Inhaltsverzeichnis

Variationen zum Thema:
Internet
Ein Einführung in Java Enterprise

von Ralph P. Lano, 1. Auflage

Für wen

das Buch richtet sich an Bachelor-Studierende im vierten oder fünften Semester. Als Voraussetzung sollte man vor allem Java mitbringen. Außerdem schadet es nichts etwas über Datenstrukturen gehört zu haben, wir brauchen auf jeden Fall die Datencontainer List, Map und Set. Da das Internet ja ein Netzwerk von Computern ist, wäre auch eine Netzwerk Vorlesung nützlich. Hier wären wichtig die Protokolle TCP/IP und evtl. ein bisschen HTTP. Einfache HTML Kenntnisse sind hilfreich, die kann man sich aber auch sehr schnell noch selbst aneignen. Wenn man eine Vorlesung zu Software Engineering gehört hat, dann werden viele Dinge in diesem Buch wahrscheinlich etwas mehr Sinn machen. Man kann das Buch aber auch ohne verwenden. Das Buch ist eigentlich nicht zum Selbststudium gedacht, sondern als Begleitung zu einer Vorlesung. Aber ausschließen will ich es nicht.

Von wem

ich bin seit 2011 Professor für Internetprogrammierung und Multimediaapplikationen im Studiengang MediaEngineering an der Technischen Hochschule Nürnberg. Von 2003 bis 2010 war ich Professor für Softwaretechnik und multimediale Anwendungen an der Hochschule Hof, und von 2010 bis 2011 Professor für Media and Computing an der Hochschule für Technik und Wirtschaft Berlin. Ich promovierte 1996 an der University of Iowa zum Thema 'Quantum Gravity: Variations on a Theme'. Von 1996 bis 1997 war ich Postdoctoral Research Associate am Centre for Theoretical Studies des Indian Institute of Science. In der Zeit von 1997 bis 2003 war ich zunächst bei Pearson Education und später bei der Siemens AG in der Softwareentwicklung und dem Projektmanagement tätig.

Über was

die Hauptthemen in diesem Buch sind Java Server Pages, Object Relational Mapping und Web Services. Wir beginnen allerdings ganz einfach mit den Basics, klären einige Begriffe wie HTTP, CSS und JavaScript und verbringen das erste Kapitel mit ganz vielen, kleinen HTML Projekten. Alle diese Projekte werden dann in späteren Kapiteln mit Leben gefüllt. Dann wird die Idee hinter Java Server Pages vorgestellt, der Zusammenhang mit Java erklärt und die ersten Schritte in Richtung dynamischer Webseiten unternommen. Nebenbei werden auch die Themen Logging, Exceptions und der JSP Lebenszyklus behandelt. Im Kapitel Request and Response geht es vor allem um erste Interaktion mit den Nutzern. Dabei werden auch Hintergründe zum HTTP Protokoll gegeben. Die Projekte sollen ein gewisses Sicherheitsbewusstsein entwickeln, aber es werden auch Themen wie Reflection und Code Generation angesprochen. Mit den Session und Application Objekten beschäftigen wir uns ausführlich, zahlreiche Beispiele verdeutlichen die jeweiligen Anwendungsszenarien und erläutern den Unterschied. In den Projekten werden dann zahlreiche Beispiele aus dem ersten Buch "webifiziert". Persistenz ist das Thema dem wir die meisten Seiten widmen, dabei liegt der Fokus auf dem Object Relational Mapping. Mit einfachen Beispielen werden die Begriffe POJO, Dao und die verschiedenen X-to-Y Beziehungen zunächst vorgestellt und anschließend vertieft. Dabei wird auch klar warum Generics super cool sind. Zum Abschluss stellen wir Servlets und Filter vor, das Hauptthema sind aber Web Services der RESTvollen Art.

Wie

lernt man Internetprogrammierung? Wie alles, durch viel üben! Deswegen ist auch dieses Buch wieder voll mit Übungsbeispielen. Die Veranstaltung so wie ich sie unterrichte besteht aus drei Komponenten: der Vorlesung, der Übung und Hausaufgaben. Die Vorlesung ist zwei Stunden pro Woche und entspricht jeweils dem ersten Teil eines Kapitels im Buch. Ein Kapitel schaffen wir in ca. ein bis zwei Wochen. In den Übungen, die vier Stunden alle zwei Wochen stattfinden, widmen wir uns dann den Projekten. Dabei schaffen wir zwischen zwei und vier der Projekte pro Übung. In der Übung arbeiten die Studierenden in Teams, meist zu zweit, um sich gegenseitig zu helfen. Die Hausaufgaben werden im zweiwöchentlichen Rhythmus bearbeitet und benötigen ca. 4 bis 5 Stunden. Es ist wichtig, dass die Studierenden alleine an der Hausaufgabe arbeiten.

Wo

finde ich die Beispiele und den Quellcode? Die gibt es auf der Webseite zum Buch: www.VariationenZumThema.de. Auch Updates, Links zur Entwicklungsumgebung, das Buch in elektronischer Version gibt's dort. Das Buch selbst gibt's bei Amazon, in Schwarz-Weiß (billig) und in Farbe (teuer).

Darf ich

die Beispiele verwenden, oder das Buch kopieren? Dieses Material steht unter der Creative-Commons-Lizenz Namensnennung - Nicht-kommerziell - Weitergabe unter gleichen Bedingungen 4.0 International (CC-BY-NC-SA 4.0) D.h. Sie dürfen das Material in jedwedem Format oder Medium vervielfältigen und weiterverbreiten, das Material remixen, verändern und darauf aufbauen. Aber Sie müssen angemessene Urheber- und Rechteangaben machen, einen Link zur Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden. Diese Angaben dürfen in jeder angemessenen Art und Weise gemacht werden, allerdings nicht so, dass der Eindruck entsteht, der Lizenzgeber unterstütze gerade Sie oder Ihre Nutzung besonders. **Sie dürfen das Material nicht für kommerzielle Zwecke nutzen.** Und wenn Sie das Material remixen, verändern oder anderweitig direkt darauf aufbauen, dürfen Sie Ihre Beiträge nur unter derselben Lizenz wie das Original verbreiten und Sie dürfen keine zusätzlichen Klauseln oder technische Verfahren einsetzen, die anderen rechtlich irgendetwas untersagen, was die Lizenz erlaubt. Um eine Kopie dieser Lizenz zu sehen, besuchen Sie http://creativecommons.org/licenses/by-nc-sa/4.0/. Der Quellcode steht unter der MIT License (http://choosealicense.com/licenses/mit/).

Warum

dieses Buch? Seit 1996 programmiere ich Java, und seit ca. 2003 halte ich die Vorlesung "Internetprogrammieren" in verschiedenen Formen, Studiengängen und Hochschulen. All die Zeit habe ich immer wieder versucht meine Vorlesung an diesem oder jenem Buch zu orientieren. Habe aber nie eines gefunden, auch kein teures, das ich wirklich empfehlen könnte, außer evtl. das Buch von Herrn Stark [1], und das gibt's nicht mehr. Auch haben die meisten Bücher viel zu wenige Beispiele, vor allem kleine Beispiele. Deswegen.

Woher

kommen die Ideen? Mehr als die Hälfte der Ideen basieren auf meinem ersten Buch [2], das ja wiederum von Mehran Sahami's Vorlesung [3] inspiriert wurde, die ja wiederum auf dem Buch von Eric Roberts [4] basiert. Der Rest hat sich einfach so ergeben in den Jahren.

Referenzen

[1] J2EE (Master Class), von Thomas Stark

[2] Variationen zum Thema: Java: Eine spielerische Einführung, von Ralph P. Lano

[3] Programming Methodology, CS106A, von Mehran Sahami, https://see.stanford.edu/Course/CS106A

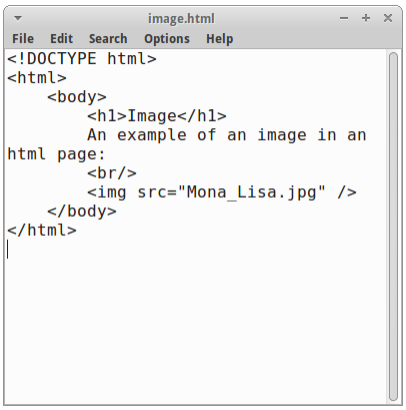
[4] The Art and Science of Java, von Eric Roberts, Addison-Wesley, 2008

Basics

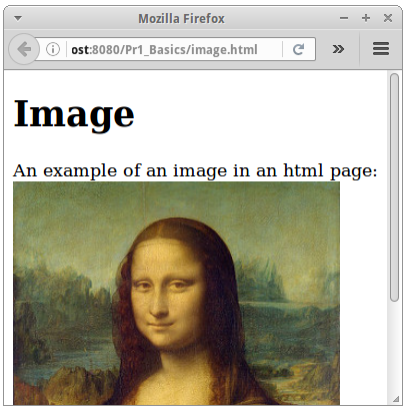
Am Anfang war HTML. Und damit hat das Web seinen Siegeszug begonnen. Ursprünglich war HTML auch ganz einfach. Und darauf wollen wir uns auch beschränken, auf die einfachen Seiten von HTML. Nach einer kurzen Einführung in HTML, vor allem Links und Forms, werden wir kurz HTTP und TCP/IP erwähnen und CSS und JavaScript ansprechen. Danach kommen dann ganz viele Beispiele an denen wir unsere HTML Kenntnisse vertiefen können.

Browser

Bevor wir uns aber mit HTML näher beschäftigen, wollen wir erst mal den Webbrowser kennen lernen. HTML ist zwar so einfach, dass man es in jedem beliebigen Editor ansehen kann, aber eigentlich ist es dafür gedacht mit einem Browser betrachtet zu werden. Vergleichen wir wie eine HTML Seite im Editor aussieht, damit wie sie in einem Browser aussieht:



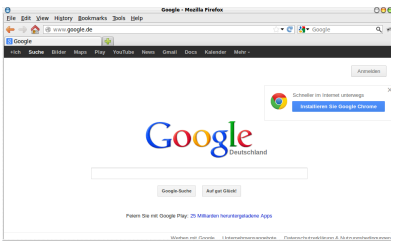
Editor

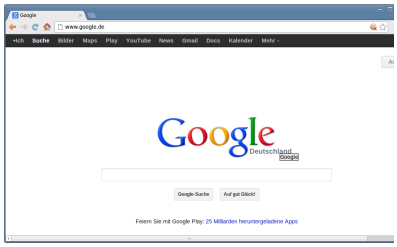


Browser

Es ist ziemlich offensichtlich, warum wir Browser verwenden wenn wir im Netz surfen und keine Editoren!

Die Erfindung von HTML geht einher mit der Erfindung des Browsers, und an beiden war Sir Tim Berners-Lee maßgeblich beteiligt. Browser gibt es inzwischen wie Sand am Meer, wie z.B. Mozilla's Firefox, Google's Chrome, aber auch text-basierte Browser wie etwa Lynx [6]:





Als Entwickler von Webseiten sollten wir darauf achten, dass unsere Seiten mit allen Browsern funktionieren. Speziell text-basierte Browser werden aber leider sehr häufig vernachlässigt. Das ist sehr Schade, denn vor allem blinde Personen sind auf text-basierte Browser angewiesen, da diese in der Regel auch über Text-to-Speech die Inhalte vorlesen können.

SEP: Alle unsere Webseiten sollten auch mit einem textbasierten Browser funktionieren, und man sollte seine Webapplikationen immer mit verschiedenen Browsern testen.

Markup

HTML steht für "HyperText Markup Language". Es handelt sich bei HTML also um eine Auszeichnungssprache, im Gegensatz zu Programmiersprachen, beispielsweise. Eine Auszeichnungssprache beschreibt ein Dokument. Zum Beispiel gibt es bei einem Buch einen Titel, einen Autor, eine Inhaltsangabe, eine Kapitelüberschrift, und natürlich auch Text, der meist in Paragraphen unterteilt ist. Eine Auszeichnungssprache für ein Buch, könnte also wie folgt aussehen:

```
<book>
    <title>Internetprogrammierung</title>
    <author>Ralph P. Lano</author>
    <chapter name="Basics">
        <paragraph>
        Am Anfang war HTML, ...
```

```
            </paragraph>
            ...
        </chapter>
        ...
</book>
```

Dabei erkennen wir auch schon die wichtigsten Merkmale von Auszeichnungssprachen:

- es gibt sogenannte "Tags" die charakteristisch für die jeweilige Auszeichnungssprache sind, wie z.B. <book>, <title>, etc.
- Tags kommen in der Regel paarweise, also auf ein <title> sollte später ein </title> folgen,
- und Tags sind verschachtelt, geben einem Dokument also eine gewisse Struktur.

HTML Markup

HTML ist die Auszeichnungssprache die Webseiten beschreibt. Es gibt über hundert verschiedene Tags, für unsere Zwecke sind aber nur ein paar davon wirklich wichtig. Jede HTML Seite beginnt mit einem <html> Tag, welches wiederum ein <head> und ein <body> Tag beinhalten kann:

```
<html>
    <head>
        <title>Title of Page</title>
    </head>
    <body>
        <h1>Hello Html!</h1>
        <p>Simple text goes inside &lt;p&gt; tags.</p>
        <ul>
            <li>this is list item 1</li>
            <li>this is list item 2</li>
        </ul>
    </body>
</html>
```

Der eigentliche Inhalt der Seite befindet sich innerhalb des <body> Tags, beschreibende Information befindet sich im <head> Tag.

SEP: Tags sollten immer geschlossen werden, also für jedes Start-Tag sollte es ein End-Tag geben.

Links

Was einen Browser von einem Editor unterscheidet sind nicht die Bilder, denn ein Textverarbeitungsprogramm wie OpenOffice kann das auch. Es sind die Links, also die Verbindung zwischen zwei Seiten. Ein Link, auch Hyperlink genannt, ist effektiv die Adresse eines Dokuments im Internet. Dieses Dokument kann eine Text oder HTML Datei sein, es kann aber auch ein Bild, eine MP3 Datei oder ein Film sein. Das HTML Tag für einen Link ist das <a> Tag:

```
<html>
    <body>
        <h1>Links</h1>
        This is an example of a simple
        <a href="forms.html">link</a>
        to another page.
    </body>
</html>
```

Hier wird die Seite "links.html" mit der Seite "forms.html" verlinkt. Klickt der Benutzer auf den Link wird die Seite"forms.html" im Browser geöffnet.

Man unterscheidet zwischen "relativen" Links, also Links die auf dem selben Server gehostet werden und "absoluten" Links, also einer festen Adresse im Internet.

SEP: Man sollte wenn möglich immer relative Links verwenden.

