshot, the Nickname is 'huber', Password is masked as '*****', Username is 'Fritz Huber', and E-Mail is 'fh@abc.de'. In the third screenshot, the Nickname is 'huber', Password is empty, Username is 'Fritz Huber', and E-Mail is 'fh@abc.de'. Arrows between the screenshots indicate the progression of the registration process.

Ob das Erzeugen des Users „huber“ geklappt hat, sehen wir aber wieder nur durch direkten Blick in die Datenbank.

7.3 Praxis: Einführung einer Messaging-Infrastruktur

Für eine brauchbare Benutzersteuerung ist es erforderlich, dass der Benutzer jederzeit informiert wird, ob seine Anweisungen erfolgreich umgesetzt wurden oder nicht. Hierfür stellen wir im Folgenden eine mögliche Lösung vor.

7.3.1 Vorüberlegungen und MVC

Oberfläche

In der RegView.jsp werden zwei Zeilen verwendet, die in der Seite oben stehen und statisch etwa folgendermaßen aussehen:

Willkommen zur BW4S-Registrierung

Bitte registrieren Sie sich

Nickname:

Passwort:

Username:

E-Mail:

```
<body>
<%>
UserBean user = (UserBean) session.getAttribute("user");
if (user == null){
    user = new UserBean();
    session.setAttribute("user", user);
}
%>
<h3>Willkommen zur BW4S-Registrierung</h3>
<h4>Bitte registrieren Sie sich</h4>
<form action=".view/RegAnnl.jsp" method="get">
```

Zur Information der Benutzer werden hier zwei Zeilen verwendet:

- Eine Zeile zur Benutzerinformation, z.B Systemstatus oder Ergebnis einer Aktion
- Eine Zeile, die dem Benutzer eine weitere Aktion vorschlägt

Diese statischen Zeilen sollen mit dem Message-Framework zukünftig dynamisch erzeugt werden, aber nach dem MVC-Prinzip möglichst ohne Code in der View.

MessageBean

Nach 6.1.5 soll die Datenverarbeitung im Model stattfinden, bei uns also in Java-Beans. Die Art der Daten wird durch die Anwendung bestimmt. Da an der Oberfläche eine Benutzerinformation und eine mögliche Aktion angegeben werden soll, muss eine passende MessageBean die Strings infoMessage und actionMessage enthalten.

Systemstatus

Der aktuelle Systemstatus (z.B. erfolgreiche oder fehlgeschlagene Aktion) wird durch die UserBean per Rückgabewert (z.B. boolean) an die RegAppl.jsp zurückgemeldet. Aus diesem Systemstatus kann die Meldung an den Benutzer abgeleitet werden.

Die RegAppl.jsp, die den Controller der Webanwendung darstellt, sollte also in der MessageBean die richtige Message erzeugen und diese über die Session an die RegView.jsp zur Anzeige bereitstellen.

7.3.2 Die MessageBean

Entsprechend der Vorüberlegung entwickeln wir die MessageBean im Package de.hs_lu.bw4s.beans wie folgt. Damit es eine Bean wird, beachten wir die Anforderungen aus 6.1.4, d.h. wir brauchen einen Konstruktor ohne Parameter und get/set-Methoden, die wir uns von Eclipse generieren lassen können:

```
package de.hs_lu.bw4s.beans;

public class MessageBean {
    String infoMessage;
    String actionMessage;

    public MessageBean(){
    }

    public String getInfoMessage() {
        return infoMessage;
    }
    public void setInfoMessage(String infoMessage) {
        this.infoMessage = infoMessage;
    }
}
```

7.3.3 Erste Meldung

Anpassung der MessageBean

Beim Start des Systems wird als Erstes die RegView.jsp mit einem leeren Formular gezeigt und es ist sinnvoll, die in 7.3.1 dargestellten Willkommenszeilen anzuzeigen. Zur Generierung dieser Message erzeugen wir in der MessageBean die folgende Methode:

```
public void setRegistrationWelcome(){
    this.infoMessage = "Willkommen zur BW4S-Registrierung";
    this.actionMessage = "Bitte registrieren Sie sich";
}
```

Und verankern diese im Konstruktor, damit diese direkt zu Beginn wirksam wird:

```
public MessageBean(){
    this.setRegistrationWelcome();
}
```

Anmerkung: Wir werden die Registrierungsfunktion später in ein Portal einbetten, in dem die erste Seite eine Anmeldeseite sein wird. Dann müssen wir diesen Konstruktor noch einmal ändern.

Anpassung der RegView.jsp

Um die MessageBean verwenden zu können, müssen wir diese zunächst in die RegView.jsp einbinden. Dies erfolgt genau wie die Einbindung der UserBean in den folgenden Schritten:

- Kopieren des Java-Fragments zur Einbindung der UserBean
- Anpassung an 9 Stellen: user → message, “user” → “message“ und UserBean → MessageBean
- Importieren der Klasse MessageBean durch Eclipse-Codevervollständigung

```

<body>
<%
UserBean user = (UserBean) session.getAttribute("user");
if (user == null){
    user = new UserBean();
    session.setAttribute("user", user);
}
MessageBean message = (MessageBean) session.getAttribute("message");
if (message == null){
    message = new MessageBean();
    session.setAttribute("message", message);
}
%>
<h3>Willkommen zur BW4S-Registrierung</h3>
<h4>Bitte registrieren Sie sich</h4>

```

Nun kann für die Anzeige der Message ebenfalls JSP verwendet werden, in diesem Fall in Form von JSP-Expressions:

```

MessageBean message = (MessageBean) session.getAttribute("message");
if (message == null){
    message = new MessageBean();
    session.setAttribute("message", message);
}
%>
<h3><%= message.getInfoMessage() %></h3>
<h4><%= message.getActionMessage() %></h4>

```

Test

Wir starten den Tomcat-Server neu (Rechtsklick auf den Server → Restart), um tatsächlich einen Systemstart zu erzwingen. Anschließend führen wir die RegView.jsp aus und erhalten dasselbe Ergebnis wie oben:

Willkommen zur BW4S-Registrierung

Bitte registrieren Sie sich

Nickname:	<input type="text"/>
Passwort:	<input type="password"/>
Username:	<input type="text"/>
E-Mail:	<input type="text"/>
<input type="button" value="Zum Login"/>	<input type="button" value="Registrieren"/>

Weitere Verfeinerung

Für den häufig auftretenden Fall dass infoMessage und actionMessage gemeinsam im html angezeigt werden, können wir diese Ausgabe auch mit einer Methode in der MessageBean zusammenfassen:

```

public String getMessageHtml(){
    String html = "";
    html += "<h3>" + this.infoMessage + "</h3>\n";
    html += "<h4>" + this.actionMessage + "</h4>\n";
    return html;
}

```

In der RegView.jsp ist dann statt 2 Zeilen nur noch eine Zeile zum Aufruf der Funktionalität nötig:

```

MessageBean message = (MessageBean) session.getAttribute("message");
if (message == null){
    message = new MessageBean();
    session.setAttribute("message", message);
}
%>
<%= message.getMessageHtml() %>

```

7.3.4 Weitere Meldungen

Die Meldungen der MessageBean sollen sämtliche Benutzerinformationen abbilden. Diese stehen im engen Zusammenhang mit dem jeweiligen Systemstatus. Im Zusammenhang mit der Registrierungsfunktionalität sind z.B. Rückmeldungen in den nachfolgend aufgeführten Situationen sinnvoll. Für jede der Situationen benötigen wir eine Methode in der MessageBean mit geeignetem Meldungstext:

Situation	Methode	info-/action-Message
Neuer Benutzer wurde erfolgreich registriert	setRegistrationSuccessful()	Benutzer + <Nickname> + wurde erfolgreich registriert Bitte gehen Sie jetzt zur Anmeldung
Benutzer konnte nicht registriert werden, weil der gewählte Nickname bereits vergeben ist	setUserAlreadyExists()	Der Nickname + <Nickname> + existiert bereits. Bitte wählen Sie einen anderen Nickname
Registrierung ist aus unbekannten Gründen fehlgeschlagen	setAnyError()	Es ist ein Fehler aufgetreten Bitte wenden Sie sich an Ihren Systemadministrator

```

public void setRegistrationSuccessful(String userid){
    this.infoMessage = "Benutzer " + userid + " wurde erfolgreich registriert";
    this.actionMessage = "Bitte gehen Sie jetzt zur Anmeldung";
}
public void setUserAlreadyExists(String userid){
    this.infoMessage = "Benutzer " + userid + " existiert bereits";
    this.actionMessage = "Bitte wählen Sie einen anderen Nickname";
}
public void setAnyError(){
    this.infoMessage = "Es ist ein Fehler aufgetreten";
    this.actionMessage = "Bitte wenden Sie sich an Ihren Systemadministrator";
}

```

7.3.5 Anpassungen in der RegAppl.jsp

In der RegAppl.jsp muss zunächst ebenfalls die MessageBean eingebunden werden, analog zur Einbindung in die RegView.jsp in 7.3.3. Der Code kann aus der RegView.jsp kopiert werden. Es ist dann nur noch der Import der Klasse MessageBean nötig:

```

MessageBean message = (MessageBean) session.getAttribute("message");
if (message == null){
    message = new MessageBean();
    session.setAttribute("message", message);
}

```

Das Ergebnis des Registrierungsversuchs wird von der Methode insertUserIfNotExists() zurückgegeben. Die folgenden Ergebnisse entsprechen genau den neu angelegten Meldungen:

- Registrierung erfolgreich – true
- User existiert bereits – false
- Sonstiger Fehler – Exception

Dies implementieren wir innerhalb der Aktionsweiche. Dabei muss SQLException importiert werden, am besten wieder mit der Eclipse-Code-Vervollständigung:

```

}else if (register.equals("Registrieren")){
    user.setUserid(userid);
    user.setPassword(password);
    user.setUsername(username);
    user.setEmail(email);
    try{
        boolean userAngelegt = user.insertUserIfNotExists();
        if (userAngelegt) message.setRegistrationSuccessful(userid);
        else message.setUserAlreadyExists(userid);
    }catch(SQLException se){
        se.printStackTrace();
        message.setAnyError();
    }
    response.sendRedirect("./RegView.jsp");
}else{

```

Wir testen die Funktionalität anhand eines neu anzulegenden Users, den wir gleich zweimal hintereinander zu registrieren versuchen:

Willkommen zur BW4S-Registrierung

Bitte registrieren Sie sich

Nickname:	<input type="text" value="gaby36"/>
Passwort:	<input type="password" value="*****"/>
Username:	<input type="text" value="Gaby Marquardt"/>
E-Mail:	<input type="text" value="gm@abc.de"/>

[Zum Login](#)

[Registrieren](#)

Benutzer gaby36 wurde erfolgreich registriert

Bitte gehen Sie jetzt zur Anmeldung

Nickname:	<input type="text" value="gaby36"/>
Passwort:	<input type="password"/>
Username:	<input type="text" value="Gaby Marquardt"/>
E-Mail:	<input type="text" value="gm@abc.de"/>

[Zum Login](#)

[Registrieren](#)

Benutzer gaby36 existiert bereits

Bitte wählen Sie einen anderen Nickname

Nickname:	<input type="text" value="gaby36"/>
Passwort:	<input type="password"/>
Username:	<input type="text" value="Gaby Marquardt"/>
E-Mail:	<input type="text" value="gm@abc.de"/>

[Zum Login](#)

[Registrieren](#)

7.3.6 Reflektion und Ausblick

Beobachtungen zum Messaging:

- Sämtliche konkreten Meldungstexte finden sich nun ausschließlich in der MessageBean und sind nicht über das System verstreut. Eine Anpassung der Meldungen, z.B. eine Fehlerbehebung, ist damit sehr einfach.
- Falls das System in einer anderen Sprache angeboten werden soll, muss nur die MessageBean ausgetauscht werden.
- Falls die Meldungen in einer externen Datei gespeichert und gepflegt werden sollen, kann diese von der MessageBean eingelesen werden.
- Für Mehrsprachigkeit ist es auch üblich, dynamisch zwischen verschiedenen Klassen oder verschiedenen externen Dateien zu wechseln.
- Der Ausgangspunkt für diese Verbesserungen ist aber die hier erfolgte Zusammenfassung der Meldungen in einer eigenen Klasse.