Forms

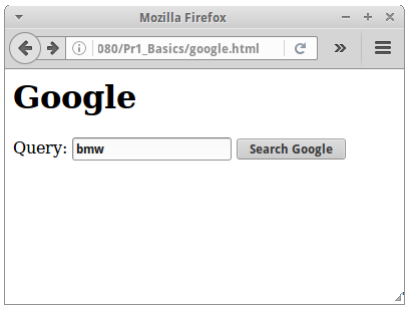
Das zweite was einen Browser ausmacht sind die Formulare. Mit Formularen kann man den Benutzer bitten Daten einzugeben, die dann ausgewertet werden können. In HTML benutzen wir dafür das <form> Tag:

```
<html>
    <body>
        <h1>Forms</h1>
        <form action="index.html" method="GET">
            UserID:
            <input type="text" name="userId"/>
            <br/>
            Password:
            <input type="password" name="password"/>
            <br/>
            <input type="submit" value=" Login "/>
        </form>
    </body>
</html>
```

Dabei ist "index.html" die Seite an die die Daten geschickt werden. Je nach Daten die man schicken möchte, gibt es verschiedene <input> Tags. Die folgenden Tags sind für uns die wichtigsten Tags:

```
<input type="text" name="lastname" value="Merkel" size="30"
maxlength="50">
<input type="password">
<input type="checkbox" checked="checked">
<input type="radio" checked="checked">
<input type="submit">
<input type="reset">
<input type="hidden">
< textarea name="Comment" rows="60" cols="20"></textarea>
<select>
  <option>Audi
  <option selected>BMW
  <option>VW
</select>
```

Es gibt aber noch ein paar andere die ganz nützlich sein können.



Google

Eine interessante Anwendung für das Form-Tag ist die Suche. Wir können eine Suche bei Google in unsere Webseite mit einbauen.

```
<html>
    <body>
        <h1>Google</h1>
        <form action="http://www.google.com/search" method="GET">
            Query:
            <input type="text" name="q"/>
            <input type="submit" value="Search Google"/>
        </form>
    </body>
</html>
```

Meta Tags

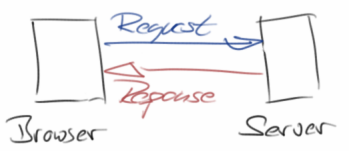
Meta Tags gehören zum Kopf Bereich einer HTML Seite. Sie geben zum Einen beschreibende Informationen zur Seite, wie z.B. description, keywords, author und robots Tag, die vor allem für Suchmaschinen interessant sein sollten. Allerdings, werden sie häufig von selbigen ignoriert.

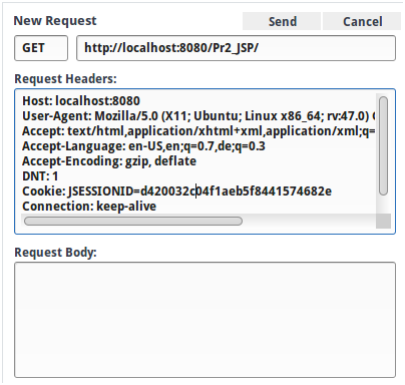
```
<html>
    <head>
        <title>Meta Tags</title>
        <meta name="description"
              content="Here you can place a short description
              of your pages content."/>
        <meta name="keywords"
              content="comma separated list of keyword,
              might be ignored by search engines, html, meta tag"/>
        <meta name="author" content="Ralph P. Lano"/>
        <meta charset="UTF-8"/>
        <meta name="robots" content="noindex, nofollow"/>
        <meta http-equiv="refresh" content="10; url=index.html" />
    </head>
    ...
</html>
```

Interessant ist allerdings das "refresh" Tag. In dem Beispiel oben besagt es, dass nach 10 Sekunden die Seite "index.html" geladen werden soll. Wir werden gleich mehrere Beispiele sehen wo das nützlich ist.

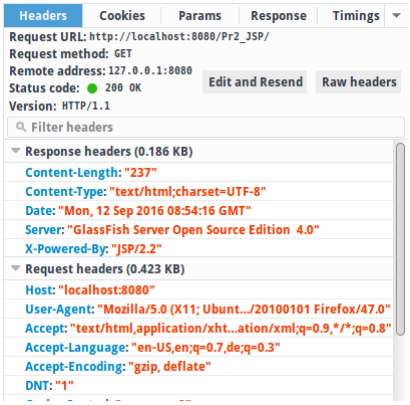
HTTP

Das Internet basiert auf dem HTTP Protokoll, ohne würde nichts gehen. Wenn wir im Browser "www.google.de" eingeben, dann sendet der Browser einen HTTP Request an den Google Server. Der Server antwortet dann mit einer HTTP Response und liefert in der Regel das was der Browser wollte.





Im HTTP Request sagt der Browser was er will, er enthält also Informationen über das gewünschte Dokument, oder allgemeiner die gewünschte Ressource. Der Request enthält aber noch zusätzliche Informationen, wie z.B. die Sprache die der Nutzer spricht, welchen Browser und Betriebssystem er verwendet, die IP Adresse des Computers von dem der Request geschickt wurde, aber auch Cookies und u.U. auch Formulardaten, falls der Nutzer ein Formular ausgefüllt hat.

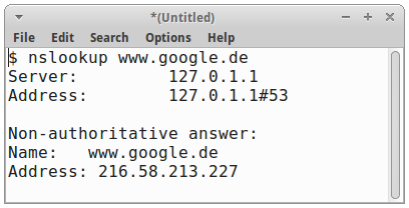


Die HTTP Response enthält natürlich das gewünschte Dokument. Das ist im sogenannten HTTP Body. Der HTTP Header enthält zusätzliche Informationen, wie z.B. den HTTP Status Code, die IP Adresse des Servers, den Datum, Cookies, und Informationen über das Dokument, dessen Typ (Content-Type) und dessen Größe (Content-Length).

Der Browser sendet immer als erstes einen Request und der Server antwortet mit der Response. Es geht nie umgekehrt. Sendet der Browser nix, antwortet der Server auch nicht. Das ist ganz zentral.

Das HTTP Protokoll gibt es in zwei Versionen, und zwar unverschlüsselt (http) und verschlüsselt (https).

TCP/IP



Wie weiß denn der Browser wo der Server überhaupt ist? Wenn wir im Browser "www.google.de" eintippen, dann ist das in etwas so wie wenn ich sage: "ruf mal den Ralph an". Wir müssen natürlich die Telefonnummer von Ralph wissen, und genau das sind IP Adressen im Internet. Und so wie es für Telefonnummern ein Telefonbuch gibt, so gibt es für *IP Adressen* den Domain Name Service (DNS). Der Browser fragt also den DNS, "Hey was ist denn die IP Adresse von Google?". Und der DNS sagt ihm dann, dass die *IP Adresse* von Google "216.58.213.227" ist. Wir können im Browser auch "216.58.213.227" anstelle von "www.google.de" eingeben, das ist dem Browser sogar lieber, weil es ihm Arbeit spart.

Eine Sache noch, und das sind die Ports. Wenn wir so mit dem Browser unterwegs sind, dann verwendet der zusätzlich zur IP (damit findet er den Server) auch noch den Port, genauer den TCP-Port. Normalerweise verwendet HTTP den Port 80, und wenn wir diesen Port verwenden, müssen wir auch nichts sagen. Der Browser denkt sich einfach, "oh der Typ hat nix gesagt, also muss er wohl Port 80 meinen". Wenn der Typ aber den Port angibt, wie z.B. "http://localhost:8080", dann ist die Zahl hinter dem Doppelpunkt, also die "8080" der Port. Insgesamt gibt es etwas über 65000 Ports. Das hört sich jetzt nach viel an, ist es aber nicht. Denn im Internet gibt es viele Nutzer. Und die Zahl 65000 heißt, dass wir maximal 65000 Nutzer gleichzeitig an unserem Server haben können. Und für Websites wie Amazon oder Facebook ist das ein Problem. Deswegen haben die auch so viele Server.

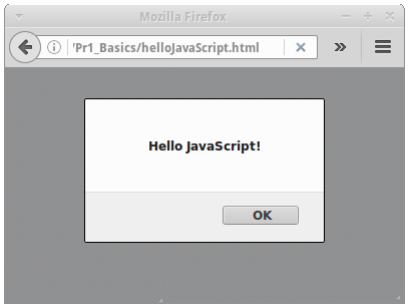
CSS

HTML beschreibt *was* dargestellt werden soll, CSS beschreibt *wie* es dargestellt werden soll. Ein einfaches Beispiel soll zeigen wie wir z.B. Überschriften in blau erscheinen lassen:

```
<html>
    <head>
        <style type="text/css">
            h1 {
                color: blue;
            }
        </style>
    </head>
    ...
</html>
```

Cascading Style Sheets (CSS) sind allerdings nicht Gegenstand dieses Buches, dafür gibt es bereits genügend andere gute Bücher.

JavaScript



Auch für JavaScript gibt es bereits eine Menge anderer guter Bücher. Hier wollen wir nur kurz zeigen, wie man JavaScript in HTML einbettet:

```
<html>
    <body>
        <script type="text/javascript">
            alert("Hello JavaScript!");
        </script>
    </body>
</html>
```

Ansonsten werden wir JavaScript in diesem Buch nur sehr selten verwenden. Aber im Anhang finden sich noch ein paar Zeilen zu JavaScript.

Review

Was haben wir in diesem Kapitel gelernt? Wir haben gelernt

- dass es verschiedene Browser gibt,
- wie Links und Forms funktionieren,
- wofür Meta Tags gut sind,
- kurz gehört was HTTP ist und was eine HTTP Request und die entsprechende HTTP Response ist,
- und was es mit TCP/IP auf sich hat.

Dass es CSS gibt und auch JavaScript wurde zwar erwähnt, mehr aber auch nicht.

Projekte

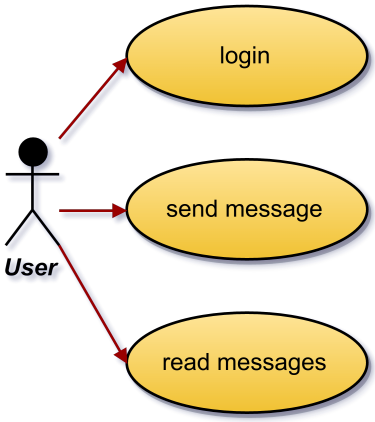
Mit den Projekten wollen wir unsere HTML Kenntnisse an Beispielen vertiefen. Alle Beispiele die wir hier behandeln, werden wir später im Buch wiedersehen, deswegen sollten wir so viele wie möglich versuchen umzusetzen. Es kommt hier nicht darauf an, dass die Seiten super hübsch oder genauso wie angedeutet aussehen, es geht vor allem um die Funktionalität.

Die Anforderungen für jedes Projekt werden kurz beschrieben, begleitet von einem UseCase Diagramm und einer Liste der zentralen UseCases. Daraus ergibt sich dann die Seitenstruktur der Webanwendung. Das mag am Anfang nicht sofort nachvollziehbar sein, wird aber mit der Zeit ganz logisch erscheinen. Es folgen dann Screenshots oder auch Mockups der jeweiligen Seiten. Diese sollen dann in HTML umgesetzt werden. Das ist das Ziel.

Obwohl es sich im Moment noch um reine HTML Seiten handelt und sie deshalb ja eigentlich auf ".html" enden müssten, werden wir die Seiten gleich mit ".jsp" enden lassen, da wir in späteren Kapiteln daraus ja JSP Seiten machen werden, und wir sparen uns dann viel Umbenennarbeit.

Falls wir einen kleinen HTML Auffrischer benötigen, dann sind die Tutorials der Website W3Schools [1] sehr hilfreich.

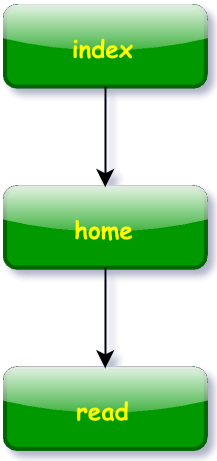
Messenger



Bei Messanger handelt es sich um eine kleine Anwendung in der sich verschiedene Nutzer Messages senden können, ähnlich wie Email. Es gibt eine Login Seite, auf der man sich einfach mit einem Alias anmeldet. Von dort gelangt man auf die Home Seite. Die erlaubt es einem Messages an andere Nutzer zu senden und hat einen Link auf die Read Seite mit der man seine Messages lesen kann. Zusammengefasst also:

- sich mit einem Alias beim Messenger einloggen
- eine Message schreiben und versenden
- eigene Messages lesen.

Daraus ergibt sich folgende einfache Seitenstruktur:



Was die Mockups angeht, könnten die wie folgt aussehen:

index.jsp *home.jsp* *read.jsp*

Die erste Seite, index.jsp, enthält ein einfaches Formular mit einem Textfeld mit dem Namen "alias" und einen Submit Knopf. Das Ziel des Formulars soll home.jsp sein.

Auf der Seite home.jsp soll es einen Link zur Seite read.jsp geben. Außerdem soll das Formular der Seite aus einer 2 mal 40 großen Textarea bestehen, einem Hidden Feld, einem Select Tag und einem Submit Knopf. Das Ziel des Formulars soll auch home.jsp sein.

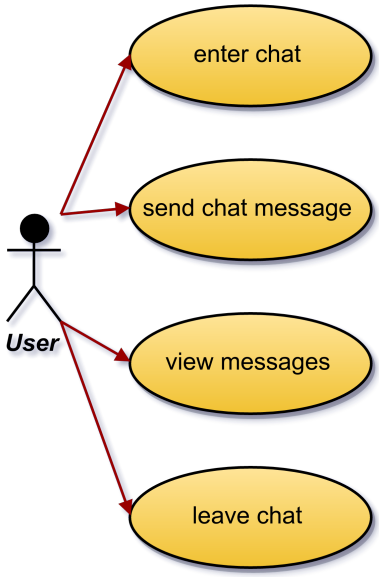
```
<input type="hidden" name="senderId" value="ralph"/>
...
<select name="receiverId">
    <option value='ralph'>ralph</option>
    <option value='vince'>vince</option>
</select>
```

Die Seite read.jsp soll einen Link zurück zur Seite home.jsp haben, und ein paar Beispiel Messages als Aufzählungsliste (bulleted list).

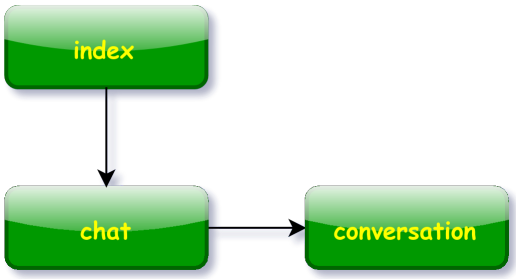
TwoPlayer Chat

Ein sehr interessantes Problem ist ein Chat zwischen zwei Personen. Wir wollen uns hier mit dem Fall beschäftigen in dem sich die zwei Personen nicht kennen. Das mag jetzt für einen Chat etwas ungewöhnlich sein, aber für Spiele in denen zwei zufällige Spieler gegeneinander spielen wollen, ist es nicht so ungewöhnlich, dieser Fall tritt sogar bei sehr vielen Spielen im Netz auf. Die UseCases sind die folgenden:

- den Chat beginnen, evtl. auf einen Partner warten
- Chat Messages schreiben
- Chat Messages ansehen
- den Chat verlassen.



Daraus ergibt sich folgende Seitenstruktur:



Was die Mockups angeht, könnten die wie folgt aussehen:

index.jsp

chat.jsp

Die erste Seite von TwoPlayer soll einfach aus etwas Text und einem Link zu Seite chat.jsp bestehen. Zusätzlich soll im <head> Teil der Seite ein Meta-Refresh Tag sein:

```
<head>
    <meta http-equiv="refresh" content="5" />
</head>
```

Die Seite chat.jsp hat zum einen ein Formular mit einem Textfeld namens "msg", einen Submit Knopf, und das Ziel des Formulars soll chat.jsp sein. Des weiteren soll es einen Link "Reset" geben der zur Seite home.jsp führt. Außerdem soll es auf der Seite einen iFrame geben:

```
<iframe src="conversation.jsp" height="100"></iframe>
```

iFrames sind sozusagen Webseiten innerhalb von Webseiten, manchmal total praktisch, sollten aber nicht zu oft verwendet werden.

Schließlich benötigen wir noch die Seite conversation.jsp. Das ist eine ganz einfache HTML Seite mit ein bisschen Text, aber auch diese Seite sollte ein Meta-Refresh Tag haben, genauso wie die Seite index.jsp.

Navigation

Alle der nachfolgenden Webseiten haben eine Navigationszeile. Navigationszeilen gibt es in vielen verschiedenen Formen, aber gemeinsam ist ihnen allen, dass sie immer gleich sind. Wie wir ja schon im ersten Semester gehört haben, sind Programmierer relativ faule Leute und wenn möglich vermeidet man doppelten Code. Wir definieren also eine *navigation.jsp* Seite:

```
<small>
    <a href="login.jsp">login</a> /
    <a href="logout.jsp">logout</a> /
    <a href="protected.jsp">protected</a>
</small>
```

und wollen, dass alle unsere Webseiten diese Navigationszeile enthalten. Das macht man mit einer "include directive":

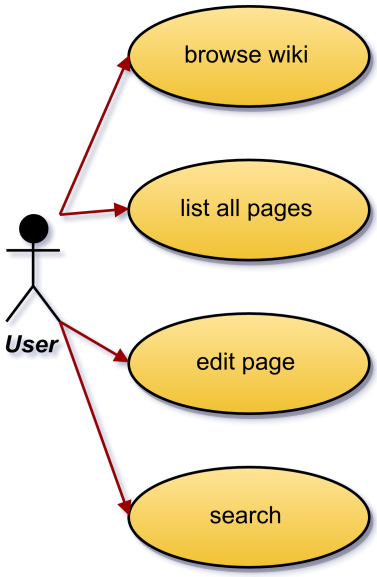
```
<html>
    <body>
        <%@include file="navigation.jsp" %>
        <h1>Login</h1>
        ...
    </body>
</html>
```

Genau verstehen müssen wir das noch nicht, aber es ist super-praktisch.

SEP: Wir sollten doppelten Code vermeiden.

Wiki

Wir alle kennen die Wikipedia. Und man denkt, dass das eine super-komplizierte Webanwendung sein muss. Interessant ist, dass wir sehr bald schon in der Lage sind ein kleines Wiki selbst zu schreiben, also sozusagen "Wikipedia 0.1".

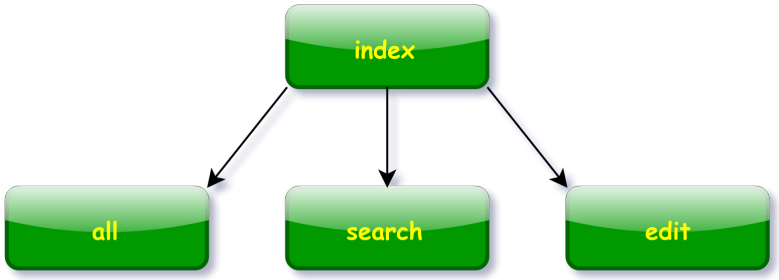


Unsere Anforderungen an unser Wiki sind folgende: Wir möchten zum einen ganz normal in unserem Wiki browsen und navigieren können, also Links folgen. Wir möchten jede beliebige Seite editieren können. Wir möchten auch neue Seiten anlegen können, in dem wir wie in Wikis üblich, einfach einen neuen Link zu einer noch nicht existierenden Seite einfügen, und das System dann, beim ersten Aufruf, diese Seite erzeugt. Schließlich, möchten wir alle Seiten in unserem Wiki auflisten können, und natürlich möchten wir unser Wiki auch durchsuchen können.

Wenn wir das kurz zusammenfassen ergeben sich folgende UseCases:

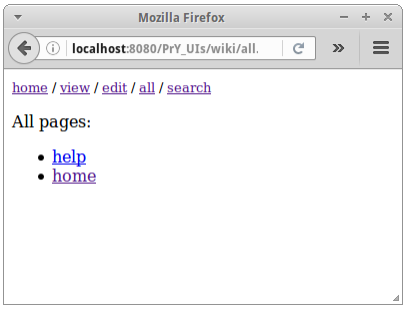
- durch das Wiki browsen
- alle Wiki Seiten auflisten
- neue Seiten erstellen
- nach Schlagwörtern suchen.

Daraus ergibt sich folgende Seitenstruktur:



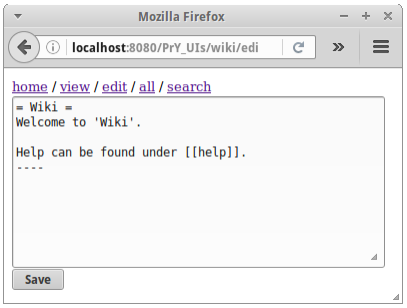
Was die Mockups angeht, könnten die wie folgt aussehen:

index.jsp



all.jsp

search.jsp



edit.jsp

Die Seiten sind eigentlich trivial, das Einzige ist, dass wir in jeder Seite die Include Directive nicht vergessen:

```
<html>
    <body>
        <%@include file="navigation.jsp" %>
        <h1>Login</h1>
        ...
    </body>
</html>
```

Der Link "help" auf der index.jsp Seite, führt wieder zurück zur index.jsp Seite (macht noch nicht viel Sinn, kommt später). Das Gleiche gilt für die Links auf der Seite all.jsp.

Das Ziel des Formulars auf search.jsp ist auch wieder index.jsp. Neben dem Submit Knopf, soll es ein Inputfeld mit dem Namen "searchTerm" geben und ein Hidden-Tag mit dem Namen "search". Wert braucht das Hidden-Tag keinen.

Auf der Seite edit.jsp gibt es wieder ein Formular mit einer 10 mal 50 großen TextArea, sowie zwei Hidden-Tags, eines mit Namen "page" und Wert "home" und ein zweites mit Namen "edit". Das Ziel des Formulars soll eine Seite "wikiLogic.jsp" sein, in die wir einfach folgenden Text reinschreiben:

```
Not yet implemented.
```

(Also einfach Text, kein HTML)

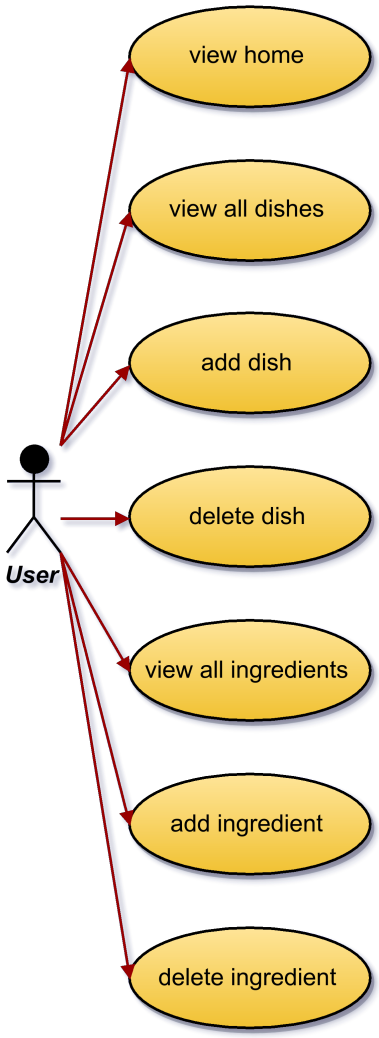
Mensa

Aus dem ersten Semester stammt das Mensa Beispiel. Damals haben wir es verwendet um die Objektorientierte Analyse zu üben. Jetzt werden wir das Beispiel als Webanwendung umsetzen. Erinnern wir uns an die Anforderungen:

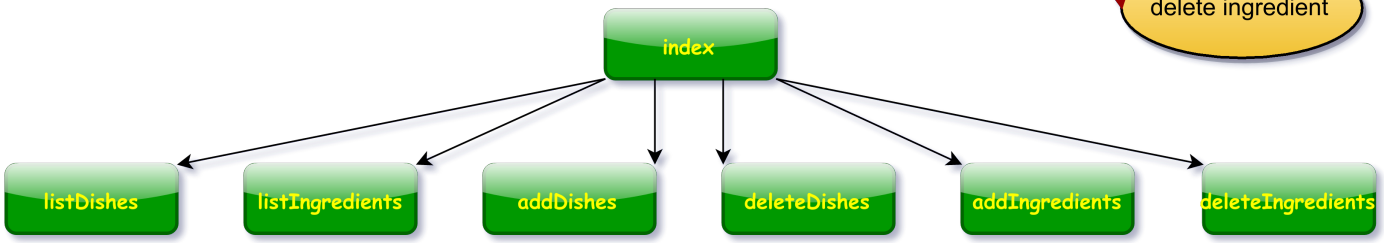
Die Mensa hat Gerichte und Zutaten. Ein Gericht hat einen Namen, einen Preis und eine Liste von Zutaten. Eine Zutat hat einen Namen, einen Preis und Kalorien. Wir können alle Gerichte anzeigen, neue Gerichte angelegen, sowie existierende Gerichte löschen. Wir können alle Zutaten auflisten, die zu einem Gericht gehören. Wir können neue Zutaten anlegen und wir können existierende Zutaten löschen, sowie alle Zutaten auflisten.

Wenn wir das zusammenfassen, speziell im Hinblick auf die Webseiten die wir benötigen, ergeben sich folgende UseCases:

- erste Seite mit heutigem Menü: index.jsp
- alle Gerichte anzeigen
- neues Gericht hinzufügen
- bestehendes Gericht löschen
- alle Zutaten auflisten
- neue Zutaten hinzufügen
- bestehende Zutat löschen.



Daraus ergibt sich folgende Seitenstruktur:



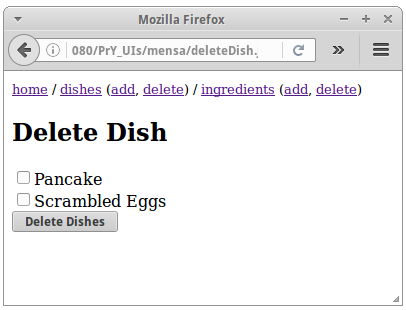
Was die Mockups angeht, könnten die wie folgt aussehen:

index.jsp

listDishes.jsp

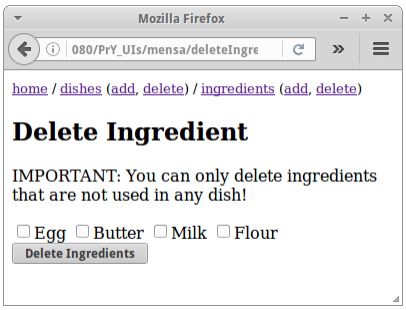
listIngredients.jsp

addDish.jsp



deleteDish.jsp

addIngredient.jsp



deleteIngredient.jsp

Mit der Übung die wir jetzt schon haben, sollten die Seiten ein Klacks sein. Die Ziele der jeweiligen Formulare ergeben sich aus der Seitenstruktur. Auch, alle Seiten haben wieder eine Navigationszeile. Was evtl. neu ist, sind die Checkboxes, z.B. für die addDish.jsp Seite würde das so aussehen:

```
<input type='checkbox' name='ingredient' value='Egg'/>Egg
<input type='checkbox' name='ingredient' value='Butter'/>Butter
<input type='checkbox' name='ingredient' value='Milk'/>Milk
<input type='checkbox' name='ingredient' value='Flour'/>Flour
```

Alle Formulare sollten als Ziel die Seite index.jsp haben. Und jedes Formular sollte noch ein Hidden-Tag beinhalten

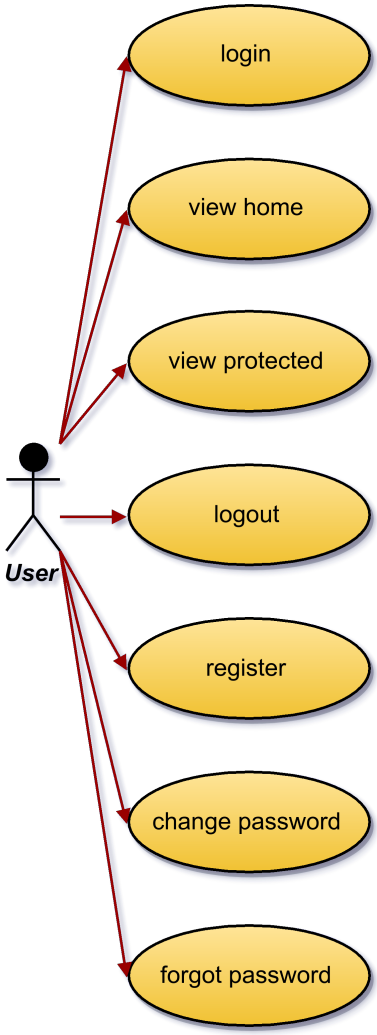
```
<input type="hidden" name="addDish"/>
```

mit dem Namen der Seite in der es sich befindet.