Beobachtungen zum Stand unserer Webanwendung:

- Die grundlegende Registrierungsfunktionalität haben wir nun fertig. In den nächsten Kapiteln werden wir nur noch kleinere Verschönerungen vornehmen, anhand derer wir auch noch weitere JSP-Mechanismen verdeutlichen werden.
- Anschließend binden wir die Registrierungsfunktionalität in einen größeren Kontext ein und kombinieren die mit einer Login-Funktionalität für ein Portal, in dem wir unterschiedliche Anwendungen verankern könnten.

8 JSP-Tags, -Direktiven, -Scopes und -Objekte

In diesem Kapitel stellen wir einige ergänzende Informationen über JSP zusammen. Insbesondere JSP-Tags werden wir dann in unsere Registrierungsanwendung einbauen.

8.1 Theorie

8.1.1 JSP-Direktiven

Mit JSP-Direktiven kann eine JSP als Ganzes beeinflusst werden, die allgemeine Syntax dafür lautet:

<% @ <typ> <name>=<wert> %>

Die wichtigsten JSP-Direktiven sind:

- Import-Funktion, z.B. <% @ page import="de.hs_lu.bw4s.beans.UserBean" %> oder auch <% @ page import="de.hs_lu.bw4s.beans.UserBean, de.hs_lu.bw4s.beans.MessageBean" %>
- Wenn keine Session zur Verfügung gestellt werden soll, z.B. um Server-Ressourcen zu sparen, kann <% @ page session="false" %> verwendet werden.
- Zur Festlegung des Inhaltstyps bzw. des Zeichensatzes: <% @ page contentType="..." %> bzw. <% @ page contentType="...; charset=..." %>
- Zur Auslagerung von Code <% @ include file ="myAusgelagerterFile.jsp" %>
- Wenn eine Fehlerseite definiert werden soll: <% @ page errorPage="meineFehlerbehandlung.jsp" %>
- Die errorPage sollte dann selbst die Direktive <% @ page isErrorPage="true" %> enthalten, damit die zusätzliche Variable exception zur Verfügung gestellt wird.

Anmerkungen:

- Direktiven werden vor der Erzeugung des HTML-Codes vollständig abgearbeitet.
- Page-Direktiven sollten wegen der Lesbarkeit immer am Anfang einer JSP stehen.
- Page-Direktiven haben ggfs. Auswirkungen auf den Response-Header, z.B. wirkt die Page-Direktive content-type analog zu setHeader("content-type", "...")
- Header müssen bei http immer vor den Daten geschickt werden.
- Spätestens wenn der Ausgabepuffer voll ist, muss der Server senden. Danach darf kein Header mehr kommen.
- Die include-Direktive wird oft als sehr nützlich betrachtet. Wenn html-Code durch Java Beans erzeugt wird (wie z.B. durch die getMessageHtml() in der MessageBean), sind includes aber meistens überflüssig.

8.1.2 JSP-Scopes

JSP unterscheidet verschiedene Geltungsbereiche, so genannte „Scopes“, für Variablen. Wir haben z.B. bereits gelernt, dass eine Java-Variablen über mehrere JSP-Scriptlets einer Seite hinweg gültig sein kann, vgl. 5.1.2. Dabei handelt es sich um den Scope „page“. Auch eine Session ist ein definierter Geltungsbereich, vgl. 7. Hier die Übersicht:

Scope	Beschreibung
page	umfasst den aktuellen JSP-Aufruf
request	umfasst alle Seitenaufrufe des aktuellen Requests. Das kann mehr sein als page, wenn mit forward() weitergeleitet wird anstatt mit response.sendRedirect(), vgl. 5.1.4 (nicht so wichtig für uns)
session	umfasst alle Seitenaufrufe eines Benutzers, solange eine Session existiert und gültig ist (sehr wichtig für uns)
application	umfasst alle Seitenaufrufe aller User, solange bis die Applikation (oder der Server) neu gestartet wird. Variablen mit dem scope „application“ sind global und es können alle Benutzer darauf zu greifen.

8.1.3 JSP-Objekte

JSP besitzt einige Objekte, die vom Entwickler benutzt werden können, ohne dass diese zuerst deklariert werden müssen. Die, die für uns wichtig sind, haben wir bereits kennengelernt. Hier die Übersicht:

Objekt / Variable	Beispiele für Methoden	Beschreibung
out	print(...), println(...)	Schreibt html-Ausgabe in den Body der Response
request	getParameter(...)	Zugriff auf Parameter aus Formularen, Datentyp für name und value ist immer String
response	sendRedirect(...)	Weiterleitung unter Mitwirkung des Browsers
session	setAttribute(..., ...), getAttribute(...)	Zur Speicherung und Weitergabe beliebiger Objekte eines Benutzers
pageContext	forward(...)	Weiterleitung innerhalb des Servers
application	setAttribute(..., ...), getAttribute(...)	Zur Speicherung und Weitergabe beliebiger Objekte aller Benutzer
config	getInitParameter(...)	Verwaltung von Konfigurationsparametern

8.1.4 JSP-Tags

Für verschiedene Stereotypen, die in JSP regelmäßig wiederkehren, wurden Abkürzungen in XML-Form bereitgestellt.

<jsp:useBean .../>

Zur Erzeugung einer Java-Variablen mit einem bestimmten Scope steht das jsp:useBean-Tag zur Verfügung. Für uns ist dies besonders interessant mit scope="session", denn

<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
ist gleichbedeutend mit dem bereits bekannten untenstehenden Fragment und wird zur Laufzeit auch in dieses Fragment übersetzt:

```
MessageBean message = (MessageBean) session.getAttribute("message");
if (message == null){
    message = new MessageBean();
    session.setAttribute("message", message);
}
```

Beobachtungen:

- Insbesondere löst jsp:useBean einen new-Befehl und damit einen Konstruktor-Aufruf aus, was der Grund ist, weshalb Java-Beans immer einen Konstruktor ohne Parameter haben müssen, vgl. 6.1.4.
- Da in jsp:useBean nur eine id angegeben werden kann, muss diese sowohl als Name der Java-Variablen als auch als name innerhalb der Session verwendet werden, wie wir dies bereits in 7.2.2 angekündigt hatten.

Um eine Session-Variable zu erzeugen, ist die Angabe von scope="session" wichtig, denn

<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" />

ist gleichbedeutend mit

<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="page" />

was wiederum gleichbedeutend ist mit einem einfachen lokalen Java-Objekt:

```
MessageBean message = new MessageBean();
```

<jsp:getProperty .../>

Nachdem ein Objekt mit jsp:useBean angelegt und ggfs. in der Session verankert wurde, kann auf Inhalte des Objekts mit dem XML-Tag jsp:getProperty zugegriffen werden. Dabei wird allerdings

nicht direkt auf das Objekt bzw. seine Attribute zugegriffen, sondern es ist eine get-Methode erforderlich.

Dadurch gibt es zwei Möglichkeiten, die für unsere oben definierte MessageBean message folgendermaßen so aussehen kann:

- Zugriff auf die Methode getInfoMessage() mit
`<jsp:getProperty name="message" property="infoMessage" />`
- Zugriff auf die Methode getMessageHtml() mit
`<jsp:getProperty name="message" property="messageHtml" />`

Beobachtungen:

- Die id der in jsp:useBean definierten Bean muss bei jsp:getProperty im name-Attribut angegeben werden
- Im ersten Beispiel wird als property der Name der Variable infoMessage angegeben. Zur Ausführung wird dies interpretiert als Aufruf der Methode getInfoMessage(). Da eine solche Methode für die Attribute einer Bean existieren muss, vgl. 6.1.4, kann damit der Inhalt der Variable gelesen werden.
- Im zweiten Beispiel wird als property messageHtml angegeben, was kein gültiger Attributname ist. Allerdings wird dies interpretiert als Aufruf der Methode getMessageHtml(), die es tatsächlich gibt und die uns ein html-Fragment für unsere Ergebnisseite liefert.

Zusammenhang mit MVC

In den JSP-Seiten, die unsere View realisieren, möchten wir keinen bzw. möglichst wenig Java-Code haben. Wir haben inzwischen erkannt, dass dies für dynamische Seiten nicht möglich ist, allerdings bieten uns die genannten JSP-Tags die Möglichkeit, die beiden bisher genutzten Mechanismen in JSP-Tags zu verwandeln.

Dadurch wird das Java zwar nicht wirklich aus der Seite entfernt, aber es gibt uns die Sicherheit tatsächlich nur diese beiden Mechanismen verwendet zu haben und nicht etwa unkontrolliert gewachsenen Java-Code. Damit ist es auch ein Hinweis auf gute Architektur, wenn wir außer diesen JSP-Tags keinen weiteren Java-Code benötigen.

Weitere JSP-Tags

Zu jsp:useBean und jsp:getProperty gibt es z.B. noch jsp:setProperty zum Setzen von Attributen.

Außerdem können die üblichen Java-Klammern auch durch JSP-Tags ersetzt werden, z.B. sind die folgenden Konstrukte jeweils gleichbedeutend:

- `<% ... %>` entspricht `<jsp:scriptlet> ... </jsp:scriptlet>`
- `<%= ... %>` entspricht `<jsp:expression> ... </jsp:expression>` oder kurz `${...}`
- `<%! ... %>` entspricht `<jsp:declaration> ... </jsp:declaration>`
- `<%@ <typ> ... %>` entspricht `<jsp:directive.typ> ... </ ... >`

All diese Konstrukte verwenden wir in dieser Veranstaltung nicht, weil sie nach Meinung des Dozenten nicht zu einer besseren Architektur beitragen.

Achtung...!

JSP-Tags sind nützlich und schön. Es gibt aber auch einige Fehlerquellen, die sorgfältig ausgeschlossen werden müssen. Wir führen diese hier auf, einerseits um zur nötigen Sorgfalt aufzurufen, andererseits zum nachschlagen, falls unerklärliche Fehler auftreten:

- JSP-Tags sind XML-Tags, insbesondere müssen diese sorgfältig geschlossen werden, d.h.
`<jsp:useBean ... />` anstatt `<jsp:useBean ... >` und
`<jsp:getProperty ... />` anstatt `<jsp:getProperty ... >`
- Durch jsp:useBean wird implizit ein new-Befehl ausgelöst, der wiederum einen Aufruf des Konstruktors ohne Parameter auslöst. Wenn also bei der Ausführung der JSP, genauer bei Ausführung der jsp:useBean ein Fehler auftritt, kann dieser darin bestehen, dass

- ein solcher Konstruktor nicht vorhanden ist,
 - ein Fehler innerhalb des Konstruktors aufgetreten ist.
- ein typischer Fehler beim Anlegen einer UserBean ist z.B. dass die Datenbankverbindung nicht funktioniert, weil die PostgreSQL-Library nicht im Java Build Path oder der Deployment Assembly eingetragen wurde.
- Das Problem daran ist, dass die Fehlermeldung meistens sehr unklar ist, es muss also bei der Fehlersuche ausdrücklich daran gedacht werden.
- In den obigen Beispielen wurde jsp:getProperty mit der Attribut-Reihenfolge <jsp:getProperty name="" property="" /> verwendet, was aus Sicht des Dozenten sinnvoll ist, da hier zuerst die Bean angesprochen wird und dann die get-Methode innerhalb der Bean. Die Eclipse-Codevervollständigung liefert aber die Reihenfolge <jsp:getProperty property="" name="" />, was in der Vergangenheit oft zu Verwirrung geführt hat.
 - Das name-Attribut der jsp:getProperty entspricht dem id-Attribut der jsp:useBean, was eine gewisse Inkonsistenz darstellt.
 - jsp:useBean hat ein class-Attribut, in dem die Klasse der anzulegenden Bean angegeben werden muss. Es ist empfehlenswert, die Klasse in diesem Attribut immer voll zu qualifizieren und sie trotzdem auch mit einem import-Statement zu importieren. Insbesondere ältere Eclipse-Versionen hatten hier Schwierigkeiten.
 - Bei dem Namen der get-Methode in jsp:getProperty gibt es gelegentlich Schwierigkeiten:
 - property="htmlFragment" → getHtmlFragment() // korrekt
 - property="HtmlFragment" → <Fehler>
 - property="HTMLFragment" → getHTMLFragment () // korrekt
 Vor diesem Hintergrund ist es empfehlenswert, immer die erste Form zu verwenden.
 - Speziell in unserer Veranstaltung kam es außerdem gelegentlich zu Verwechslungen von useBean und UserBean.