Login

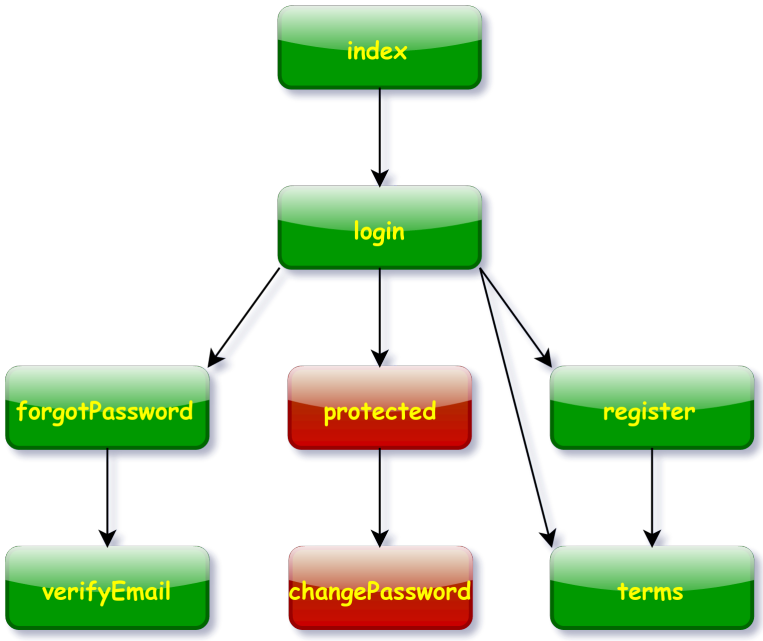
Für fast jede Webanwendung gibt es einen Login. Es gibt also einen öffentlichen und einen geschützten Bereich. Wir wollen das in einem einfachen Beispiel durchspielen. Unsere Login Anwendung soll folgende Anforderungen erfüllen:

- öffentliche Seiten: index.jsp, terms.jsp
- eine geschützte Seite: protected.jsp
- man soll sich einloggen können: login.jsp
- man soll sich ausloggen können: logout.jsp
- man soll sich registrieren können: register.jsp
- Passwort ändern können: changePassword.jsp
- Passwort vergessen dürfen: forgotPassword.jsp.



Das sind so in etwas die Seiten die die meisten Websites bieten rund um das Login.

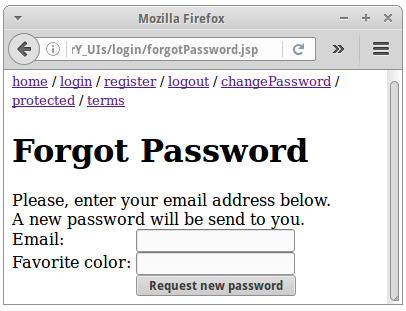
Daraus ergibt sich folgende Seitenstruktur:



Was die Mockups angeht, könnten die wie folgt aussehen:

index.jsp

login.jsp



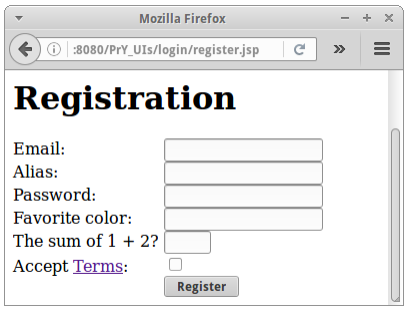
forgotPassword.jsp

.

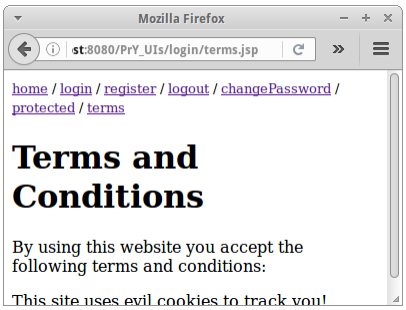
verifyEmail.jsp

protected.jsp

changePassword.jsp



register.jsp



terms.jsp

Alle Seiten haben wieder eine gemeinsame Navigationsleiste. Die Seiten home.jsp, protected.jsp und tems.jsp sind absolut trivial.

Die Seiten logout.jsp und verifyEmail.jsp bestehen einfach aus einem

```
Not yet implemented.
```

Die Seite login.jsp besteht aus einem Formular mit einem Textfeld mit Namen "emailId" und einem Passwortfeld mit Namen "password", sowie einem Hidden-Tag namens "login". Das Ziel des Formular soll protected.jsp sein, und ganz wichtig die Methode sollte "POST" sein. Außerdem soll es einen Link auf die Seite forgotPassword.jsp geben.

Die Seite forgotPassword.jsp besteht aus einem Formular mit zwei Textfeldern mit Namen "emailId" und "favoriteColor", sowie ein Hidden-Tag namens "forgotPassword". Das Ziel des Formular soll verifyEmail.jsp sein, und ganz wichtig ist wieder die Methode sollte "POST" sein.

Die Seite changePassword.jsp besteht aus einem Formular mit drei Passwortfelder mit den Namen "password", "newPassword1" und "newPassword2", sowie einem Hidden-Tag namens "changePassword". Das Ziel des Formular soll verifyEmail.jsp sein, und ganz wichtig ist wieder die Methode sollte "POST" sein.

Die Seite register.jsp besteht aus einem Formular mit drei Textfeldern mit Namen "emailId", "alias" und "favoriteColor", einem Passwortfeld mit Namen "password", sowie ein Hidden-Tag namens "registration". Außerdem soll die Seite ein numerische Feld namens "sum" enthalten mit assoziierten Hidden-Tag namens "result":

```
<input type="number" name="sum" size="4"/>
<input type="hidden" name="result" value="3"/>
```

Und wir benötigen noch eine Checkbox für die Terms and Conditions:

```
<input type="checkbox" name="acceptTerms"/>
```

Das Ziel des Formular soll protected.jsp sein, und ganz wichtig ist wieder die Methode sollte "POST" sein. Schließlich soll es noch einen Link auf die Seite terms.jsp geben.

Wir haben es bisher vermieden CSS Stylesheets zu verwenden, aber hier macht es Sinn eine Ausnahme zu machen. Wir verwenden das folgende Stylesheet namens "style.css":

```
form  { display: table;      }
p     { display: table-row;  }
label { display: table-cell; white-space: nowrap; padding-right: 5px;}
input { display: table-cell; }
```

Darin werden Styles für die Tags *form*, *p*, *label* und *input* definiert. Alles was wir tun müssen, ist im Header jeder JSP Seite das Stylesheet einzubinden:

```
<head><link rel="stylesheet" type="text/css" href="style.css"></head>
```

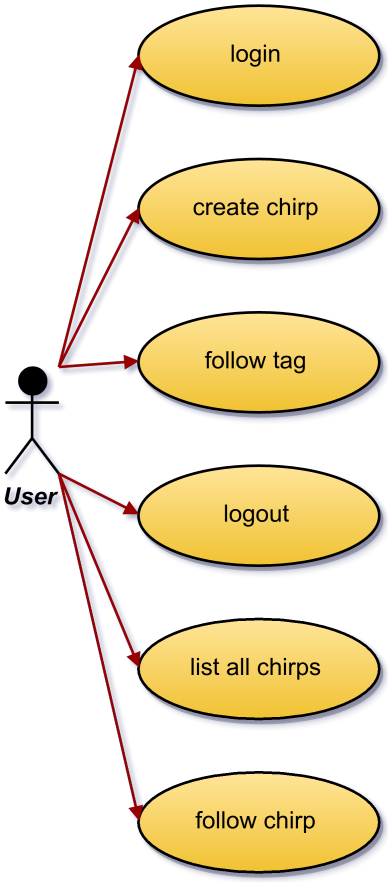
und wenn wir unsere Formulare definieren auf folgende Struktur zu achten:

```
<form action="protected.jsp" method="POST">
    <p>
        <label>Email:</label>
        <input type="text" name="emailId"/>
    </p>
    <p>
        <label>Password:</label>
        <input type="password" name="password"/>
    </p>
    ...
</form>
```

Dann sehen die Formulare viel angenehmer aus.

Chirpr

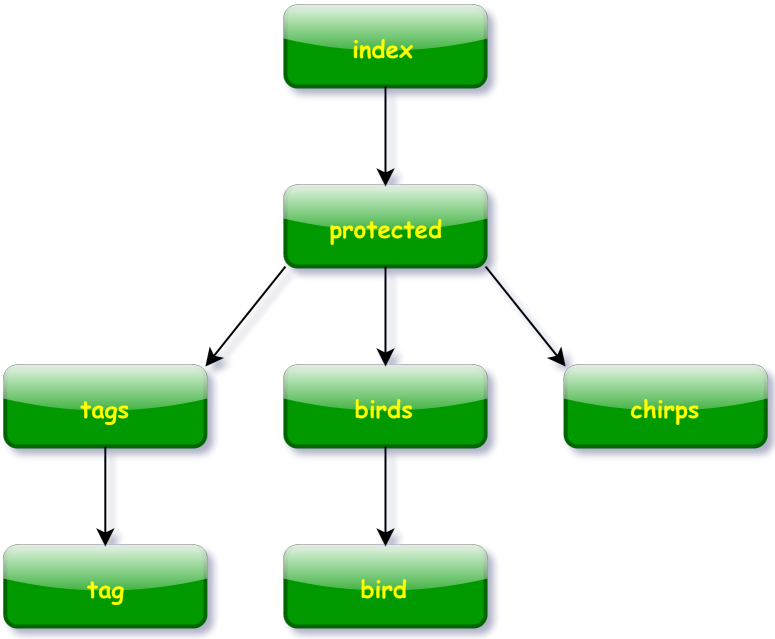
"Zwitschern" heißt auf Englisch "chirp", und darum geht es bei Chirpr: es handelt sich um ein Mini-Soziales-Netzwerk in dem man kurze Nachrichten (Chirps) mit maximal 42 Zeichen posten kann. Man kann in seinen Nachrichten auch Tags verwenden, um diese einem gewissen Thema zuzuordnen. Das Ganze ist natürlich von einer relativ bekannten Webanwendung die mit Vögeln zu tun hat inspiriert.



Beginnen wir wieder mit unseren Anforderungen: Nutzer sollen

- sich identifizieren: login.jsp
- neue Chirps erstellen können: protected.jsp
- gewissen Tags folgen können: tags.jsp
- gewissen Birds folgen können: birds.jsp
- alle Chirps auflisten können: chirps.jsp
- sich ausloggen können: logout.jsp.

Daraus ergibt sich folgende Seitenstruktur:



Was die Mockups angeht, könnten die wie folgt aussehen:

index.jsp

protected.jsp

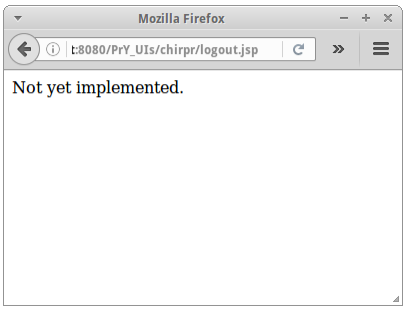
tags.jsp

tag.jsp

birds.jsp

bird.jsp

chirps.jsp

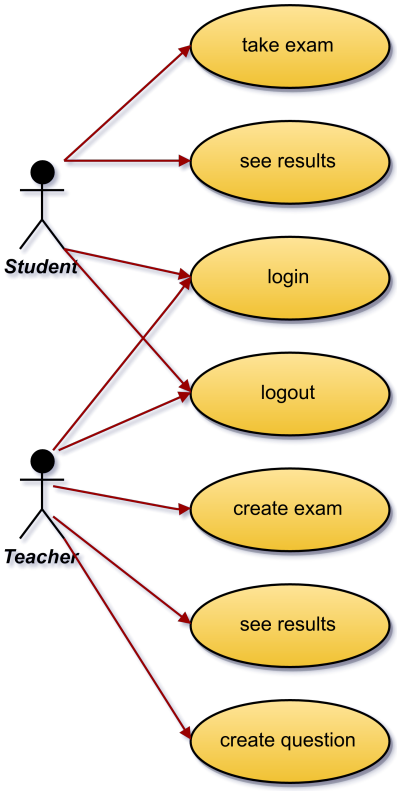


logout.jsp

Hier gibt es nicht viel zu sagen, aus der Seitenstruktur ergeben sich die Links, und mit dem bisher Gelernten sind die Seiten ganz einfach zu erstellen.

Quizs

Quizs ist eine Webanwendung mit der man anhand von Multiplechoicefragen sein Wissen überprüfen kann. Außerdem kann man auch Prüfungen erstellen. Auch diese Anwendung ist bereits aus dem ersten Semester bekannt, damals haben wir Teile als Swing Anwendung programmiert aber natürlich ohne Datenbank.



Beginnen wir wieder mit unseren Anforderungen. Aus der Sicht eines Studierenden können wir

- uns identifizieren: login.jsp
- an einer Prüfung teilnehmen: exam.jsp
- die Resultate einsehen: results.jsp
- uns ausloggen: logout.jsp.

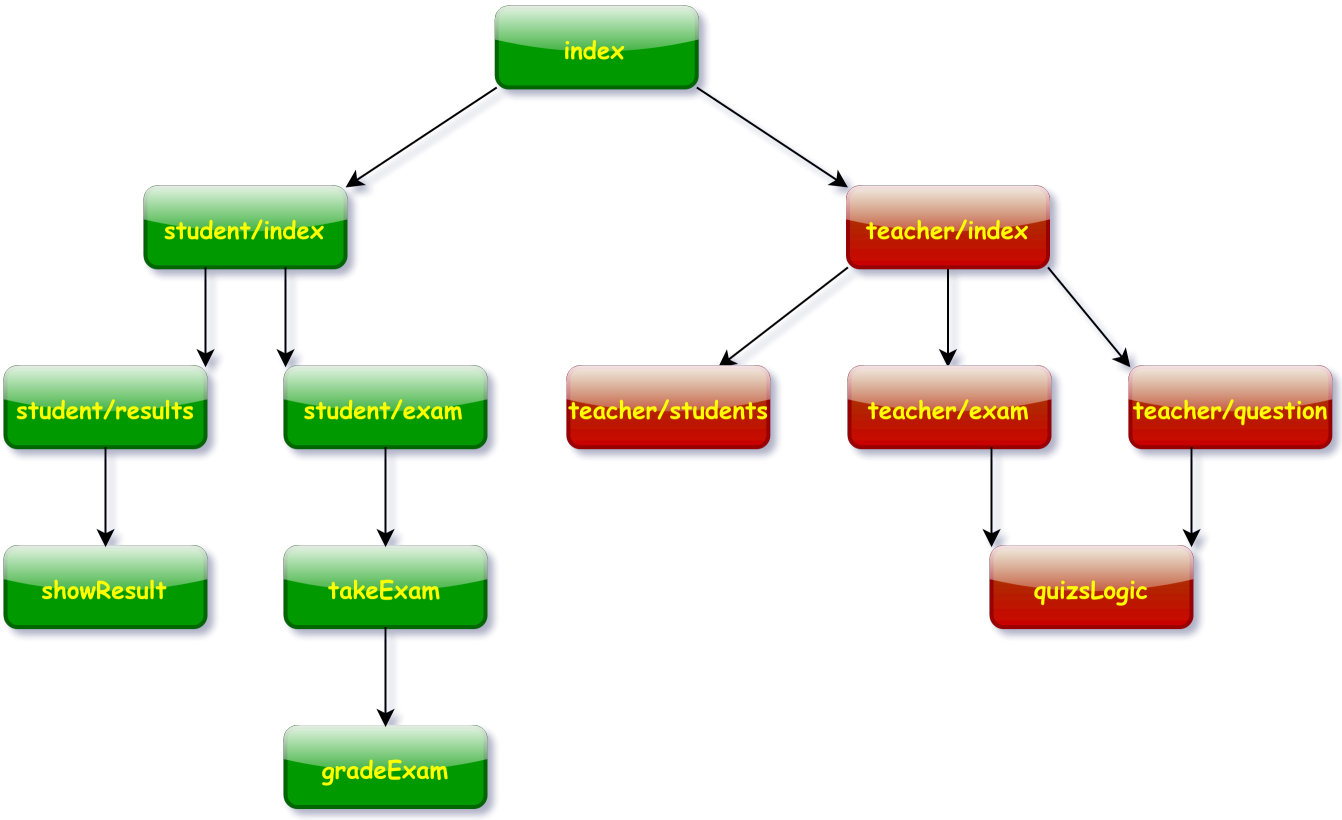
Falls ein Studierender noch nicht existiert, wird einfach ein neuer angelegt.

Aus Sicht eines Lehrenden können wir

- uns identifizieren: login.jsp
- neue Fragen erstellen: question.jsp
- neue Prüfungen erstellen: exam.jsp
- alle Studierende auflisten: results.jsp
- uns ausloggen: logout.jsp.

Lehrende soll es nur einen geben: den 'teacher'. Wenn sich also jemand mit der ID 'teacher' einlogged, hat er die teacher Privilegien. Es gibt also zwei verschiedene Nutzer-Typen in dieser Webanwendung.

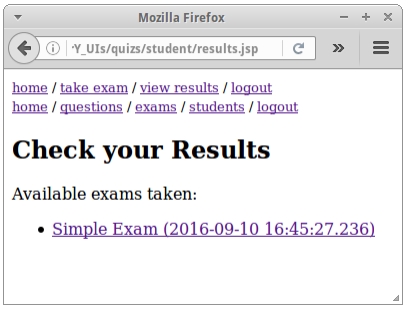
Daraus ergibt sich folgende Seitenstruktur:



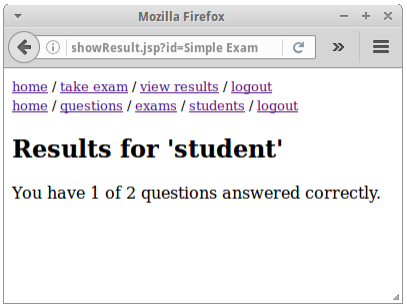
Was die Mockups angeht, könnten die wie folgt aussehen:

index.jsp

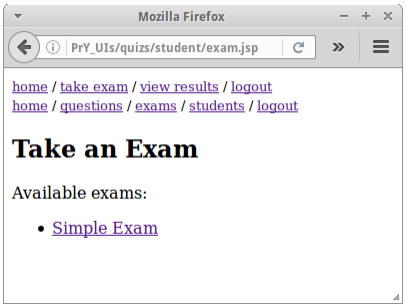
student/index.jsp



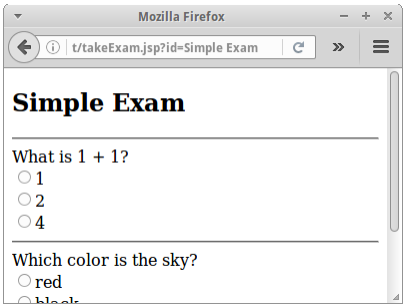
student/results.jsp



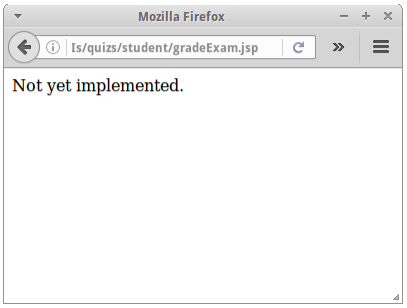
student/showResult.jsp



student/exam.jsp

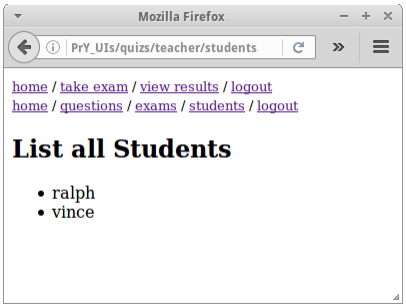


student/takeExam.jsp

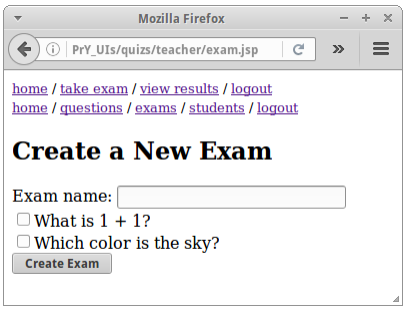


student/gradeExam.jsp

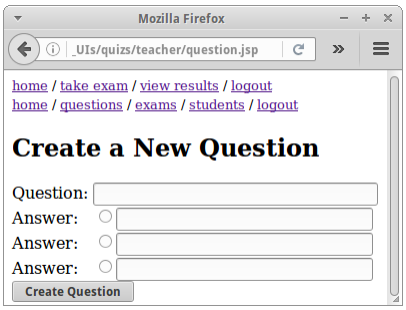
teacher/index.jsp



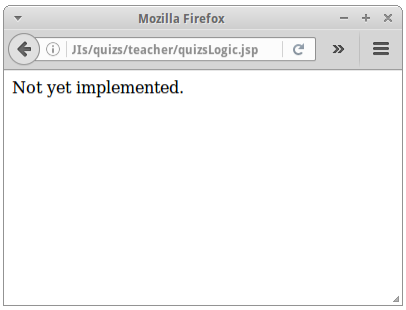
teacher/students.jsp



teacher/exam.jsp



teacher/question.jsp



teacher/quizsLogic.jsp

Auch hier gibt es nicht viel zu sagen, aus der Seitenstruktur ergeben sich die Links, und auch diese Seiten sind wieder ganz einfach zu erstellen. Wir sollten nur darauf achten, dass alle Seiten für Studierende sich im Unterverzeichnis "/student/" befinden, und alle Seiten für Lehrende im Unterverzeichnis "/teacher/". Evtl. eine Anmerkung zur Seite takeExam.jsp: wie gruppiert man Radiobuttons? Das geht über den Namen:

```
What is 1 + 1?<br/>
<input type='radio' name='0' value='1'/>1<br/>
<input type='radio' name='0' value='2'/>2<br/>
<input type='radio' name='0' value='3'/>4<br/>
<hr/>
Which color is the sky?<br/>
<input type='radio' name='1' value='1'/>red<br/>
<input type='radio' name='1' value='2'/>black<br/>
<input type='radio' name='1' value='3'/>blue<br/>
<hr/>
```

Radiobuttons mit dem gleichen Namen gehören zusammen.

Research

In diesem Kapitel haben wir uns auf das absolut Notwendige beschränkt. Es gibt aber natürlich noch viele andere Dinge die wir im Internet recherchieren können.

Browsers

Leider ist es nicht so, dass unsere Webseiten auf allen Browsern gleich aussehen oder, wenn wir sehr viel JavaScript oder ähnliches verwenden, überhaupt funktionieren. Deswegen muss man seine Webanwendung auf den verschiedenen Browser testen.

Es kommt häufig vor, dass unser Kunde verlangt, dass seine Seiten auf 90% aller Webbrowser laufen sollen. Um zu wissen was 90% bedeutet muss man natürlich erst einmal wissen welche Browser wie populär sind. Das letzte Mal als ich nachgesehen habe, war das folgende Popularität:

- Google Chrome (72%)
- Firefox (17%)
- Internet Explorer (5%)
- Safari (3%)
- Opera (1%)

In diesem Fall müssten wir unsere Anwendung also mindestens auf den ersten drei Browsern testen. Da sich diese Zahlen innerhalb von ein paar Monaten ändern, sollten wir jetzt mal recherchieren, wie denn die Verteilung heute aussieht: http://www.w3schools.com/browsers/default.asp. Auch zu beachten ist in wiefern unsere Anwendung auf mobilen Endgeräte laufen soll.

Markup Languages

HTML ist eine Markup Sprache. Markup Sprachen sollte man nicht mit Programmiersprachen verwechseln, denn das sind sie nicht. Neben HTML gibt es noch ganz viele, wichtig ist XML, interessant sind aber auch die Ursprünge SGML und GML. HTML5 ist was heute meistens verwendet wird, aber man findet auch noch ältere Versionen im "Feld". Es macht Sinn sich im Internet mal zu den Hintergründen all dieser "ML" Sprachen zu erkundigen.

MIME

Die "Multipurpose Internet Mail Extensions", kurz MIME, sind extrem wichtig wenn wir irgendetwas im Internet machen. Die wichtigste ist wohl "text/html" oder "image/png", aber es gibt noch ganz viele andere. Auch hier ist das Internet eine gute Möglichkeit sich über MIME und die MIME Types einen Überblick zu verschaffen.

RFC 2616

RFC 2616 [2] ist die Spezifikation des HTTP Protokolls. Wir sollten uns die Spezifikation mal ansehen, nach bekannten Namen unter den Autoren suchen, und im Inhaltsverzeichnis mal nach Request und Response suchen, sowie identifizieren welche HTTP Methoden und welche HTTP Status Codes es gibt. Speziell, wenn wir im letzten Kapitel uns mit Webservice s beschäftigen, wird diese RFC sehr wichtig werden.

CSS and JavaScript

Wir werden uns in diesem Buch nur ganz am Rande mit CSS und JavaScript beschäftigen. Beide sind aber relativ wichtig. Deswegen macht es durchaus Sinn sich hier ein paar Tutorials anzusehen. M.E. die besten gibt es bei W3Schools. Wenn man altmodisch ist kann man sich aber auch ein Buch kaufen, wie z.B. 'Bulletproof WebDesign' von Dan Cederholm [3] oder 'JavaScript: The Definitive Guide' von David Flanagan [4].

HTML Editors

Wenn ich HTML schreibe, dann verwende ich meist einen einfachen Texteditor. Manchmal ist es aber praktisch eine WYSIWYG Editor zu verwenden. Da gibt es ganz viele verschiedene, auch einige unter einer Open Source Lizenz wie z.B. BlueGriffon [5]. Wenn man viel mit der Erstellung von Webseiten zu tun hat, macht es durchaus Sinn sich hier einen Überblick zu verschaffen, denn mit einem guten Tool lässt sich die eigene Produktivität locker verdoppeln.

Fragen

1. Wer war einer der Erfinder von HTML?
2. Wer war einer der Erfinder von HTTP?
3. Nennen Sie die Namen von drei verschiedenen Browsern.
4. Erklären Sie den Unterschied zwischen URI, URL and URN.
5. Warum sollte man nur relative und keine absoluten Links verwenden?
6. Wofür sind Meta-Tags gut?
7. Was ist MIME und wofür wird es verwendet?
8. Schreiben Sie HTML Code, der es Ihnen erlaubt eine Wikipedia Suche in Ihre Webseiten zu integrieren. (Hinweis: wenn Sie nach "BMW" in Wikipedia suchen, so wird dies durch folgende URI bewerkstelligt: en.wikipedia.org/w/index.php?search=BMW).

9. Schreiben Sie eine HTML Seite die eine Suche in Google durchführt (die URI für eine Suche bei Google lautet: www.google.com/search?q=bmw)

10. Schreiben Sie HTML Code, der einen Benutzer nach Namen und Passwort fragt, und die eingegebenen Daten mittel POST Request and die Seite 'www.server.com/login.jsp' schickt.

11. Angenommen, Sie haben zwei Seiten, “1.html” und “2.html”. Wie müsste das Link-Tag aussehen, welches einem Link von Seite “1.html” auf Seite “2.html” setzt?

12. Skizzieren Sie wie die folgende Webseite aussieht.

```
<HTML>
   <HEAD>
      <TITLE>A dynamic JSP page</TITLE>
   </HEAD>
   <BODY>
      <FONT color="green" face="arial">
         <CENTER>Hello World!</CENTER>
         <UL>
            <LI>Welcome to the J2EE MasterClass</LI>
            <LI>Today is Monday.</LI>
         </UL>
      </FONT>
   </BODY>
</HTML>
```

Referenzen

Zu allen Themen die einem neu erscheinen, kann man erst mal in der Wikipedia nachschauen, um sich eine Überblick und ersten Eindruck zu verschaffen.