8.2 Praxis: Nutzung von JSP-Tags

8.2.1 Nutzung von jsp:useBean

Sowohl in RegView.jsp als auch in RegAppl.jsp ersetzen wir den Java-Code

```
<body>
<%
UserBean user = (UserBean) session.getAttribute("user");
if (user == null){
    user = new UserBean();
    session.setAttribute("user", user);
}
MessageBean message = (MessageBean) session.getAttribute("message");
if (message == null){
    message = new MessageBean();
    session.setAttribute("message", message);
}>
```

durch gleichbedeutende jsp:useBean-Tags:

```
<body>
<jsp:useBean id="user" class="de.hs_lu.bw4s.beans.UserBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
```

In der RegView.jsp führt dies zu einer deutlichen Reduktion des expliziten Java-Codes.

In der RegAppl.jsp führt dies einerseits zu einer Darstellung, die mit der RegView.jsp übereinstimmt, also damit vereinheitlicht ist. Außerdem müssen nur noch zwei Zeilen überflogen werden, um zu sehen, welche Beans in der RegAppl.jsp eine Rolle spielen.

Beobachtung:

- Um den Bean Code für UserBean so anzupassen, dass er auf die MessageBean passt, mussten wir in 7.3.3 noch 9 Stellen ändern. Mit jsp:useBean müssen wir nur noch 2 Stellen ändern.

- Da es sich bei jsp:useBean immer noch um Java-Code handelt (auch wenn dieser versteckt ist), sollten wir die Klassen importieren. Dies erreichen wir wieder mit der Eclipse-Codevollständigung:

```
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<%
String userid = request.getParameter("userid");
String password = request.getParameter("password");
```

MessageBean - de.hs_lu.bw4s.beans
MessageBean - de.hs_lu.bw4s.beans

Hier sehen Sie zwei Einträge:

- Der erste vervollständigt nur den Klassennamen,
- der zweite fügt außerdem ein import-Statement ein.

Verwenden Sie deshalb immer den zweiten Eintrag und/oder prüfen Sie, dass das import-Statement tatsächlich auch erzeugt wurde.

8.2.2 Nutzung von jsp:getProperty

In der RegView.jsp befinden sich bisher noch einige JSP-Expressions, die wir durch gleichbedeutende jsp:getProperty-Tags ersetzen:

- <%= message.getMessageHtml() %>
→ <jsp:getProperty name="message" property="messageHtml" />
- <%= user.getUserid() %> → <jsp:getProperty name="user" property="userid" />
- <%= user.getUsername() %> → <jsp:getProperty name="user" property="username" />
- <%= user.getEmail() %> → <jsp:getProperty name="user" property="email" />

Im Ergebnis besitzt die RegView.jsp die folgende Gestalt:

```
<body>
<jsp:useBean id="user" class="de.hs_lu.bw4s.beans.UserBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<jsp:getProperty name="message" property="messageHtml" />
<form action=".\\jsp\\RegAppl.jsp" method="get">
  <table>
    <tr>
      <td>Nickname:</td>
      <td><input type="text" name="userid" value="

```

Beobachtung:

- Die RegView.jsp enthält nun tatsächlich keinen sichtbaren Java-Code mehr. Wir betrachten das als Zeichen für eine gute Architektur.
- Hinter dem ersten jsp:getProperty Tag (property="messageHtml") verbirgt sich eine get-Methode, die nicht zu einem Attribut gehört. Die anderen jsp:getProperty-Tags beziehen sich jeweils direkt auf ein Attribut.
- Durch die Ersetzung der JSP-Expressions durch jsp:getProperty-Tags kommt es zu einer Häufung von “-Zeichen. (Gänsebeinchen), die auf den ersten Blick verwirrend erscheint.
- Da die jsp:getProperty-Tags aber auf dem Server verarbeitet werden, erhält der Client nur noch die “-Zeichen außerhalb der getProperty-Tags. Damit kommen weder Server noch Client durcheinander.

8.2.3 Oberflächen-Kosmetik

Statische Version

Nachdem wir nun Code-Kosmetik zur Verschönerung des JSP-Codes betrieben haben, möchten wir nun auch noch Oberflächen-Kosmetik betreiben, indem wir die folgende statische Kopfzeile einfügen, die von nun an in allen sichtbaren JSP-Seiten angezeigt werden soll:



Der statische html-Code hierfür sieht, je nach Version des Logos, etwa so aus:

```
<body>
<jsp:useBean id="user" class="de.hs_lu.bw4s.beans.UserBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<table>
  <tr>
    <td>
      <h3>BW4S-Portal</h3>
    </td>
    <td>
      
    </td>
  </tr>
</table>
<jsp:getProperty name="message" property="messageHtml" />
```

Überlegungen zur Verbesserung

In einer umfangreichen Webanwendung kann es viele sichtbare Seiten geben. Wir möchten den statischen Code nicht in all diese Seiten kopieren, weil dies sonst bei späteren Änderungen einen hohen Änderungsaufwand mit sich bringt.

Es wäre nun denkbar, diesen Code-Bestandteil in eine eigene Datei auszulagern und mit include in alle Seiten einzubeziehen, vgl. 8.1.1. Besser scheint uns aber die folgende Lösung:

- Wir entwickeln eine Java-Bean GUIBean,
- die eine Methode getHeadlineHtml() enthält, die den oben dargestellten statischen Code zurückgibt.
- Wir binden die Bean und den Methodenaufruf in die RegView.jsp ein.

Implementierung

Entsprechend der genannten Überlegungen implementieren wir die GUIBean:

```
package de.hs_lu.bw4s.beans;

public class GUIBean {

    public GUIBean(){}

    public String getHeadlineHtml(){
        String html = "";
        html += "<table>\n";
        html += "  <tr>\n";
        html += "    <td>\n";
        html += "      <h3>BW4S-Portal</h3>\n";
        html += "    </td>\n";
        html += "    <td>\n";
        html += "      <img src='../img/FHLU_Logo_CMYK.jpg' width='256' height='75'\n";
        html += "            alt='Hier sollte das Logo der HS Lu stehen'\n";
        html += "            title='Logo der HS Lu'\n";
        html += "      />\n";
        html += "    </td>\n";
        html += "  </tr>\n";
        html += "</table>\n";
        return html;
    }
}
```

Anmerkung:

- Wir haben den Rahmen der Methode kopiert, den statischen Code hineinkopiert, html += " davor und \n"; dahinter geschrieben, außerdem die inneren " (Gänsebeinchen, Double Quotes) durch ' (Hochkomma, Single Quotes) ersetzt.

Außerdem implementieren wir die Änderung in der RegView.jsp:

```
<body>
<jsp:useBean id="user" class="de.hs_lu.bw4s.beans.UserBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<jsp:useBean id="gui" class="de.hs_lu.bw4s.beans.GUIBean" scope="session" />

<jsp:getProperty name="gui" property="headlineHtml" />
<jsp:getProperty name="message" property="messageHtml" />
```

Anmerkung:

- Wir haben eine jsp:useBean-Zeile hinzugefügt (andere jsp:useBean-Zeile kopiert und 2 Anpassungen vorgenommen) und eine jsp:getProperty-Zeile hinzugefügt, in der wir mit property="headlineHtml" den Aufruf der Methode getHeadlineHtml() veranlassen.

Test

Zum Test führen wir die Registrierung wieder aus und schauen uns den html-Quellcode an:

```
<body>

  BW4S-Portal  Hochschule
  Ludwigshafen am Rhein

  Willkommen zur BW4S-Registrierung

  Bitte registrieren Sie sich

  Nickname: 
  Passwort: 
  Username: 
  E-Mail: 
  
  
```

Willkommen zur BW4S-Registrierung

Bitte registrieren Sie sich

Nickname:

Passwort:

Username:

E-Mail:

Beobachtung: Der ursprünglich statische Code ist tatsächlich hier angekommen.

Die Fortführung des Tests durch erneute Registrierung eines Nickname überlassen wir dem Leser.

9 JSP-Architektur und -Deklarationen

9.1 Theorie

9.1.1 JSP als Java Klasse

Die JSP-Architektur baut vollständig auf Java auf. Insbesondere werden JSPs in Java-Klassen umgewandelt, um ausgeführt werden zu können.

Genauer anhand der RegView.jsp:

- Wenn die RegView.jsp geändert (und gespeichert) wird, erzeugt Eclipse daraus eine Java-Datei RegView_jsp.java mit einer Java-Klasse RegView_jsp und compiliert diese.
- RegView_jsp erbt von HttpJspBase und diese erbt von HttpServlet, die in den Packages javax.servlet.jsp bzw. javax.servlet.http definiert sind. Die JSP ist damit selbst ein Servlet im Sinne der Vererbung („is-a“-Beziehung)
- Innerhalb der RegView_jsp gibt es eine Methode

```
public abstract void _jspService(javax.servlet.http.HttpServletRequest arg0,
                                  javax.servlet.http.HttpServletResponse arg1)
                                  throws javax.servlet.ServletException, java.io.IOException.
```

Darin werden die Inhalte unserer JSP als reiner Java-Code eingefügt

Die folgende Abbildung zeigt dies schematisch:



Beobachtung:

- Insbesondere können wir in einer JSP bisher keine zusätzlichen Methoden entwickeln, weil wir uns ja schon im Kontext einer einzelnen Methode befinden.

Anmerkungen:

- Die Erzeugung und Compilierung der RegView_jsp.java dauert eine gewisse Zeit, weshalb Aktualisierungen von JSPs erst nach einigen Sekunden wirksam werden.
- RegView_jsp.java und RegView_jsp.class werden in Unterverzeichnissen von .metadata\plugins\org.eclipse.wst.server.core\...\BW4S angelegt.
- Was wir in RegView.jsp als html schreiben, wird in der _jspService(...) als out.print(...) eingetragen.
- In der Oberklasse HttpServlet gibt es zu jedem http-Befehl eine eigene Methode, z.B.

```
protected void doGet(javax.servlet.http.HttpServletRequest req,  
                     javax.servlet.http.HttpServletResponse resp)  
                     throws javax.servlet.ServletException, java.io.IOException,  
                     doGet(...), doPost(...), etc. In JSP werden diese alle in der Methode _jspService(...)
```

zusammengefasst.
- Die Variablen out, session, etc. stammen ebenfalls aus den Oberklassen HttpJspBase, etc.

9.1.2 Deployment und Ausführung der JSP im TomCat

Um die RegView.jsp starten zu können, wird auf dem TomCat ein einziges Objekt der neu generierten Klasse RegView_jsp erzeugt. Der Aufruf der JSP entspricht dann der Ausführung der Methode _jspService(...) dieses Objekts.

Damit gehen alle Aufrufe an dasselbe Objekt im Hauptspeicher. Soweit Attribute dieses Objekts angesprochen werden, sprechen alle Benutzer dasselbe Attribut an. Damit besitzen diese Attribute den Scope „application“, vgl. 8.1.2.

Die Erzeugung solcher Attribute (und Methoden) ist Thema des folgenden Abschnitts.

9.1.3 JSP-Deklarationen

Da objektorientierte Softwareentwicklung darauf angelegt ist, die Funktionalität in kleine Teile aufzuteilen, wünschen wir uns die Möglichkeit, auch in JSPs zusätzliche Methoden und auch zusätzliche Attribute anlegen zu können, auf die dann in der JSP zugegriffen werden kann. Speziell für Attribute hat dies den Nebeneffekt, dass diese global sind (scope „application“), vgl. 8.1.2 und 9.1.2.

Dafür müssen wir den Java-Kontext der _jspService(...)-Methode verlassen. Hierfür gibt es ein weiteres JSP-Sprachkonstrukt, nämlich die JSP-Deklaration `<%! ... %>`. Wir verdeutlichen dies anhand eines Beispiels, bei dem wir aufgrund der Einfachheit auf die MVC-Architektur verzichten.