[1] W3Schools Online Web Tutorials, www.w3schools.com/

[2] RFC 2616, tools.ietf.org/html/rfc2616

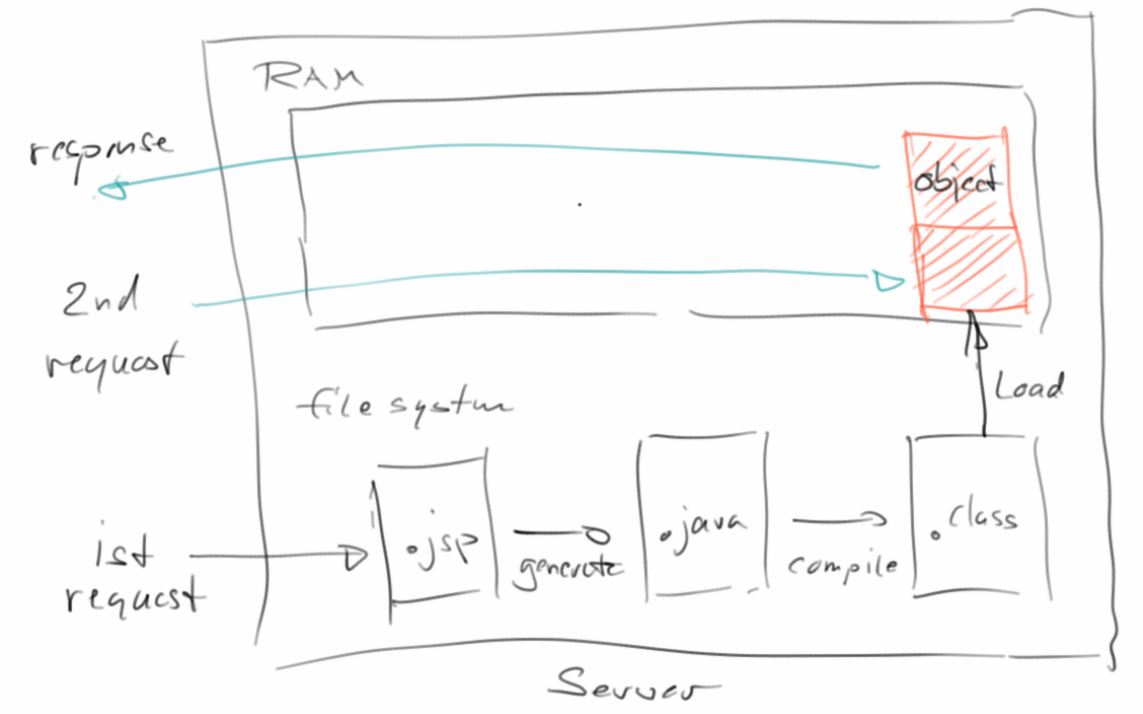
[3] Bulletproof WebDesign von Dan Cederholm

[4] JavaScript: The Definitive Guide von David Flanagan

[5] BlueGriffon, bluegriffon.org

[6] Lynx, http://lynx.browser.org/

First Steps



Ursprünglich war das Web sehr statisch: es gab Text und Hypertext Dokumente (also Textdokumente mit Links), mit und ohne Bildern. Dynamisch waren die Webseiten aber noch nicht. Das änderte sich erst mit den ersten CGI-Skripten, die noch in C und Pearl geschrieben waren, später kamen dann PHP, ASP und JSP hinzu. In letzter Zeit sind auch Python und JavaScript sehr populär geworden. JSP steht für Java Server Pages, und es ist eine Möglichkeit mit Java dynamische Webseiten zu schreiben. In diesem Kapitel geht es vor allem darum erste Schritte mit JSP zu unternehmen und das an zahlreichen Beispielen zu üben.

Dynamische Webseiten

Um dynamische Webseiten erstellen zu können gibt es zwei Möglichkeiten:

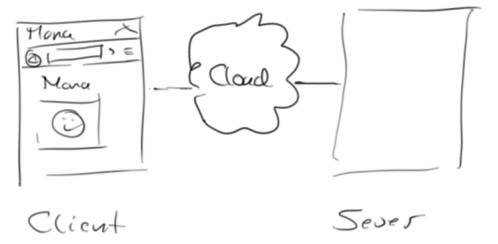
- clientseitig: z.B. mittels JavaScript
- serverseitig: z.B. mittels JSP oder PHP

Der Vorteil clientseitiger Webanwendungen ist, dass sie wenig Ressourcen auf dem Server benötigen. Die Seite wird einmal heruntergeladen und danach erfolgen fast alle Berechnungen im Browser. Bei serverseitigen Webanwendungen hingegen werden alle Berechnungen auf dem Server ausgeführt, was in der Regel dazu führt, dass man je nach Anwendung sehr viele Server benötigt.

Aus Kostengründen würde man also clientseitige Webanwendungen wohl bevorzugen. Allerdings gibt es eben sehr viele Anwendungen die sich nicht clientseitig realisieren lassen: z.B. hat Google Maps ca. 20 Petabyte an Daten. Die passen weder auf die Festplatte eines normalen Laptops, noch in ein normales Handy. Auch Websites wie Amazon oder Facebook würden nie als clientseitige Webanwendungen funktionieren. Da aber beide Arten ihre jeweiligen Stärken haben, werden diese häufig kombiniert. Deswegen wollen wir hier nicht in schwarz und weiß denken.

Client-Server

Microsoft's Word oder Adobe's Photoshop sind Beispiele für Anwendungen die als Desktop Applikationen sehr gut funktionieren. Google Maps oder Amazon hingegen sind Beispiele die nur mit einer Internetanbindung sinnvoll funktionieren. Diese Art von Anwendungen bestehen aus einem Server, der bei Google oder Amazon steht, wo die Daten sind und die Geschäftslogik ausgeführt wird, und einen Client, meist einem Webbrowser, der es dem Benutzer erlaubt auf die Daten zuzugreifen. Die grundlegende Idee hinter jeder Webanwendung ist die einer Client-Server Anwendung. Haben wir das erst einmal verinnerlicht, wird alles ganz einfach.



Der Client muss nicht notwendigerweise ein Browser sein: Google Maps auf mobilen Endgeräten ist ein schönes Beispiel für eine Client-Server Anwendung bei der der Client eine App ist.

Web Server

Wenn wir von Server reden meinen wir immer einen Webserver. Da gibt es viele verschiedene, wir werden Tomcat [2] und GlassFish [3] verwenden. Beide kommen automatisch mit der Entwicklungsumgebung NetBeans [4], werden aber auch im professionellen Umfeld verwendet. Eine kurze Einführung zu NetBeans und Webserver findet sich im Anhang.

Hello JSP

Nachdem wir jetzt die Grundlagen erläutert haben, können wir endlich anfangen unsere erste JSP Seite zu schreiben. JSP Seiten sind Text oder HTML Dateien, die aber in ".jsp" enden. Ein einfaches "Hello JSP" Beispiel soll den Anfang machen:

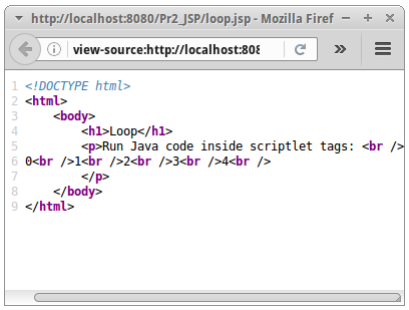

```
<%
    out.println("Hello JSP!");
%>
```

So wie es bei Java Anwendungen das "System.out.println()" gibt, so gibt es bei JSP Seiten das "out.println()". Ansonsten schreibt man innerhalb der sogenannten Scriptlet Tags "<%" und "%>" ganz normales Java.

Man kann auch HTML und Java mischen:


```
<html>
    <body>
        <h1>Loop</h1>
        <p>Run Java code inside scriptlet tags:
<br />
<%
    for (int i = 0; i < 5; i++) {
        out.print("" + i + "<br />");
    }
%>
        </p>
    </body>
</html>
```

SEP: JSP Dateinamen sollten der Camel-Case Konvention folgen, und immer mit einem Kleinbuchstaben beginnen [1].



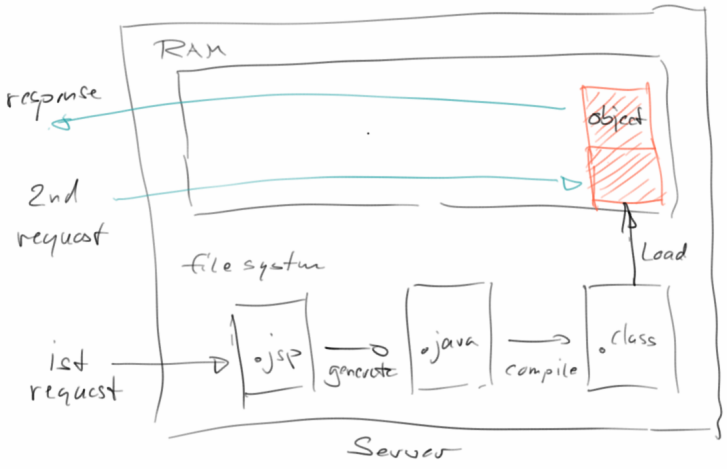
Was passiert mit dem Java?

Interessant ist allerdings, und das ist ganz wichtig, was beim Browser ankommt. Wenn wir mit der rechten Maustaste "Seitenquelltext anzeigen" (View Source) auswählen, dann sehen wir was wirklich beim Browser ankommt:

```
<html>
    <body>
        <h1>Loop</h1>
        <p>Run Java code inside scriptlet tags: <br />
0<br />1<br />2<br />3<br />4<br />
        </p>
    </body>
</html>
```

Wir sehen also, dass der Teil innerhalb der Scriptlet Tags ersetzt wird durch die Ausgabe des *out.println(...)*. Java Code erreicht den Browser nie.

Was passiert aber mit dem Java Code? Der wird im sogenannten "Servlet Engine" übersetzt und ausgeführt, also Tomcat in unserem Fall. Das passiert auf der Serverseite.



Schauen wir uns das mal ganz konkret am Beispiel von "loop.jsp" an. Das erste Mal wenn wir die Seite aufrufen, wird auf Serverseite vom Servlet Engine aus der JSP Seite die Java Servlet Klasse *loop_jsp* dynamisch generiert. Die kann man im Verzeichnis "/glassfish-4.0/glassfish/domains/domain1/generated/jsp/Pr2_JSP/org/apache/jsp/" finden:

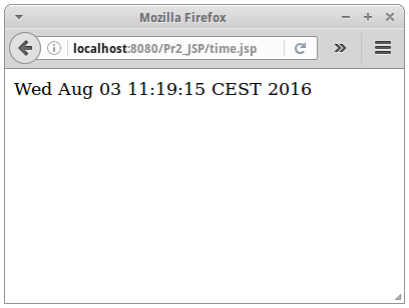
```
public final class loop_jsp extends HttpJspBase ... {
  ...

  public void _jspService( HttpServlet Request request,
                           HttpServlet Response response)
        throws java.io.IOException, Servlet Exception {
    ...
    out.write("<html>\n");
    out.write("    <body>\n");
    out.write("        <h1>Loop</h1>\n");
    out.write("  <p>Run Java code inside scriptlet tags: <br />\n");

    for (int i = 0; i < 5; i++) {
        out.print("" + i + "<br />");
    }

    out.write("\n");
    out.write("        </p>\n");
    out.write("    </body>\n");
    out.write("</html>");
    ...
  }
}
```

Und wenn wir den Java Code mit dem ursprünglichen JSP vergleichen, sehen wir was da passiert: alle HTML Elemente werden einfach mittels *out.println(...)* eingebettet, während der Java Code einfach eins-zu-eins in die Methode *_jsp Service ()* an die entsprechende Stelle kopiert wird. Diese *loop_jsp* Klasse wird dann auf dem Server kompiliert, von der JVM in den RAM geladen und anschließend auf dem Server ausgeführt. An den Browser wird dann lediglich das Resultat geschickt.

Die dynamische Java Code Generierung und das anschließende Kompilieren kostet natürlich Zeit, weswegen der erste Aufruf einer JSP Seite immer etwas länger dauert. Da dieser Schritt aber nur beim erstmaligen Aufrufen stattfinden muss, sind die nachfolgenden Aufrufe sehr viel schneller, auch im Vergleich zu anderen ähnlichen Technologien. Dieser kleine Nachteil ist leicht in Kauf zu nehmen, wenn man dies der Einfachheit der Erstellung von JSP Seiten und den damit verbunden Ersparnissen bei der Entwicklungszeit gegenüberstellt.

Ein anderer wichtiger Vorteil dieser serverseitigen Technologien, also JSP und Servlets, ist der Schutz des eigenen geistigen Eigentums (Intelectual Property). Denn im Gegensatz zu clientseitigen Technologien, wie z.B. JavaScript oder Java Applets, wird kein Sourcecode zum Browser geschickt. Der Quellcode wird am Server übersetzt und ausgeführt, der Client bekommt nur die Resultate des ausgeführten Codes zu sehen. JSP und Servlets sind auch die Ursache für den großen Erfolg Java's.

Time



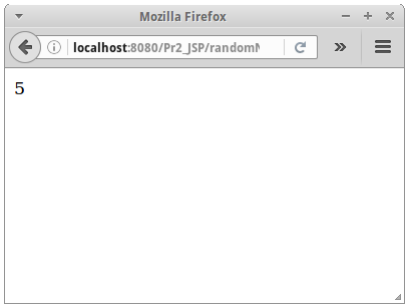
Wenn wir im Hinterkopf behalten, dass JSP Seiten immer in Java Klassen übersetzt werden, dann werden uns die nächsten Schritte viel einleuchtender erscheinen. Beginnen wir mit dem Benutzen von standard Java Klassen. Das folgende Beispiel zeigt, wie wir Java's *Date* Klasse in einer JSP Seite verwenden können:

```
<%@page import="java.util.Date"%>
<%
    out.println( new Date() );
%>
```

Das geht auch mit fast jeder anderen Java Klasse so. Der Syntax für den Import ist etwas gewöhnungsbedürftig, erinnert aber an den normaler Java Klassen.

Die Frage die sich vielleicht stellt, ist welche Klassen kann man nicht verwenden? Es sind die Klassen die mit Graphik zu tun haben, also z.B. alle AWT und Swing Klassen. Macht Sinn.

Random / Lottery

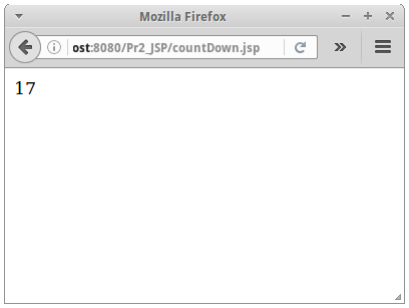


Auch eigene Methoden können wir schreiben. Dazu gibt es ein spezielles JSP Tag, das sogenannte Declaration Tag: <%! ... %>. Im Gegensatz zu normalen Scriptlet Tags, muss das Declaration Tag am Anfang einer JSP Seite stehen.

```
<%!
    private int nextInt(int max) {
        return (int) (Math.random() * max);
    }
%>
<%
    out.println(nextInt(6) + 1);
%>
```

Es gelten die normalen Java Regeln für Methoden. Im Beispiel oben sehen wir wie die Methode *nextInt()* deklariert und anschließend benutzt wird.

CountDown



Da aus einer JSP Seite ja eine Klasse generiert wird, stellt sich die Frage ob man auch Instanzvariablen und Konstanten deklarieren kann? Das CountDown Beispiel zeigt, dass das wie bei Methoden auch, im Declaration Tag gemacht werden kann:

```
<%!
    private int counter = 30;
%>
<%
    out.println(counter--);
%>
```

Damit der Countdown funktioniert, müssen wir die Seite neu laden. Es gibt mehrere Möglichkeiten eine Seite neu zu laden, bei Firefox geht das z.B. mittels:

- klicken auf den Reload Button
- des Tastenkürzel *Strg-R*
- der Funktionstaste *F5*.