9.1.4 Beispiel Zähler

Oberfläche

Wir legen eine neue JSP Zaehler.jsp an und darin ein Formular, das einen Zähler anzeigt und einen Button besitzt, mit dem dieselbe JSP erneut aufgerufen werden kann:

```
<title>BW4S-Zähler</title>
</head>
<body>
<h4>Willkommen beim globalen BW4S-Zähler</h4>
<form action=".//Zaehler.jsp" method="get">
    Der aktuelle Zählerstand ist: ...<br>
    <input type="submit" name="increment" value="+1" />
</form>
</body>
```

JSP-Deklaration

Wir stellen mittels JSP-Deklaration bereit:

- eine globale Variable int zaehler und
- eine Methode void increment(), die den Zähler um eins erhöht

```
<body>
<%! public int zaehler = 0; %>
<%! public void increment(){
    zaehler++;
} %>
<h4>Willkommen beim globalen BW4S-Zähler</h4>
```

JSP-Scriptlet

Wir implementieren Java-Code, der den Button „+1“ einliest und, wenn dieser geklickt wurde, den Zähler um eins erhöht, bevor dieser erneut ausgegeben wird:

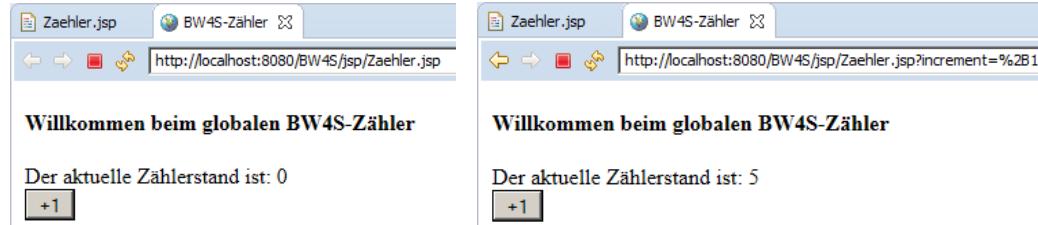
```
<%! public int zaehler = 0; %>
<%! public void increment(){
    zaehler++;
} %>
<%
    String increment = request.getParameter("increment");
    if (increment == null) increment = "";
    if (increment.equals("+1")){
        this.increment();
    }
%>
```

Außerdem implementieren wir eine JSP-Expression, die den Zähler an der richtigen Stelle ausgibt:

```
<h4>Willkommen beim globalen BW4S-Zähler</h4>
<form action=".//Zaehler.jsp" method="get">
    Der aktuelle Zählerstand ist: <%= this.zahler %><br>
    <input type="submit" name="increment" value="+1" />
</form>
```

Test / Ausführung

Zunächst starten wir die JSP im Tomcat auf einem Rechner und klicken einige Male auf „+1“:



Als nächstes ermitteln wir die IP-Adresse des aktuellen Rechners, z.B. per Eingabeaufforderung und ipconfig. Das Ergebnis könnte z.B. 143.93.208.27 sein.

Wir verwenden nun einen anderen Rechner, greifen mit der IP-Adresse auf den ersten Rechner zu und rufen dort die JSP auf mit z.B. <http://143.93.208.27:8080/BW4S/jsp/Zaehler.jsp> und klicken dann wieder auf den +1-Button:



Anmerkung: Evtl. muss zuerst auf Netzwerkebene der Zugriff erlaubt werden (Serverdienst, Firewall).

Beobachtung: Unterschiedliche Benutzer greifen auf dieselbe Seite und dieselben Attribute zu.

9.1.5 Zusammenfassung

- Zur Ausführung einer JSP wird ein einziges Objekt der JSP-Klasse auf dem TomCat Server gestartet. Seitenaufrufe von Benutzern wenden sich an eine JSP-Servicemethode dieses Objekts. Alle Benutzer rufen dieselbe Methode desselben Objekts auf.
- Variablen (Attribute) und Methoden des Objekts sind global (scope „application“), stehen also allen Benutzern gemeinsam zur Verfügung.
- Zur Erzeugung neuer Attribute und Methoden wird die JSP-Deklaration <%! ... %> verwendet.

9.2 Praxis: denullify()

Da wir unsere Registrierungsfunktion sauber gegliedert und die Aktionen in die UserBean verlagert haben, besteht wenig Bedarf an zusätzlichen Methoden innerhalb unserer JSPs.

Deshalb demonstrieren wir den Gebrauch von JSP-Deklarationen nur an einer sehr kleinen Methode, die uns den null-check nach dem Einlesen von Button-bezogenen Request-Parametern in der RegAppl.jsp vereinfachen soll.

9.2.1 Die Methode

Die Methode denullify(String s) wird in der RegAppl.jsp implementiert. Sie soll einen String annehmen und diesen unverändert wieder ausgeben, wenn er nicht null ist, andernfalls soll sie einen leeren String („“) ausgeben:

```
<%! public String denullify(String s) {
    if (s==null) return "";
    else return s;
} %>
```

Noch kürzer wäre die Methode mit dem ternären Operator ?: zu schreiben:

```
<%! public String denullify(String s) {return (s==null)?"":s;} %>
```

9.2.2 Einsatz der Methode

Die Methode kann in der RegAppl.jsp eingesetzt werden, um die folgenden Zeilen zu verschönern:

```
String zumlogin = request.getParameter("zumlogin");
String register = request.getParameter("register");
if (zumlogin == null) zumlogin = "";
if (register == null) register = "";
```

Unter Nutzung der denullify()-Methode lauten diese:

```
String zumlogin = this.denullify(request.getParameter("zumlogin"));
String register = this.denullify(request.getParameter("register"));
```

9.2.3 Reflektion und Erweiterung

Reflektion

Grundsätzlich macht die denullify()-Methode den Code lesbarer und ist deshalb von Vorteil.

Die Methode denullify() müsste in der vorliegenden Form auch in allen zukünftigen Controller-JSPs (Appl-s) angelegt werden. Da dies wieder Code-Kopien bedeutet, ist dies prinzipiell unerwünscht. Grundsätzlich wäre es deshalb besser, auch hier auf die JSP-Deklaration zu verzichten und die Denullifizierung durch eine Hilfsmethode in einer separaten Bean zu implementieren.

Erweiterung

Wir erinnern uns daran, dass die denullification dazu dient, eine NullPointerException beim Aufbau der Aktionsweiche zu vermeiden. Eine weitere Gefahrenquelle für NullPointerExceptions gibt es, wenn String-Arrays verwendet werden, z.B. beim Einlesen von Parametern mit mehrfachen values, z.B. checkboxen oder select-Boxen (Auswahlfelder) mit multiple-Eigenschaft. Beispiel:

```
String[] kreditkarten = request.getParameterValues("kreditkarten");
```

Beim durchlaufen des Arrays mit

```
for(int index = 0; index < kreditkarten.length(); index++) {...}
```

kann nämlich beim Aufruf von length() die NullPointerException auftreten, wenn das Array kreditkarten null ist, nicht aber, wenn es ein leeres Array ist.

Dieselbe NullPointerException tritt auf, wenn dasselbe Array mit einer foreach-Schleife durchlaufen wird: `for(String kreditkarte : kreditkarten) {...}`

Dabei ist der Fehler schwerer zu finden, weil der Aufruf von length() hier nicht mehr direkt sichtbar ist.

Aus diesem Grund ist es sinnvoll, auch bei String-Arrays von vornherein auszuschließen, dass diese null werden können, indem wir etwa eine entsprechende denullify-Methode bereitstellen:

```
<%! public String[] denullify(String[] sa) {return (sa==null)?new String[0]:sa;} %>
```

und beim Einlesen des String-Arrays verwenden:

```
String[] kreditkarten = this.denullify(request.getParameterValues("kreditkarten"));
```

10 Anmeldeseite und Portal

Bislang haben wir uns auf die Entwicklung einer einfachen Registrierungsanwendung konzentriert und dabei die verschiedenen Elemente von JSP und deren Rollen in der MVC-Architektur kennengelernt.

Wir wollen dies nun erweitern zu einem Web-Portal, dessen Benutzer nach erfolgreicher Anmeldung auf einer Portal-Seite eine Anwendung auswählen und ausführen können.

10.1 Theorie: Übersicht

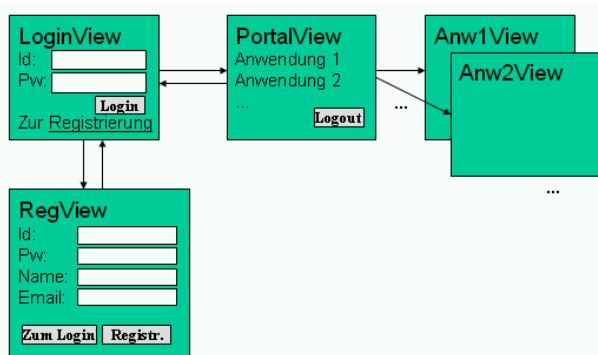
Die folgende Architektur wurde für diese Veranstaltung entwickelt und zeigt beispielhaft, wie eine gut strukturierte Server-basierte MVC-Webanwendung gegliedert sein kann. Trotzdem sind natürlich auch andere Architekturen denkbar, so dass die nachfolgenden Ausführungen nicht allgemeingültig sind.

Insbesondere können JavaScript-basierte Anwendungen, bestimmte Praxisunternehmen oder bestimmte Webentwicklungsframeworks von den hier aufgestellten Regeln abweichen.

10.1.1 Benutzersicht / Oberflächen

Aus Sicht der Benutzer soll unsere Portalanwendung aus den folgenden Seiten aufgebaut sein:

- Eine Anmeldeseite LoginView.jsp, die aus einem Login-Formular besteht und außerdem einen Link auf die Registrierungsseite besitzt.
- Unsere bereits bekannte Registrierungsseite RegView.jsp, die außerdem eine Sprungmöglichkeit auf die Login-Seite besitzt.
- Eine Portalseite, die durch erfolgreiche Anmeldung erreicht wird. Hier soll es möglich sein, aus verschiedenen Anwendungen eine auszuführen. Außerdem soll ein Logout möglich sein, der uns wieder zur Login-Seite zurückführt.
- Diverse Anwendungen, wobei wir hier auf die Lösungen der Übungsaufgaben zurückgreifen können.

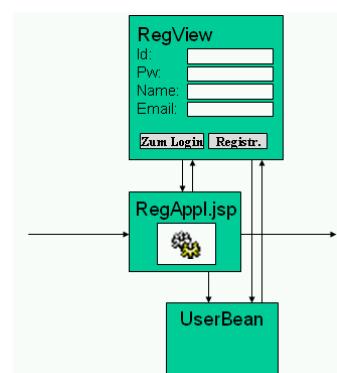


10.1.2 Aufbau in Komponenten

Aus Entwicklersicht ist der Aufbau etwas komplizierter, da wir bei der Registrierungsanwendung alleine bereits eine RegView.jsp, eine RegAppl.jsp und eine UserBean berücksichtigen müssen.

Begriff „Komponente“

Ein solches Paket aus View, Appl (Controller) und Bean (Model) bezeichnen wir als „Komponente“.



Aufbau einer Komponente

Da der Controller die Anwendungssteuerung übernehmen soll, stellt dieser (die Appl) auch den Einstiegspunkt in die Komponente dar und ebenso den Ausstiegspunkt aus der Komponente.

Im Fall der Registrierungsanwendung bedeutet dies Start mit der RegAppl.jsp ohne übertragenen Button. Es wird also der „else“-Zweig ausgeführt und entsprechend unserer bisherigen Programmierung die RegView.jsp angezeigt.

Eine Komponente besteht immer aus

- Genau einem Controller, der über den Programmablauf entscheidet und ggfs. Funktionalität in einer Bean oder in mehreren Beans aufruft.
- Einer oder mehreren Views, die
 - ggfs. vom Controller aufgerufen werden,
 - Daten aus den beteiligten Beans abrufen und anzeigen können,
 - und immer zu ihrem Controller zurückspringen.
- Bei Bedarf einer oder mehreren Beans, deren Funktionalität vom Controller aufgerufen wird und die außerdem bei Aufruf Daten an die Views zurückgeben.