Eine kleine Anmerkung am Rande: obwohl es den Anschein hat, dass alle drei "Reloads" genau das gleich tun, ist das nicht der Fall.

Zwei Browser

Eigentlich schreibt man ja Webapplikationen, damit sie von mehreren Benutzern genutzt werden können. Das führt zu sehr interessanten Effekten, und den ersten sehen wir in unserem *CountDown* Beispiel: Wenn wir auf die Seite mit einem Browser zugreifen, dann funktioniert das total super. Was passiert aber wenn wir einen zweiten Browser aufmachen, und einmal in dem einen Browser einen Reload machen, und dann in dem anderen?

Naiv würden wir hoffen, dass jeder Browser seinen eigenen Zähler hat, also bei 30 beginnt, und dann bei jedem Reload nur seinen Zähler herunter zählt. Das ist aber nicht so. Anscheinend gibt es nur einen Zähler, auf den beide Browser zugreifen. Wenn wir aber kurz nachdenken, ist das eigentlich total logisch: Der Code wird ja auf dem Server ausgeführt, und dort gibt es nur ein Objekt, auf das zufälliger Weise von zwei (oder mehr) Browsern zugegriffen wird. Die Internetprogrammierung hat viele interessante Fallstricke, und in unserem CountDown Beispiel treffen wir den ersten.

Eine kleine Anmerkung: die Aussage, dass es nur ein Objekt von jeder JSP Seite gibt ist nicht immer richtig. Normalerweise ist das zwar der Fall, allerdings kann man das in den Server Einstellungen beeinflussen: man kann dem Server z.B. sagen, dass er bei hoher Last mehr als ein Objekt für eine gegebene JSP Seite instanziieren soll.

SEP: Man sollte seine Webapplikationen immer mit mehreren Browsern gleichzeitig testen.

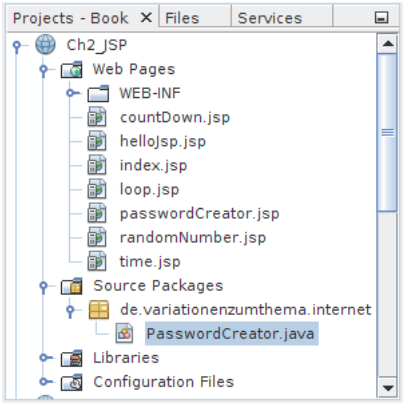
Eigene Klassen

Wir haben gesehen, dass JSP Seiten eigentlich nur Java Klassen sind, und als solche können wir alle Java Klassen verwenden, wir können Instanzvariablen und Konstanten definieren, und wir können auch Methoden definieren. Bleibt die Frage, können wir auch lokale und anonmyme Klassen definieren, oder sogar andere selbstgeschriebene Klassen verwenden. Und die Antwort ist: "jo, das können wir".

Das Beispiel *PasswordCreator* zeigt wie das geht. Ein eigene Klasse zu verwenden geht genauso wie eine der Java Klassen:

```
<%@page import="de.variationenzumthema.internet.PasswordCreator"%>
<%
    out.println(PasswordCreator.generatePassword());
%>
```

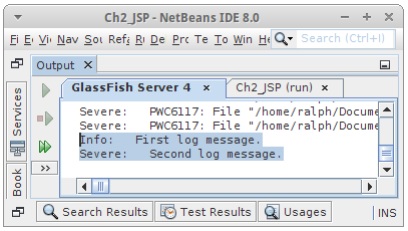
Wir müssen unsere Klasse *PasswordCreator* natürlich irgendwo definieren. In Netbeans befinden sich diese im Verzeichnis "/src/java/" unseres Projektes. Die Klasse selbst ist eine ganz normale Java Klasse:




```
package de.variationenzumthema.internet;

public class PasswordCreator {
    private static final String small =
        "abcdefghijklmnopqrstuvwxyz";
    private static final String big =
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private static final String numbers =
        "0123456789";
    private static final String symbols =
        "!#$%&'()*+,-./:;<=>?@[]^_{|}";

    public static String generatePassword() {
        String password = "";
        password += small.charAt(nextInt(small.length()));
        password += big.charAt(nextInt(big.length()));
        password += numbers.charAt(nextInt(numbers.length()));
        password += symbols.charAt(nextInt(symbols.length()));
        password += small.charAt(nextInt(small.length()));
        password += big.charAt(nextInt(big.length()));
        password += numbers.charAt(nextInt(numbers.length()));
        password += symbols.charAt(nextInt(symbols.length()));
        return password;
    }
```

```
    private static int nextInt(int max) {
        return (int)(Math.random()*max);
    }
}
```

SEP: Wenn wir eigene Klassen in JSP Seiten verwenden, sollten diese niemals im *default* Paket sein.



Logging

Als wir noch normales Java geschrieben haben, also J2SE, gab es immer die Konsole. Man konnte mit "System.out.println()" etwas auf der Konsole ausgeben. Die "Konsole" als solche gibt es auf einem Server nicht. Dafür gibt es aber Logdateien. Meist nur eine, manchmal aber auch mehrere. Wenn wir also auf einen Server etwas mit ausgeben wollen, dann landet das in der Logdatei. In GlassFish kann man alles was in die Logdatei geschrieben wird im "Output" Fenster unter GlassFish Server sehen. Man kann aber auch direkt zur Logdatei gehen, die findet man unter Services->Server->GlassFish, dann Rechtsklick und 'View Server Log' auswählen.

Jetzt ist es aber so, dass System.out.println() nicht gerade die feine englische ist wenn man auf dem Server lebt. Da gibt es nämlich die Klasse *Logger*, die für alles was mit Logdateiein zu tun hat viel besser geeignet ist. Verwendet wird die Klasse wie folgt:

```
<%@page import="java.util.logging.*"%>
<html>
    <body>
        <h1>Logging</h1>
<%
    // one way to log to the servers log files is via
    System.out.println("First log message.");

    // another more professional way is via
    Logger logger = Logger.getLogger(this.getClass().getName());
    logger .severe("Second log message.");
%>
    </body>
</html>
```

Der Vorteil von Logger ist, dass er viel spezifischer ist, was die Log-Message angeht. So gibt es verschiedene Levels für Log-Messages:

- **info:** einfach nur so mal sehen ob der Code ausgeführt wird
- **warning:** etwas ist ungewöhnlich, aber wahrscheinlich kein Problem
- **severe:** etwas ist wirklich nicht in Ordnung.

Es gibt noch ein paar andere Log-Levels, aber die drei sind die wichtigsten. Der Vorteil des Loggers ist nun, dass man ihn unterschiedlich konfigurieren kann: auf der Maschine eines Entwicklers würde man alle Log-Messages sehen wollen, aber auf der Production-Machine, möchte man eigentlich nur diejenigen geloggt haben, die wirklich schwerwiegende Fehler darstellen. Denn zu viel loggen kann die Performance eines Servers negativ beeinflussen. Und es ist schon häufiger passiert, dass ein Server abgestürzt ist, weil er keinen Plattenplatz mehr frei hatte. Der Grund dafür: richtig, Riesen-Log-Files.

Review

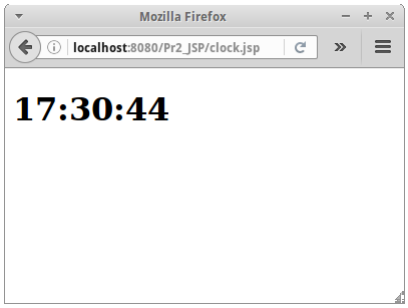
Wir haben unsere ersten Schritte in Richtung dynamischer Webprogrammierung unternommen. Dabei haben wir

- von Clients und Servern gehört.
- das Scriplet Tag und das Declaration Tag kennen gelernt,
- gesehen wie wir Instanzvariablen, Methoden und Klassen in JSP Seiten verwenden können
- und wie das mit dem Logging funktioniert.

Projekte

Die Projekte die folgen zeigen einige einfache Anwendungen die mit JSP erstellt werden können und vermitteln aber auch noch einige andere hilfreiche Konzepte in diesem Zusammenhang.

Clock



Das Beispiel *Time* das wir oben gesehen haben war sehr statisch, d.h. die Uhrzeit hat sich nicht verändert. Das ist generell ein Problem mit Webanwendungen. Mithilfe des Meta Refresh Tags können wir etwas Dynamik in unsere Webanwendungen bringen. Wir veranlassen den Browser einfach einmal pro Sekunde die Seite neu zu laden. Wenn wir dann noch die Klasse SimpleDateFormat zum Formatieren der Uhrzeit verwenden, kann sich das Resultat schon sehen lassen.

```
<html>
    <head>
        <meta http-equiv="refresh" content="1" />
    </head>
    <body><h1>
        <%
            SimpleDateFormat formater =
                    new SimpleDateFormat("HH:mm:ss");
            Date now = new Date();
            out.println(formater.format(now));
        %>
    </h1></body>
</html>
```

Ein kleiner Nachteil dieser Anwendung ist, dass sie eine relativ hohe Last auf unserem Server erzeugt. Wenn nämlich viele Leute unsere Seite aufrufen, und jeder die Seite einmal in der Sekunde läd, dann kommt da einiges an Datenvolumen und Last auf unseren Server zu. Eine Anwendung wie Clock würde wahrscheinlich besser mittels JavaScript implementiert.

Visitor*

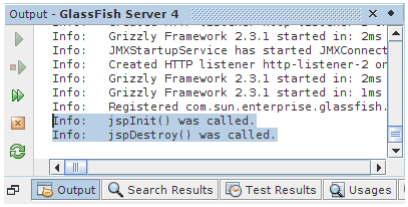
Basierend auf dem CountDown Beispiel können wir auch einen Visitor Counter implementieren.

```
<%!
    private int visitorCounter = 0;
%>
<html>
    <body>
        <h2>Welcome, Visitor Nr. <%= ++visitorCounter %></h2>
    </body>
</html>
```

Wenn wir die Seite von verschiedenen BrowserTabs oder anderen Browsern besuchen, werden wir sehen, dass jedesmal wenn die Seite von irgendjemandem geladen wird, der Zähler um eins hoch gezählt wird.

Obwohl es so aussieht, wie wenn die Anwendung tadellos funktionieren würde, ist etwas Vorsicht geboten. Bei geringer Serverlast und wenig Nutzern funktioniert die Anwendung auch einwandfrei. Überlegen wir kurz was schiefgehen kann: Wir haben ja gelernt, dass aus einer JSP Seite eine Java Klasse generiert wird. Diese wird dann kompiliert, und der Webserver, genauer der Servletcontainer, instanziiert dann *ein* Objekt von dieser Klasse. Das ist wenigstens was wir hoffen. Meistens ist das auch so. Allerdings unter sehr hoher Last, oder wenn der Servletcontainer entsprechend konfiguriert ist, können es auch mal zwei oder noch mehr Objekte sein. Würde dann unser Visitor Counter noch funktionieren? Was auch passieren kann, wenn andere Seiten unter hoher Last sind, dass unser Objekt vom Garbage Collector eingesammelt wird, weil es gerade nicht gebraucht wird. Dann würde unser Zähler beim nächsten Aufruf wieder bei Null anfangen.

Init and Destroy*

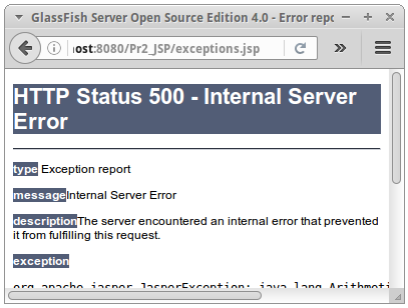


Manchmal möchten wir, dass etwas beim ersten Ausführen einer JSP Seite passiert, z.B. irgendeine Initialisierung. Oder wir möchten kurz bevor der Server runtergefahren wird noch etwas in einer Datei oder Datenbank sichern. Das können wir mit den zwei vordefinierten Methoden *jspInit()* und *jspDestroy()* lösen. Beide müssen im Deklarationsteil des JSP stehen, sind ja Methoden:

```
<%!
    public void jspInit() {
        System.out.println("jspInit() was called.");
    }

    public void jspDestroy() {
        System.out.println("jspDestroy() was called.");
    }
%>
```

Die Methode *jspInit()* wird nur einmal aufgerufen, und zwar wenn unsere Seite das erst Mal geladen wird. Umgekehrt wird die Methode *jspDestroy()* aufgerufen wenn die Anwendung "undeployed" wird, oder der Server runtergefahren wird.

Exceptions



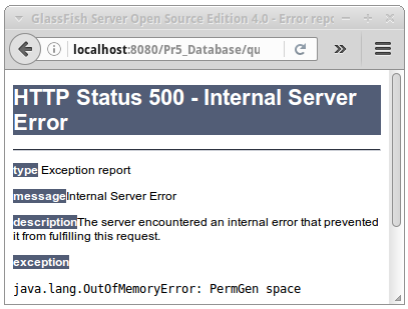
Wie mit jedem Java Code kann es auch bei JSP Seiten zu Exceptions kommen. Und wie wir das bereits im ersten Semester gelernt haben, sollten wir Exceptions immer behandeln. Bei Webanwendungen kommt noch hinzu, dass unbehandelte Exceptions u.U. dazu führen können, dass Hacker oder andere Bösewichte an kritische Informationen gelangen können. Deswegen sollten wir extra vorsichtig sein, und wenn möglich es zu keinen unbehandelten Exceptions kommen lassen.

Wenn eine Exception auftritt, macht es Sinn diese zu loggen:

```
<%@page import="java.util.logging.*"%>
<html>
    <body>
        <h1>Exceptions</h1>
<%
    try {
        int x = 5 / 0;

    } catch ( Exception ex) {
        Logger logger = Logger.getLogger(this.getClass().getName());
        logger.severe("Exception occured: "+ex);
    }
    out.println("Exception was caught, life is good.");
%>
    </body>
</html>
```

Und natürlich sollte man regelmässig seine Logfiles durchgehen um nach Unerwartetem zu suchen.

ErrorPermGenSpace



Weil wir gerade bei Fehlern sind: an manchen Fehlern sind wir Schuld und an manchen Fehlern sind die bösen Nutzer Schuld. Dafür gibt es Exception Handling. Aber es gibt auch Fehler für die weder die Entwickler noch die Nutzer etwas können. Im Prinzip gibt es da drei Hauptverdächtige:

- der Webserver / Servlet Engine: z.B. der Fehler rechts ist so einer, scheint ein Memory Leak zu sein,
- die Datenbank: manchmal werden Datenbank Verbindungen nicht geschlossen, oder es gibt offene Locks, etc.,
- das Betriebssystem: z.B. Festplatte ist voll.

Manchmal ist es auch die Hardware, aber das ist sehr selten.