Insbesondere gelten in unserer Beispielarchitektur die folgenden Regeln:

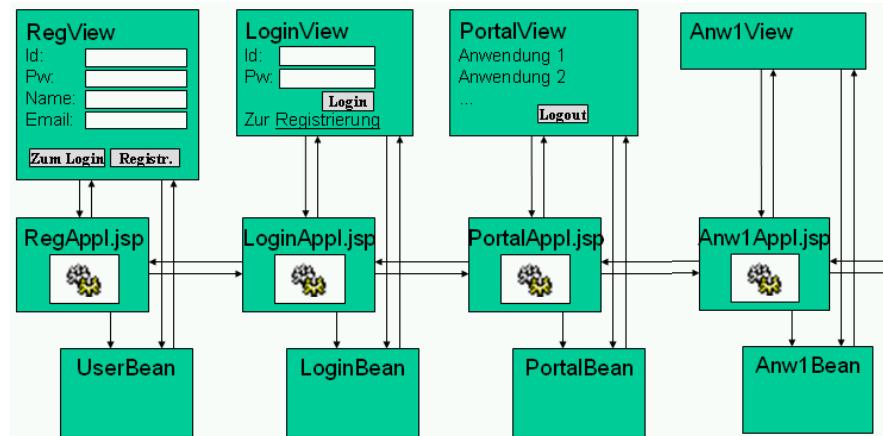
- Eine View kann in diesem Sinne nur zu einer Komponente bzw. nur zu einem Controller gehören. Umgekehrt kann ein Controller durchaus mehrere Views haben und steuern.
- Aus einer View wird niemals in eine andere Komponente gesprungen. Ebenso wird niemals aus einem Controller in die View einer anderen Komponente gesprungen.

Komponenten in der Portalanwendung

Für die Portalanwendung können wir also davon ausgehen, dass wir auch noch folgende Elemente benötigen:

- Eine LoginAppl.jsp und eine LoginBean
- Eine PortalAppl.jsp und eine PortalBean
- Für jede Anwendung ebenfalls ein Paket aus View(s), Appl und Bean(s)

Die folgende Abbildung fasst dies zusammen:



10.2 Praxis: Login- und Portal-Komponente

Im Folgenden zeigen wir, wie die in 10.1.2 genannten Komponenten prototypisch (und vereinfacht) aussehen könnten.

10.2.1 Login-Komponente

LoginBean

Die Login-Bean muss anhand der Angaben für userid (Nickname) und password entscheiden können, ob der User existiert und ob das Passwort korrekt ist. Hierfür benötigt die LoginBean analog zur UserBean Attribute

- String userid
- String password
- java.sql.Connection dbConn, um eine Datenbankverbindung zu speichern

und außerdem

- ein Attribut boolean isLoggedIn, um zu speichern, ob der User eingeloggt ist.

Die LoginBean braucht nun

- get- und set-Methoden für die genannten Attribute (erzeugt durch Eclipse)
- einen Konstruktor, der die genannten Attribute vorbereitet:

```
public class LoginBean {  
  
    String userid;  
    String password;  
    boolean isLoggedIn;  
    Connection dbConn;  
  
    public LoginBean() throws NoConnectionException{  
        this.userid = "";  
        this.password = "";  
        this.isLoggedIn = false;  
        this.dbConn = new PostgreSQLAccess().getConnection();  
    }  
}
```

- Eine Methode checkUserIdPassword(), die true oder false zurückgibt, je nachdem, ob der User mit diesem Passwort in der Datenbank existiert oder nicht.

Anstelle einer separaten LoginBean könnte auch die UserBean verwendet werden, da diese bereits die Attribute userid und password sowie die Datenbankverbindung dbConn enthält. Das Attribut isLoggedIn ist aber ein dynamisches Attribut und widerspricht dem eher statischen Charakter der UserBean. Wir erzeugen deshalb eine eigene LoginBean.

Die Methode checkUserIdPassword() in der LoginBean ist allerdings der Methode checkUserExists() in der UserBean sehr ähnlich. Genauer: Sie unterscheidet sich von dieser nur dadurch, dass

- das Passwort noch mit in die SQL-Abfrage aufgenommen wird
- und deshalb auch ein weiteres setString(...)-Statement nötig wird:

```
public boolean checkUserIdPassword() throws SQLException{  
    String sql = "SELECT userid FROM account "  
    + "WHERE userid = ? AND password = ?";  
    System.out.println(sql);  
    PreparedStatement myStat = dbConn.prepareStatement(sql);  
    myStat.setString(1, this.getUserid());  
    myStat.setString(2, this.getPassword());  
    ResultSet dbRes = myStat.executeQuery();  
    return dbRes.next();  
}  
  
public boolean checkUserExists() throws SQLException{  
    String sql = "SELECT userid FROM account "  
    + "WHERE userid = ?";  
    System.out.println(sql);  
    PreparedStatement myStat = this.dbConn.prepareStatement(sql);  
    myStat.setString(1, this.getUserid());  
    ResultSet dbRes = myStat.executeQuery();  
    return dbRes.next();  
}
```

LoginView.jsp

Entsprechend der Vorgaben in 10.1.1 benötigt die Login-View ein Login-Fomular mit einem Login-Button und einem Link zur Registrierungsanwendung:



Willkommen zur BW4S-Registrierung

Bitte registrieren Sie sich

Nickname:

Passwort:

Noch kein Nickname? Registrieren Sie sich [hier](#).

Zunächst überlegen wir uns, welche Daten in der Login-Komponente verarbeitet werden sollen und welche Beans wir hierfür benötigen:

- Wir verwenden die GUIBean und die MessageBean aus der Registrierungskomponente
- Wir benötigen eine LoginBean, wie sie oben entwickelt wurde.

Da unsere LoginView.jsp einige Ähnlichkeit hat mit der bereits existierenden RegView.jsp, legen wir die LoginView.jsp nicht mit New → JSP-File an, sondern als Kopie der RegView.jsp. Wir nehmen dann folgende Änderungen vor:

- Bei den import-Statements ersetzen wir den import für UserBean durch LoginBean

```
<%@page import="de.hs_lu.bw4s.beans.GUIBean"%>
<%@page import="de.hs_lu.bw4s.beans.MessageBean"%>
<%@page import="de.hs_lu.bw4s.beans.LoginBean"%>
```
- Bei den jsp:useBean-Tags ersetzen wir die UserBean mit id="user" durch eine LoginBean mit id="login":

```
<jsp:useBean id="Login" class="de.hs_lu.bw4s.beans.LoginBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<jsp:useBean id="gui" class="de.hs_lu.bw4s.beans.GUIBean" scope="session" />
```
- Die jsp:getProperty-Tags für Message und Headline übernehmen wir unverändert:

```
<jsp:getProperty name="gui" property="headlineHtml" />
<jsp:getProperty name="message" property="messageHtml" />
```
- Im Formular passen wir die action an zu LoginAppl.jsp (anstatt RegAppl.jsp) und übernehmen in der Tabelle die ersten beiden Zeilen. Allerdings passen wir im jsp:getProperty-Tag den name an zu "login" anstatt "user":

```
<form action=".\\jsp\\LoginAppl.jsp" method="get">
  <table>
    <tr>
      <td>Nickname:</td>
      <td><input type="text" name="userid" value=<jsp:getProperty name="login" property="userid" /></td>
    </tr>
    <tr>
      <td>Passwort:</td>
      <td><input type="password" name="password" value="" /></td>
    </tr>
  </table>
```
- Die Zeilen für username und email entfallen, anstatt zweier Buttons steht in der Tabelle nur noch ein Login-Button.

```
<tr>
  <td></td>
  <td><input type="submit" name="Login" value="Login" /></td>
</tr>
</table>
```
- Das Formular wird vervollständigt, indem hinter dem Formular noch ein Link implementiert wird, der den Sprung zur Registrierung ermöglicht (Erklärung s.u.):

```
</table>
</form>
Noch kein Nickname? Registrieren Sie sich <a href=".\\jsp\\LoginAppl.jsp?zurReg=zurReg">hier</a>.
```

Anmerkungen:

- Da wir die RegView.jsp kopiert haben, haben wir viel Arbeit gespart, allerdings kann es sehr leicht passieren, dass eine der genannten Anpassungen übersehen wird, was eine unangenehme Fehlerquelle darstellt.
- Der abschließend dargestellte Link [hier](.\\jsp\\LoginAppl.jsp?zurReg=zurReg) simuliert die Parameterübergabe eines geklickten Buttons mit name="zurReg" und value="zurReg". Deshalb kann dies in der LoginAppl.jsp einheitlich und analog zu dem Button login verarbeitet werden.

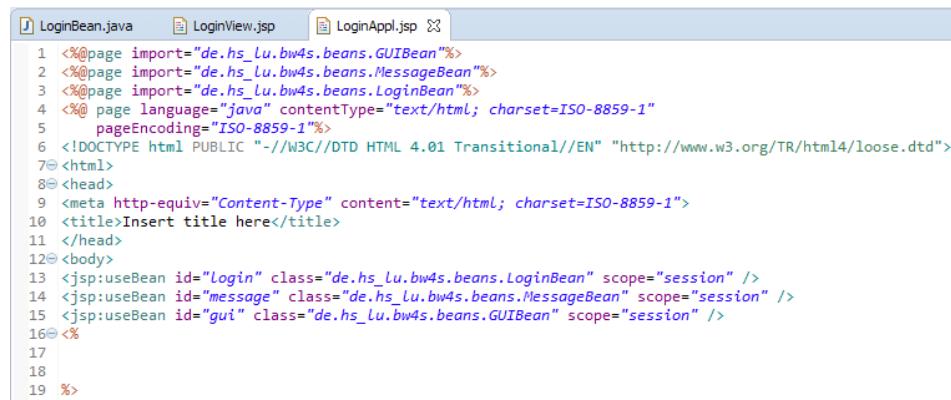
- Entsprechend 10.1.2 darf die LoginView.jsp nur in die LoginAppl.jsp springen und nicht etwa direkt in die RegAppl.jsp oder auf die RegView.jsp. Vor diesem Hintergrund bietet uns der simulierte Button zurReg die Möglichkeit, den Sprung über die LoginAppl.jsp durchzuführen.

LoginAppl.jsp: Einstieg und Übersicht

Zur Entwicklung der LoginAppl.jsp legen wir zunächst mit New → JSP-File die JSP-Datei an. Dann orientieren wir uns an dem Kochrezept aus Anhang 11.1, in dem wir unsere Erfahrungen aus der Entwicklung der RegAppl.jsp zusammengefasst haben:

- Beans in die Session einbinden
- Parameter aus dem Request übernehmen
- Aktionsweiche: Alle Möglichkeiten die der Nutzer hat, werden aufgeteilt
- Aktionen ausführen (Innerhalb der Weichen)

Die Beans der LoginAppl.jsp sind dieselben wie die der LoginView.jsp, deshalb können wir die entsprechenden jsp:useBean-Tags und auch die zugehörigen import-Direktiven übernehmen. Der Rest der LoginAppl.jsp besteht aus einem einzigen JSP-Scriptlet:



```

1 <%@page import="de.hs_lu.bw4s.beans.GUIBean"%>
2 <%@page import="de.hs_lu.bw4s.beans.MessageBean"%>
3 <%@page import="de.hs_lu.bw4s.beans.LoginBean"%>
4 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
5   pageEncoding="ISO-8859-1"%>
6 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
7<%>
8<html>
9 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
10 <title>Insert title here</title>
11 </head>
12<%>
13 <jsp:useBean id="login" class="de.hs_lu.bw4s.beans.LoginBean" scope="session" />
14 <jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
15 <jsp:useBean id="gui" class="de.hs_lu.bw4s.beans.GUIBean" scope="session" />
16<%>
17
18
19 %>

```

Die Request-Parameter kommen aus der LoginView.jsp und entsprechen dort den Eingabefeldern userid und password, dem submit-Button login sowie dem Link zur Registrierung, der einen Button-Klick zurReg simuliert. Für den echten und den simulierten Button führen wir außerdem den null-check durch. Da der Variablenname „login“ bereits für unsere LoginBean vergeben ist, muss für den login-Button ausnahmsweise auf einen anderen Variablennamen ausgewichen werden:

```

<%
String userid      = request.getParameter("userid");
String password    = request.getParameter("password");
String loginButton = request.getParameter("login");
String zurReg      = request.getParameter("zurReg");
if (loginButton == null) loginButton = "";
if (zurReg      == null) zurReg      = "";
%>

```

Die möglichen Aktionen, die auf der LoginView.jsp ausgelöst werden können, entsprechen den Buttons login (bzw. der Variablen loginButton) und zurReg. Wir entwickeln daraus unsere Aktionsweiche. Hier wird der Wert der Variablen loginButton mit dem value des input-Tags login verglichen. Insbesondere hat das input-Tag login (mit kleinem „l“) in der LoginView.jsp den value „Login“ (mit großem „L“):

```

if (loginButton.equals("Login")){
else if (zurReg.equals("zurReg")){
else{
}

```

Wir bearbeiten nun die nötigen Aktionen von unten nach oben.

LoginAppl.jsp: else-Zweig / keine Aktion

Der letzte else-Zweig behandelt alle Fälle, in denen keine der vorhergesehenen Aktionen auszuführen ist. In diesem Fall springen wir zurück auf die LoginView.jsp, damit der Benutzer sich anmelden kann. Wir brauchen dafür allerdings noch eine geeignete Message, die wir in der MessageBean in einer Methode setLoginWelcome() implementieren:

```
if (loginButton.equals("Login")){
}else if (zurReg.equals("zurReg")){
}else{
    message.setLoginWelcome();
    response.sendRedirect("./LoginView.jsp");
}
```

Da die Anmeldekomponente den natürlichen Einstieg in das System darstellt, setzen wir die Methode setLoginWelcome() außerdem auch als Default-Methode im Konstruktor der Message-Bean:

```
public class MessageBean {
    String infoMessage;
    String actionMessage;

    public MessageBean(){
//        this.setRegistrationWelcome();
        this.setLoginWelcome();
    }

    public void setLoginWelcome(){
        this.infoMessage = "Willkommen zur BW4S-Anmeldung";
        this.actionMessage = "Bitte melden Sie sich an";
    }
}
```

LoginAppl.jsp: Zur Registrierung

Im Aktionszweig zurReg wird ein Sprung in die RegAppl.jsp und damit in eine andere Komponente durchgeführt. Wir springen nicht direkt in die RegView.jsp, einerseits, um die Schnittstellen schlank zu halten, andererseits damit die RegAppl.jsp die Möglichkeit besitzt, vor der Anzeige der RegView.jsp evtl. noch weitere Login auszuführen.

Als Message verwenden wir die bereits bekannte setRegistrationWelcome():

```
if (loginButton.equals("Login")){
}else if (zurReg.equals("zurReg")){
    message.setRegistrationWelcome();
    response.sendRedirect("./RegAppl.jsp");
}else{
    message.setLoginWelcome();
    response.sendRedirect("./LoginView.jsp");
}
```

Anmerkung: In der RegAppl.jsp landen wir damit im dortigen else-Zweig der unvorhergesehenen Aktionen.

LoginAppl.jsp: Die Anmeldung

Für die Anmeldung müssen wir analog zur insert-Funktionalität der UserBean

- userid und password in die Bean eintragen
- in der Bean die Methode checkUseridPassword() ausführen
- entsprechend dem Ergebnis der Methode die Message und die Weiterleitung einrichten

Die ersten zwei Schritte sind unproblematisch, außer dass bei der Ausführung der Methode eine Exception auftreten kann:

```
if (loginButton.equals("Login")){
    login.setUserid(userid);
    login.setPassword(password);
    boolean loginSuccessful = login.checkUseridPassword();
```

Für die Ermittlung und Behandlung der Ergebnisse richten wir uns nach der folgenden Tabelle. Die einzige Aktion außer Message setzen und weiterleiten ist das setzen des isLoggedIn-Flags in der LoginBean:

Status	Anmeldung erfolgreich, userid / password sind gültig	Anmeldung fehlgeschlagen, userid / password sind ungültig	Fehler
Erkennbar an	loginSuccessful = true	loginSuccessful = false	Exception in der Methode checkUserIdPassword()
Aktion	Setze isLoggedIn auf true		
Methode in MessageBean	setLoginSuccessful()	setLoginFailed()	setAnyError() (Methode existiert schon)
Sprung nach	PortalAppl.jsp (existiert noch nicht)	LoginView.jsp	LoginView.jsp

Die unterschiedlichen Ergebnismöglichkeiten bilden wir folgendermaßen ab. Die SQLException ist außerdem zu importieren, z.B. mit Eclipse-Codevervollständigung:

```
if (loginButton.equals("Login")){
    login.setUserId(userid);
    login.setPassword(password);
    try{
        boolean loginSuccessful = login.checkUserIdPassword();
        if (loginSuccessful){
            ...
        }else{
            ...
        }
    }catch(SQLException se){
    }
}
```

Neue Methoden der MessageBean sind die Folgenden:

```
public void setLoginSuccessful(){
    this.infoMessage = "Willkommen am BW4S-Portal";
    this.actionMessage = "Bitte wählen Sie eine Anwendung aus";
}
public void setLoginFailed(){
    this.infoMessage = "Ihre Anmeldung ist fehlgeschlagen";
    this.actionMessage = "Bitte versuchen Sie es noch einmal";
}
```

Aktionsaufrufe, Message und Sprünge müssen in der LoginAppl.jsp nur noch hinzugefügt werden:

```
if (loginButton.equals("Login")){
    login.setUserId(userid);
    login.setPassword(password);
    try{
        boolean loginSuccessful = login.checkUserIdPassword();
        if (loginSuccessful){
            login.setLoggedIn(true);
            message.setLoginSuccessful();
            response.sendRedirect("./PortalAppl.jsp?comeFrom=LoginAppl");
        }else{
            message.setLoginFailed();
            response.sendRedirect("./LoginView.jsp");
        }
    }catch(SQLException se){
        message.setAnyError();
        response.sendRedirect("./LoginView.jsp");
    }
}else if (zurRueg.equals("zurRueg")){
    ...
}
```

10.2.2 Messages in den weiteren Zweigen der RegAppl.jsp

Neben der LoginAppl.jsp beobachten wir nun auch wieder die RegAppl.jsp. Seit 5.2.4 besitzt die RegAppl.jsp die folgenden Aktionszweige, für die noch kein Messaging implementiert wurde

```
if (zumLogin.equals("Zum Login")){
    // Aktionscode hier
    response.sendRedirect("./LoginAppl.jsp");
}else if (register.equals("Registrieren")){
    // Aktionscode hier
    response.sendRedirect("./RegView.jsp");
}
```

Inzwischen besitzt die MessageBean die hierfür benötigten Methoden, die wir in der RegAppl.jsp aufrufen:

```

if (zumlogin.equals("Zum Login")){
    message.setLoginWelcome();
    response.sendRedirect("./LoginAppl.jsp");
} else if (register.equals("Reегистрирован")){
}

```

```

} else{
    message.setRegistrationWelcome();
    response.sendRedirect("./RegView.jsp");
}

```

10.2.3 Messages und Sprünge zwischen zwei Apps

Eine Schwierigkeit der bisherigen Architektur ergibt sich bei Sprüngen von einer Appl in eine andere, z.B. beim Sprung von der LoginAppl.jsp zur RegAppl.jsp:

```

} else if (zurReg.equals("zurReg")){
    message.setRegistrationWelcome();
    response.sendRedirect("./RegAppl.jsp");
} else{
}

```

```

} else{
    message.setRegistrationWelcome();
    response.sendRedirect("./RegView.jsp");
}

```

Die Ausgangs-Appl setzt eine Message, springt dann auf die Ziel-Appl, landet dort im else-Zweig, wo die Message erneut gesetzt wird. In diesem Beispielfall ist dies egal, weil beide Male dieselbe Message gesetzt wird, aber z.B: beim Sprung von LoginAppl.jsp in die PortalAppl.jsp wird dies nicht mehr der Fall sein.

Um dieses Problem zu lösen, müssen wir zunächst entscheiden, welche der beiden Messages gelten soll. Da in der Ausgangs-Appl der aktuelle Systemstatus festgestellt wird und in der Ziel-Appl nur der unspezifische else-Zweig aktiv ist, möchten wir die Message der Ausgangs-Appl behalten.

Wir führen deshalb zur Lösung des Problems einen zusätzlichen Parameter comeFrom ein, mit dem wir der Ziel-Appl mitteilen, dass wir von einer anderen Appl herkommen. Wir schicken diesen Parameter immer mit, wenn wir von einer Appl in eine andere springen.

Anpassungsbedarf gibt es zunächst

- in der RegAppl.jsp, wenn zum Login gesprungen wird:

```

if (zumlogin.equals("Zum Login")){
    message.setLoginWelcome();
    response.sendRedirect("./LoginAppl.jsp?comeFrom=RegAppl");
} else if (register.equals("Reегистрирован")){
}

```

- in der LoginAppl.jsp, wenn in die PortalAppl.jsp oder in die RegAppl.jsp gesprungen wird:

```

if (loginSuccessful){
    message.setLoginSuccessful();
    response.sendRedirect("./PortalAppl.jsp?comeFrom=LoginAppl");
} else{
    ...
} else if (zurReg.equals("zurReg")){
    message.setRegistrationWelcome();
    response.sendRedirect("./RegAppl.jsp?comeFrom=LoginAppl");
} else{
}

```

Natürlich müssen die Ziel-Apps den Parameter auch einlesen und auswerten. Da eine equals()-Abfrage nötig sein wird, muss der Parameter auch null-gecheckt werden. Der folgende Code zum Einlesen muss in RegAppl.jsp und LoginAppl.jsp eingepflegt werden:

```

String comeFrom = request.getParameter("comeFrom");
if (comeFrom == null) comeFrom = "";

```

Der else-Zweig beider Apps kann nun dahingehend erweitert werden, dass die Message nur dann gesetzt wird, wenn der comeForm-Parameter nicht gesetzt ist.

Für die RegAppl.jsp:

```

} else if (!comeFrom.equals("")){
    response.sendRedirect("./RegView.jsp");
} else{
    message.setRegistrationWelcome();
    response.sendRedirect("./RegView.jsp");
}

```

Für die LoginAppl.jsp:

```

} else if (!comeFrom.equals("")){
    response.sendRedirect("./LoginView.jsp");
} else{
    message.setLoginWelcome();
    response.sendRedirect("./LoginView.jsp");
}

```

Anmerkung: Dieses Verfahren funktioniert auch, wenn weitere Komponenten einbezogen werden, in unserem Fall die Portal-Komponente.

10.2.4 Portal-Komponente

PortalBean

Aufgabe der PortalBean ist es, die vorhandenen Anwendungen bereitzustellen. Dies kann natürlich in unterschiedlichem Komplexitätsgrad erfolgen. In professionellen Portalen werden die Anwendungen in einer Datenbank gespeichert sein und eine eigene Administrationsoberfläche besitzen. In unserem vereinfachten Fall speichern wir die Anwendungen als String in einem Vector in der PortalBean und füllen diesen Vector<String> im Konstruktor der PortalBean:



```
1 package de.hs_lu.bw4s.beans;
2
3 import java.util.Vector;
4
5 public class PortalBean {
6
7     Vector<String> alleAnwendungen;
8
9     public PortalBean() {
10         this.alleAnwendungen = new Vector<String>();
11         this.alleAnwendungen.add("A4Appl");
12         this.alleAnwendungen.add("ShowUsersAppl");
13     }
14 }
```

Beobachtungen:

- Die Klasse Vector muss aus dem Package java.util importiert werden.
- Der Vector alleAnwendungen muss im Konstruktor zunächst mit new instanziert werden.
- Die Anwendungen sind hier „hart codiert“, das heißt für eine Änderung der Anwendungsauswahl muss in den Programmcode eingegriffen werden.

Anmerkung: Damit PortalBean eine Bean wird, müssten für das Attribut alleAnwendungen get- und set-Methoden bereitgestellt werden. Diese Methoden sind aber nur sinnvoll, wenn mit jsp:getProperty darauf zugegriffen werden soll. Deshalb erlauben wir uns hier eine kleine Verletzung der Regel und verzichten auf die Standard-get- und set-Methoden.

Stattdessen stellen wir eine Methode getAnwendungenAlsHtml() bereit, die aus den Anwendungen anklickbare html-Links erzeugt:

```
public String getAnwendungenAlsHtml(){
    String html = "";
    for (String anwendung : alleAnwendungen){
        html += "<a href='./PortalAppl.jsp?anwendung=" + anwendung + "'>" + anwendung + "</a><br>\n";
    }
    html += "<br>\n";
    return html;
}
```

Anmerkungen:

- Da auch die PortalView nur auf die PortalAppl springen soll und nicht direkt in die Anwendung, konstruieren wir die Links wieder so, dass diese als Request-Parameter übertragen werden. Erst in der PortalAppl werden dann die echten Anwendungs-URLs erzeugt.
- Durch „
“ wird ein Zeilenumbruch in der dargestellten Webseite erzeugt, durch „\n“ wird ein Zeilenumbruch im html-Quellcode erzeugt. Dieser ist zwar auf der Webseite nicht sichtbar, macht aber den html-Quellcode lesbarer, z.B. für die Fehlersuche.

PortalView.jsp

In der PortalView werden die Kopfzeile, die Message und die Anwendungen angezeigt. Außerdem soll ein Logout-Button das Verlassen des Portals erlauben:



Willkommen zur BW4S-Anmeldung

Bitte melden Sie sich an

[A4Appl](#)
[ShowUsersAppl](#)

[Logout](#)

Beobachtung: Durch den Direktaufruf der PortalView wird hier eine falsche Message angezeigt. Wäre die Seite nach einem erfolgreichen Login angezeigt worden, hätte die Message gestimmt. Vor diesem Hintergrund wird klar, dass die Anwendung immer über die Appl gestartet werden soll und niemals direkt über die View.

Die Seite wird von dem folgenden JSP-Code erzeugt. Es werden zunächst eine PortalBean, eine MessageBean und eine GUIBean verwendet.

```
<body>
<jsp:useBean id="portal" class="de.hs_lu.bw4s.beans.PortalBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<jsp:useBean id="gui" class="de.hs_lu.bw4s.beans.GUIBean" scope="session" />

<jsp:getProperty name="gui" property="headlineHtml" />
<jsp:getProperty name="message" property="messageHtml" />

<jsp:getProperty name="portal" property="anwendungenAlsHtml" />

<form action=".\\jsp\\PortalAppl.jsp" method="get">
    <input type="submit" name="logout" value="Logout" />
</form>
</body>
```

Wird die html-Seite fehlerhaft angezeigt, müssen wir die get-Methoden der Beans überprüfen. Vorher ist es sinnvoll, den html-Quellcode der Seite anzusehen, um den Fehler zielsicher suchen zu können. In unserem Fall kann der Quellcode genutzt werden, um die Syntax der Anwendungs-Links zu überprüfen:

```
<body>

<table>
    <tr>
        <td>
            <h3>BW4S-Portal</h3>
        </td>
        <td>
            <img src='..\\img\\FHLU_logo_CMYK.jpg' width='256' height='75'
                  alt='Hier sollte das Logo der HS Lu stehen'
                  title='Logo der HS Lu'
            />
        </td>
    </tr>
</table>

<h3>Willkommen zur BW4S-Anmeldung</h3>
<h4>Bitte melden Sie sich an</h4>

<a href=".\\PortalAppl.jsp?anwendung='A4Appl'">A4Appl</a><br>
<a href=".\\PortalAppl.jsp?anwendung='ShowUsersAppl'">ShowUsersAppl</a><br>
<br>

<form action=".\\jsp\\PortalAppl.jsp" method="get">
    <input type="submit" name="logout" value="Logout" />
</form>
</body>
```

PortalAppl.jsp: Einstieg und Übersicht

Zur Entwicklung der PortalAppl.jsp legen wir zunächst mit New → JSP-File die JSP-Datei an. Wie in 10.2.1 orientieren wir uns dann an dem Kochrezept aus Anhang 11.1.

Die Beans der PortalAppl.jsp sind zunächst dieselben wie die der PortalView.jsp, deshalb können wir die entsprechenden jsp:useBean-Tags und auch die zugehörigen import-Direktiven übernehmen. Für die Logout-Funktionalität müssen wir aber auch auf die LoginBean zugreifen können und benötigen dafür den import und den jsp:useBean. Der Rest der PortalAppl.jsp besteht aus einem einzigen JSP-Scriptlet:

```

<%@page import="de.hs_lu.bw4s.beans.GUIBean"%>
<%@page import="de.hs_lu.bw4s.beans.MessageBean"%>
<%@page import="de.hs_lu.bw4s.beans.PortalBean"%>
<%@page import="de.hs_lu.bw4s.beans.LoginBean"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="Login" class="de.hs_lu.bw4s.beans.LoginBean" scope="session" />
<jsp:useBean id="portal" class="de.hs_lu.bw4s.beans.PortalBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<jsp:useBean id="gui" class="de.hs_lu.bw4s.beans.GUIBean" scope="session" />
<%

```

<%>

Die Request-Parameter kommen aus der PortalView.jsp und sind logout und anwendung, entsprechend dem Button bzw. den Anwendungs-Links. Wie in 10.2.3 verwenden wir außerdem den Parameter comeFrom. Für alle drei Parameter führen wir den null-check durch:

```

<%
String logout = request.getParameter("logout");
String anwendung = request.getParameter("anwendung");
String comeFrom = request.getParameter("comeFrom");
if (logout == null) logout = "";
if (anwendung == null) anwendung = "";
if (comeFrom == null) comeFrom = "";

%>

```

Die möglichen Aktionen für die Aktionsweiche entsprechen diesen Parametern:

- Der Parameter „logout“ (mit kleinem ,l‘) wird mit dem value „Logout“ (mit großem ,L‘) verglichen.
- Bei den Parametern anwendung und comeFrom können unterschiedliche Werte auftreten.
- Natürlich wird auch hier ein else-Zweig benötigt

```

if (logout.equals("Logout")){
}else if (!anwendung.equals("")){
}else if (!comeFrom.equals("")){
}else{
}

```

PortalAppl.jsp: Logout

Zum Logout sind die folgenden Schritte umzusetzen:

- Setze das isLoggedIn-Flag in der LoginBean auf false.
- Entwickle eine Message setLoggedOut() und rufe diese auf.
- Springe zurück in die Login-Komponente. Genauer: Springe in die LoginAppl.jsp. Da wir hier von einer Appl zur anderen springen, geben wir den comeFrom-Parameter mit.

Wir entwickeln also setLoggedOut() in der MessageBean

```

public void setLoggedOut(){
    this.infoMessage = "Sie haben sich abgemeldet";
    this.actionMessage = "Bitte melden Sie sich wieder an";
}

```

und fügen den restlichen Code in der PortalAppl.jsp ein:

```

if (logout.equals("Logout")){
    login.setLoggedIn(false);
    message.setLoggedOut();
    response.sendRedirect("./LoginAppl.jsp?comeFrom='PortalAppl'");
}else if (!anwendung.equals("")){

```

PortalAppl.jsp: Anwendung auswählen

Zur Auswahl einer Anwendung sind die folgenden Schritte umzusetzen:

- Konstruiere die Anwendungs-URL aus dem Anwendungsnamen
- Springe zu der URL

Das Setzen einer Message ist nur erforderlich, wenn die Anwendung unser Message-Framework ebenfalls unterstützt. Dies ist aber bisher nicht der Fall. Dasselbe gilt für den Parameter comeFrom, der ja ein Teil des Message-Frameworks ist.

```
else if (!anwendung.equals("")){
    String anwendungsURL = "./" + anwendung + ".jsp";
    response.sendRedirect(anwendungsURL);
} else if (!comeFrom.equals("")){
```

PortalAppl.jsp: comeFrom- und else-Zweig

Wenn von einer anderen Komponente in die PortalAppl gesprungen wird (zum jetzigen Stand des Systems kann das nur die Login-Komponente sein), dann ist die Message schon gesetzt und wir springen ohne Weiteres in die PortalView.jsp. Diese Situation ist gegeben, wenn der comeFrom-Parameter gesetzt ist:

```
else if (!comeFrom.equals ""){
    response.sendRedirect("PortalView.jsp");
} else {
```

Wenn auf andere unvorhergesehene Art in die PortalAppl.jsp gesprungen wird, landen wir im else-Zweig. Dieser Fall darf eigentlich nicht auftreten und könnte auf einen Angriffsversuch hindeuten. Wir behandeln diesen wie folgt:

- Wir prüfen, ob in der LoginBean das Flag isLoggedIn gesetzt ist
- Falls ja, setzen wir eine Willkommens-Message und springen in die PortalView.jsp
- Falls nein, setzen wir eine Security-Message und springen mit einem comeFrom-Parameter in die LoginAppl.jsp

Zuerst müssen wir also in der MessageBean die Willkommens- und die Security-Message entwickeln:

```
public void setPortalWelcome(){
    this.infoMessage = "Willkommen am BW4S-Portal";
    this.actionMessage = "Bitte wählen Sie eine Anwendung aus";
}
public void setNotLoggedIn(){
    this.infoMessage = "Sie sind nicht angemeldet";
    this.actionMessage = "Bitte melden Sie sich an";
}
```

Anschließend können wir die oben angegebenen Schritte programmieren:

```
} else {
    if (login.isLoggedIn()){
        message.setPortalWelcome();
        response.sendRedirect("PortalView.jsp");
    } else {
        message.setNotLoggedIn();
        response.sendRedirect("LoginAppl.jsp?comeFrom=PortalAppl");
    }
}%>
```

10.2.5 Testen

An dieser Stelle wäre ein Test der bisherigen Funktionalität angebracht. Die folgenden Schritte sollten funktionieren:

- Aufruf der LoginAppl.jsp → LoginView.jsp wird angezeigt mit Willkommens-Message
- Klick auf Registrieren Sie sich hier → RegView.jsp wird angezeigt mit korrekter Message
- Dateneingabe und klick auf Registrieren → RegView.jsp wird angezeigt mit Erfolgsmeldung
- Klick auf „zum Login“ → LoginView.jsp wird angezeigt mit Willkommens-Meldung
- Eingabe userid/password und klick auf Login → PortalView wird angezeigt mit Willkommens-Meldung
- Klick auf Logout → LoginView.jsp wird angezeigt mit Meldung „Sie haben sich abgemeldet“

Falls die Anwendungen bereits zur Verfügung stehen, kann auch der Sprung vom Portal in die Anwendungen getestet werden.

Die nächsten Tests zeigen, was bei entsprechend unvorhergesehenen Aufrufen passieren kann:

- Aufruf PortalAppl.jsp ohne vorherige Anmeldung → über else-Zweig Rücksprung über LoginAppl.jsp nach LoginView.jsp mit SecurityMessage → kein Problem
Anm.: Der folgende Screenshot zeigt, dass auf der LoginView.jsp anstatt einer Anmeldung einfach in der URL-Zeile die PortalAppl.jsp aufgerufen werden kann:

The left screenshot shows the URL <http://localhost:8080/BW4S/jsp/PortalAppl.jsp>. It displays a message "Sie haben sich abgemeldet" and a login form with fields for Nickname and Password, and a "Login" button. Below the form is a link "Noch kein Nickname? Registrieren Sie sich [hier](#)".
The right screenshot shows the URL <http://localhost:8080/BW4S/jsp/LoginView.jsp>. It displays a message "Sie sind nicht angemeldet" and a login form with fields for Nickname and Password, and a "Login" button. Below the form is a link "Noch kein Nickname? Registrieren Sie sich [hier](#)".

- Aufruf PortalView.jsp ohne vorherige Anmeldung → Anzeige der PortalView.jsp ohne Änderung der Message, allerdings könnten die Anwendungen jetzt aufgerufen werden:

The left screenshot shows the URL <http://localhost:8080/BW4S/jsp/PortalView.jsp>. It displays a message "Sie sind nicht angemeldet" and a login form with fields for Nickname and Password, and a "Login" button. Below the form is a link "Noch kein Nickname? Registrieren Sie sich [hier](#)".
The right screenshot shows the URL <http://localhost:8080/BW4S/jsp/PortalView.jsp>. It displays a message "Sie sind nicht angemeldet" and a login form with fields for Nickname and Password, and a "Login" button. Below the form are links "A4Appl" and "ShowUsersAppl".

- Aufruf PortalAppl.jsp?comeFrom=LoginAppl ohne vorherige Anmeldung → über comeFrom-Zweig ebenfalls Weiterleitung in die PortalView.jsp:

The left screenshot shows the URL <http://localhost:8080/BW4S/jsp/PortalAppl.jsp?comeFrom=LoginAppl>. It displays a message "Sie haben sich abgemeldet" and a login form with fields for Nickname and Password, and a "Login" button. Below the form is a link "Noch kein Nickname? Registrieren Sie sich [hier](#)".
The right screenshot shows the URL <http://localhost:8080/BW4S/jsp/PortalView.jsp>. It displays a message "Sie sind nicht angemeldet" and a login form with fields for Nickname and Password, and a "Login" button. Below the form are links "A4Appl" and "ShowUsersAppl".

Das letzte Beispiel zeigt, wie Angreifer profitieren können, wenn sie die inneren Funktionsweisen einer Anwendung kennen.

10.2.6 Beheben der Authentisierungslücken

Die letzten beiden Beispiele zeigen, wie ein Angreifer durch unvorhergesehene Aufrufe den Authentisierungsprozess (die Anmeldung) umgehen kann. Um dies zu beheben haben wir bereits in der LoginBean das Flag isLoggedIn eingebaut, das bei einer erfolgreichen Anmeldung auf true und bei der Abmeldung wieder auf false gesetzt wird.

Jede JSP, die nur nach erfolgreicher Anmeldung besucht werden darf, sollte nun prüfen, dass dieses Flag tatsächlich true ist und andernfalls mit Security Message in die LoginAppl.jsp springen. In unserem Fall handelt es sich dabei um die PortalAppl.jsp und die PortalView.jsp.

Anpassung PortalAppl.jsp

In der PortalAppl.jsp ist die Anpassung sehr einfach:

- An den Anfang der Aktionsweiche setzen wir eine Prüfung des isLoggedIn-Flags.
- Falls dieses false ist, führen wir dieselbe Aktion durch wie bereits im comeFrom-Zweig für den Fall der fehlenden Anmeldung implementiert haben.
- Nur falls isLoggedIn true ist, führen wir den Rest der Aktionsweiche durch.

```
if (!login.isLoggedIn()){
    message.setNotLoggedIn();
    response.sendRedirect("LoginAppl.jsp?comeFrom=PortalAppl");
} else if (logout.equals("Logout")){
    login.isLoggedIn(false);
```

Damit ist das dritte der obigen Beispiele nicht mehr problematisch:

The image contains two side-by-side screenshots of a web application interface. Both screenshots show the same basic layout: a header with the 'BW4S-Portal' logo and the text 'Hochschule Ludwigshafen am Rhein'. Below the header, there is a message area and a form area. The left screenshot displays the message 'Sie haben sich abgemeldet' (You have logged out) and the form fields 'Nickname:' and 'Passwort:' with their respective input boxes. A 'Login' button is located below the form. The right screenshot displays the message 'Sie sind nicht angemeldet' (You are not logged in) and the same form fields. Both screenshots also include a link 'Noch kein Nickname? Registrieren Sie sich [hier](#)' (Still no nickname? Register here).

Anpassung PortalView.jsp, Variante 1

In der PortalView.jsp ist ebenfalls eine einfache Anpassung möglich, allerdings verletzten wir damit unser Paradigma, dass in der View kein Code enthalten sein darf. Wir müssten dieses Paradigma also für den Fall der Security-Prüfung aufweichen. Die folgenden Schritte wären dann durchzuführen:

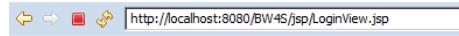
- Einbindung der LoginBean mit jsp:useBean und import-Statement

```
<%@page import="de.hs_lu.bw4s.beans.GUIBean"%>
<%@page import="de.hs_lu.bw4s.beans.MessageBean"%>
<%@page import="de.hs_lu.bw4s.beans.PortalBean"%>
<%@page import="de.hs_lu.bw4s.beans.LoginBean"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>BW4S-Registrierung</title>
</head>
<body>
<jsp:useBean id="login" class="de.hs_lu.bw4s.beans.LoginBean" scope="session" />
<jsp:useBean id="portal" class="de.hs_lu.bw4s.beans.PortalBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<jsp:useBean id="gui" class="de.hs_lu.bw4s.beans.GUIBean" scope="session" />
```

- Implementierung eines JSP-Scriptlets, das mit Security-Message und comeFrom-Parameter auf die PortalAppl.jsp weiterleitet:

```
<body>
<jsp:useBean id="login" class="de.hs_lu.bw4s.beans.LoginBean" scope="session" />
<jsp:useBean id="portal" class="de.hs_lu.bw4s.beans.PortalBean" scope="session" />
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />
<jsp:useBean id="gui" class="de.hs_lu.bw4s.beans.GUIBean" scope="session" />
<%
if (!login.isLoggedIn()){
    message.setNotLoggedIn();
    response.sendRedirect("./PortalAppl.jsp?comeFrom=PortalView");
}
%>
<jsp:getProperty name="gui" property="headlineHTML" />
<jsp:getProperty name="message" property="messageHTML" />
```

Ein Test könnte so aussehen, dass wir nach einer Abmeldung oder nach einem Server-Neustart die PortalView.jsp direkt aufrufen und landen auf der LoginView.jsp:



Willkommen zur BW4S-Anmeldung

Bitte melden Sie sich an

Nickname:
Passwort:

Noch kein Nickname? Registrieren Sie sich [hier](#).

Sie sind nicht angemeldet

Bitte melden Sie sich an

Nickname:
Passwort:

Noch kein Nickname? Registrieren Sie sich [hier](#).

Anpassung PortalView.jsp, Variante 2

Wir haben bereits früher Java-Code durch jsp:getProperty-Tags ersetzt und damit erreicht, dass der Code aus der View in die Bean verlagert wird.

Um dies auch für den gezeigten Security-Code zu erreichen, brauchen wir

- das html-Tag <meta http-equiv="refresh" content="0; URL=../PortalAppl.jsp" >, das eine Weiterleitung auf die angegebene URL auslöst, wenn es im Header der html-Seite angegeben wird,
- eine Methode getLoginCheckRedirectHtml(), die wir in der LoginBean implementieren und die das angegebene html-Tag erzeugt:

```
public String getLoginCheckRedirectHtml(){  
    String html = "";  
    if (!this.isLoggedIn()){  
        html = "<meta http-equiv=\"refresh\" content=\"0; URL=../PortalAppl.jsp\" >";  
    }  
    System.out.println("isLoggedIn ist " + this.isLoggedIn());  
    return html;  
}
```

- ein jsp:getProperty-Tag, das wir im Header der PortalView.jsp einbinden und das, falls isLoggedIn false ist, die getLoginCheckRedirectHtml() aufruft und damit das html-Tag <meta ...> in die PortalView.jsp schreibt. Da das jsp:useBean-Tag vor dem jsp:getProperty-Tag kommen sollte, ziehen wir auch die jsp:useBean-Tags in den html-Header der JSP:

```
<head>  
<jsp:useBean id="Login" class="de.hs_lu.bw4s.beans.LoginBean" scope="session" />  
<jsp:useBean id="portal" class="de.hs_lu.bw4s.beans.PortalBean" scope="session" />  
<jsp:useBean id="message" class="de.hs_lu.bw4s.beans.MessageBean" scope="session" />  
<jsp:useBean id="gui" class="de.hs_lu.bw4s.beans.GUIBean" scope="session" />  
<jsp:getProperty name="login" property="loginCheckRedirectHtml" />
```

Ein Test ergibt, dass dieses Verfahren ebenfalls funktioniert, allerdings ist kurz die Anzeige der PortalView.jsp zu sehen, bevor die Weiterleitung wirksam wird.

Reflektion

Beide Verfahren verwenden Java-Code, wobei das zweite Verfahren diesen in einem jsp:getProperty-Tag versteckt. Allerdings weist das zweite Verfahren schlechtere Funktionalität auf, wie die PortalView kurz zu sehen ist, was eigentlich unerwünscht ist.

Vor diesem Hintergrund greifen wir doch lieber auf Verfahren 1 zurück und nehmen in Kauf, dass das „kein Code in der View“-Paradigma durch den Security-Code verletzt wird.

11 Anhang

11.1 Anhang 1: Kochrezept zur Entwicklung einer Webanwendung

Kochrezept für die gesamte Webanwendung

1. Oberfläche bauen (HTML, JSP)
2. Daten analysieren und Bean planen/anlegen/einrichten inklusive Aktionen
3. Appl anlegen und mit Views verbinden
4. Appl ausarbeiten: s.u.
5. Ausgabe in Views einbinden

Controller (Appl):

1. Beans in die Session einbinden
2. Parameter aus dem Request übernehmen
 - Parameter einlesen mit getParameter(...) oder getParameterValues(...)
 - Buttons und Collections/Arrays auf null checken und ggfs. ändern
3. Aktionsweiche: Alle Möglichkeiten die der Nutzer hat, werden aufgeteilt
4. Aktionen ausführen (Innerhalb der Weichen)
 - Bean ggfs. mit Parameter-Daten füllen
 - Bean beauftragen, die Aktion auszuführen
 - Ergebnis auswerten, Fehlerbehandlung, z.B. mit try/catch
 - für jeden möglichen Ausgang Message setzen und
 - auf nächste View springen / weiterleiten (Screen Flow)

11.2 Anhang 2: Systemeinrichtung

Im Folgenden haben wir die einzelnen Schritte zur Einrichtung unserer Systemumgebung noch einmal gesammelt zusammengestellt. Einige dieser Schritte wurden in vorigen Kapiteln ausführlich erklärt, andere vielleicht weniger.

Die Aufstellung hier erhebt keinen Anspruch auf Vollständigkeit, sollte aber geeignet sein, die technischen Schritte im Wesentlichen noch einmal nachzuschlagen.

Einrichtung Java

Java JDK 1.8 (v102), es muss JDK sein, JRE reicht nicht. Herunterladen und installieren

Einrichtung Apache Webserver und Tomcat

Apache httpd herunterladen von httpd.apache.org, Installation wie im entsprechenden Kapitel beschrieben.

Tomcat 8.0 herunterladen und Installation wie im entsprechenden Kapitel beschrieben von tomcat.apache.org und dort referenzierten Seiten. Installation durch Entpacken! Kein Service Installer!

Einrichtung PostgreSQL

Postgres DB-Server 9.5 (9.5.4.1) unter ... \Programme\PostgreSQL\9.5

PostgresAdmin 3 (1.22.1) unter ... \Programme\ PostgreSQL\9.5\bin\pgAdmin3.exe

PostGreSQL JDBC-Treiber z.B. unter ... \Programme\pgJDBC\postgresql-9.4.1208.jar

- dbDrivername = "org.postgresql.Driver";

Lokale Anmeldedaten

- dbURL = "jdbc:postgresql://localhost:5432/BWUEBDB"; statt "localhost" auch "127.0.0.1"
- pg-Server user postgres, Passwort pgadmin, Port 5432

Anmeldedaten RZ HS Lu

- dbURL = "jdbc:postgresql://143.93.200.243:5432/BWUEBDB";
- pg-Server user user1, Passwort pgusers, Port 5432

Einrichtung Eclipse

Eclipse-Mars (4.5): Herunterladen von eclipse.org und Installation durch Entpacken

Workspace Z:\Eclipse-Mars-AS2-WiSe16

Dynamic Web Project BW4S

Package de.hs_lu.bw.jdbc mit JDBCAccess, PostgreSQLAccess, NoConnectionException

Package de.hs_lu.bw4s.beans

Window Preferences

- Java -> Installed JRE -> checken, dass da JDK 1.8 steht
- Java -> Compiler auch 1.8
- Server -> Runtime Environments -> TomCat 8.0
- General -> Editors -> Text Editors -> Spelling -> <Kästchen deaktivieren>

New Project -> Dynamic Web Project "BW4S" -> Target Runtime TomCat 8, dynamic web module 3.1

Projekteigenschaften (BW4S -> Properties), prüfen und ggfs. nachjustieren

- Java Compiler 1.8
- Project Facets -> bisher keine Änderung (Java 1.8. Dynamic Web Module 3.1)
- Targeted Runtimes -> TomCat 8
- Java Build Path -> Libraries ->
 - o Ggfs. Add Library -> Server Runtime -> TomCat 8 (geht nur, wenn TomCat in den Window Preferences schon eingetragen ist)
 - o Add External Libraries... -> D:\Programme\pgJDBC\postgresql-9.4.1208.jar
 - o Ggfs. Add Library -> Junit -> Junit 3
- Deployment Assembly -> Add... -> Java Build Path Entries -> postgresql-9.4.1208.jar