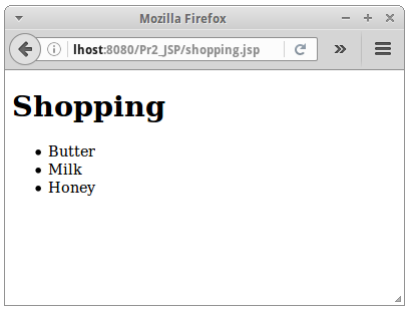
index.jsp

Für jede unserer zwölf Netbeans Projekte benötigen wir eine "index.jsp" Seite. Die unterscheiden sich lediglich im Inhalt, aber der Aufbau dieser Seiten ist eigentlich identisch. Wie üblich, wollen wir eigentlich so wenig Code wie möglich ändern müssen, wir schreiben REUSE ganz groß.

Eine Möglichkeit das zu tun, ist den Inhalt der Seite über ein Array treiben zu lassen:

```
...
String[] jspPages = {
    "clock", "visitor", "exceptions",
    "guestBook", "editor", "file", "pizzaMenu"
};
...
```

Damit sollte es möglich sein die Seite rechts dynamisch zu generieren. Lediglich die Überschrift müssen wir noch anpassen.



Shopping

Der Klassiker unter den Internetanwendungen ist die Shopping Anwendung. Hier in unserem ersten Versuch, wollen wir nur den Inhalt des Warenkorbs (einfach ein Vector) ausgeben. Dazu definieren wir erst mal den Warenkorb:

```
<%
    Vector<String> cart = new Vector<String>();
    cart.add("Butter");
    cart.add("Milk");
    cart.add("Honey");
    ...
%>
```

und daraus generieren wir dann das HTML rechts.

Vector vs. List

Es mag etwas ungewöhnlich erscheinen warum wir im letzten Beispiel die Klasse *Vector* verwendet haben und nicht die Klasse *ArrayList*, die wir bisher immer verwendet haben. Der Grund liegt darin, dass Webanwendungen nicht notwendigerweise *threadsafe* sind. Wir werden später mehr dazu hören, aber im Moment wollen wir lediglich anmerken, dass die Klassen Vector und Hashtable threadsafe sind, die Klassen ArrayList, LinkedList, HashMap und TreeMap dagegen nicht. Aber wie gesagt später mehr dazu.

Editor

Im nächsten Kapitel möchten wir einen kleinen Texteditor schreiben. Dazu brauchen wir einmal eine HTML Seite mit einem Formular und einer TextArea sowie einem Submit Knopf. Der Inhalt des Formulars soll an die Seite *editorLogic.jsp* geschickt werden, die wir im nächsten Kapitel schreiben werden. Was wir hier allerdings schon machen wollen, ist den Inhalt einer Datei innerhalb der TextArea anzuzeigen:

```
<textarea name="text" rows="4" cols="40">
<%
    try {
        BufferedReader br =
            new BufferedReader(new FileReader("editor.txt"));
        StringBuffer sb = new StringBuffer();
        while (true) {
            String line = br.readLine();
            if (line == null) {
                break;
            }
            sb.append(line);
        }
        out.println(sb);
    } catch (Exception e) {
        out.println("Type something here and save it!");
    }
%></textarea>
```

Natürlich muss die Datei *editor.txt* existieren, sonst kommt es zu einer Exception. Wenn wir die Seite aber das erste Mal ausführen, kann es es diese Datei noch gar nicht geben. Deswegen fangen wir die Exception ab und geben dort einen kleinen Hinweis für den Nutzer.

Files

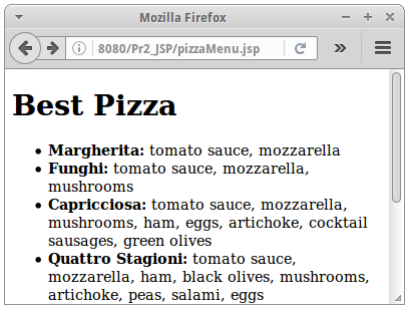
Ganz ungefährlich ist das Arbeiten mit Dateien nicht. Wenn wir z.B. irgendwo dem Nutzer erlauben einen Dateinamen einzugeben, dann könnte ein Schlaumeier auf die Idee kommen "../../../../../../" als Dateiname einzugeben:

```
<%
    try {
        File f =
            new File("../../../../../../../");
        //out.println(f.getCanonicalPath() + "<br/>");
        String[] files = f.list();
        Arrays.sort(files);
        for (String fl : files) {
            out.println(fl + "<br/>");
        }

    } catch (Exception e) {
        out.println(e);
    }
%>
```

Die Problematik sollte offensichtlich sein. Falls nicht, dann freuen sich die Hacker auf unsere Webanwendungen...

SI: Wir solllten niemals dynamische Dateinamen zulassen.

PizzaMenu



Das eine oder andere Restaurant hat inzwischen sein Menü digitalisiert. Manche habe es im Internet, manche aber auch auf einem Monitor im Restaurant. In diesem ersten Teil wollen wir aus dem folgenden String,

```
<%
    String menu = "Best Pizza\n"
       + "Margherita: tomato sauce, mozzarella\n"
       + "Funghi: tomato sauce, mozzarella,
mushrooms\n"
       + "Capricciosa: tomato sauce, mozzarella, mushrooms, ham,"
       + "eggs, artichoke, cocktail sausages, green olives\n"
       + "Quattro Stagioni: tomato sauce, mozzarella, ham, black "
       + "olives, mushrooms, artichoke, peas, salami, eggs\n"
       + "Vegetariana: tomato sauce, mozzarella, mushrooms, onion, "
       + "(artichoke), sweet corn, green peppers\n"
       + "Quattro Formaggi: tomato sauce, and 4 assorted cheeses, "
       + "generally mozzarella, parmesan cheese, blue cheese, and goat
cheese\n";
%>
```

eine hübsche HTML Seite wie rechts generieren. Dabei sind evtl. die folgenden beiden Zeilen hilfreich:

```
BufferedReader br = new BufferedReader(new StringReader(menu));
...
String[] pizz = line.split(":");
...
```

Wir gehen also davon aus, dass die erste Zeile die der Nutzer eingibt der Name des Restaurants ist, in unserem Beispiel "Best Pizza". Die nächsten Zeilen listen die verschiedenen Pizzen, jeweils mit dem Namen der Pizza beginnend, einem Doppelpunkt, gefolgt von den Zutaten. Im nächsten Kapitel sehen wir dann wie wir damit eine Webanwendung machen können mit der man dann das Menü öfter mal ändern kann.

Vielleicht noch erwähnenswert: All die Seiten die wir jetzt schon geschrieben haben, können wir nicht nur auf unserem Computer ansehen, sondern auch auf anderen Computern. Wir müssen dazu lediglich die IP Adresse des Rechners wissen auf dem unser Server läuft (bei mir ist das gerade die "192.168.178.43"), und tippen dann folgendes in die Adresszeile des Browsers auf dem anderen Computer:

```
http://192.168.178.43:8080/Pr2_JSP/pizzaMenu.jsp
```

Ziemlich cool, oder? Wenn wir eine externe IP Adresse haben (eher unwahrscheinlich) dann könnte sogar die ganze Welt jetzt schon unsere Webseiten ansehen!

Research

Auch in diesem Kapitel gibt es einige Themen die man noch durch Eigenrecherche vertiefen kann.

Servlet Engines und Webserver

Wir wollen im Internet nach den Begriffen "Servlet Engines", manchmal auch "Servlet Container" genannt und Webservern suchen. Dabei sollten wir mindestens fünf oder sechs verschiedene finden. Uns sollte klar werden was der Unterschied zwischen Servlet Engines und Webservern ist, und welcher wohl für unsere Zwecke am besten geeignet ist. Wir sollten uns aber auch Kriterien überlegen, nach denen wir einen Servlet Engines und einen Webservern für einen Produktionsserver unserer zukünftigen Firma auswählen würden.

JSP, PHP und ASP

Neben JSP gibt es auch noch andere Webtechnologien, wie z.B. PHP, ASP, aber evtl. auch JavaScript oder Python. Wir sollten uns auch die Konkurrenz mal ansehen und darüber etwas in Erfahrung bringen. Schließlich sollten wir auch eine kleine Tabelle machen, in der wir diese nach verschiedenen Kriterien gegenüber stellen, wie z.B.

- Performance
- Popularität
- Einfachheit, auch zu Lernen
- Sicherheit
- Unterstützung eines Testing-Frameworks wie JUnit
- Unterstützung von Reflection und Generics

CGI

Das Common Gateway Interface (CGI) ist ein uralt Standard für den Datenaustausch zwischen Servern und Clients, und Basis von dynamischen Webanwendungen. Es macht Sinn sich in die Thematik einmal einzulesen, vor allem wenn man ein bisschen ein Interesse für die Geschichte des Internets hat.

Fragen

1. Wir wollen das Datenvolumen abschätzen, das eine Anwendung wie Clock auf unserem Server verursacht. Gehen Sie davon aus, dass die Seite 1 kByte groß ist, dass die Seite sich einmal pro Sekunde neu lädt, und dass wir im Durchschnitt 1000 Nutzer haben, die diese Seite im Browser offen haben. Schätzen Sie das Datenvolumen im Monat ab, denn dafür bezahlen Sie Ihrem Serviceprovider.

2. In JSP gibt es Deklarationen (<%!...%>) und Code-Fragmente (<%...%>). Wann und wofür würden Sie welches verwenden?

3. Erklären Sie wann die beiden Methoden jspInit() und jspDestroy() aufgerufen werden, und wofür man sie verwenden könnte.

4. Schreiben Sie eine JSP Seite, lotto.jsp, die sechs Zufallszahlen zwischen 1 und 49 erzeugt und diese sortiert ausgibt. Ein Bonus wäre, wenn es keine Duplikate gibt. Hinweis, die folgenden Zeile könnte hilfreich sein:
   <%@page import="java.util.TreeSet"%>

5. Schreiben Sie eine JSP Anwendung, die aus drei JSP Seiten besteht:
   - login.jsp: Auf dieser Seite kann der Benutzer seinen Namen und sein Passwort eingeben. Die Aktion des Form Tags sollte auf die Seite 'check.jsp' weiterleiten.
   - check.jsp: Diese Seite sollte den eingegebenen Namen und das Passwort auf Richtigkeit überprüfen (dabei dürfen Sie in Ihrem Code feste Werte für den korrekten Namen und das korrekte Passwort verwenden). Sind die Eingaben des Benutzers richtig, so sollte der Benutzer auf die Seite 'welcome.jsp' weitergeleitet werden, andernfalls sollte er auf die 'login.jsp' Seite zurückgeschickt werden.
   - welcome.jsp: Diese Seite sollte einen Text anzeigen, dass sich der Benutzer erfolgreich angemeldet hat. Beachten Sie bitte, dass nach einer erfolgreichen Anmeldung im session Objekt irgendwie kenntlich gemacht werden sollte, dass der Benutzer sich erfolgreich angemeldet hat.

6. Wann wird eine JSP Seite kompiliert?
   O Beim Hochladen auf den Server
   O Jedes Mal wenn ein User darauf zugreift
   O Das erste Mal wenn ein User darauf zugreift

Referenzen

Hier gibt's nur eine Referenz. Aber natürlich kann man in der Wikipedia nachschauen, wenn man mehr wissen will. Immer ein guter Anfang. Man sollte da eben nur nicht aufhören.

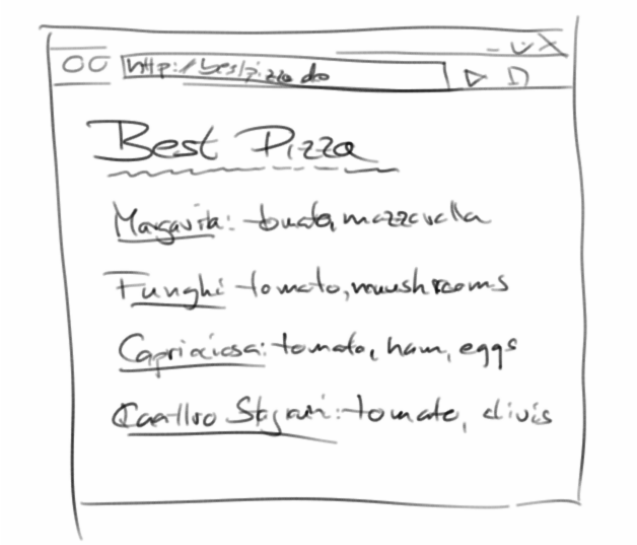
[1] Guidelines, Patterns, and code for end-to-end Java applications, http://www.oracle.com/technetwork/java/namingconventions-139351.html

[2] Tomcat, http://tomcat.apache.org/

[3] GlassFish, https://glassfish.java.net/

[4] NetBeans, https://netbeans.org/

Request and Response



Nach den bisherigen Einführungen, beginnnen wir in diesem Kapitel die wichtigsten Grundsteine zu legen. Wir beschäftigen uns mit dem Request und Response Objekten. Damit können wir auf User Input reagieren. Und wenn wir uns ans erste Semester erinnern, dann heißt das: jetzt fängt es an cool zu werden!

Request

Wenn der Browser einen HTTP Request an den Server sendet, dann übermittelt er dabei einiges an Informationen. In JSP werden diese Informationen in dem JSP-Objekt *request* gebündelt und können dann sehr einfach verwendet werden, wie wir gleich sehen werden.

Reverse Text

Im ersten Kapitel haben wir einfache HTML Formulare erstellt. Allerdings, was wir noch nicht konnten war irgend etwas mit den eingegebenen Daten anzufangen. Dazu kommen wir jetzt. Nehmen wir an wir haben folgendes Formular:

```
<html>
    <body>
        <h1>Reverse Text</h1>
        <p>Enter text to reverse:</p>
        <form action="reverseTextLogic.jsp" method="GET" >
            <input type="text" name="textToReverse" />
            <input type="submit" value="Reverse" />
        </form>
    </body>
</html>
```

Das Action-Attribut des Form Tags besagt wohin die vom Benutzer eingegebenen Daten geschickt werden sollen. In diesem Beispiel an die Seite *reverseTextLogic.jsp*:

```
<%!
    private String reverseText(String s) {
        String ret = "";
        for (int i = s.length()-1; i >= 0; i--) {
            ret += s.charAt(i);
        }
        return ret;
    }
%>
<%
    String text = request.getParameter("textToReverse");
    out.println("The reverse text is: " + reverseText(text));
%>
```

Das ist eine ganz normale JSP Seite. Neu ist hier die Verwendung des *request* Objekts. Wir bitten das *request* Objekt uns den Parameter namens "*textToReverse*" zu geben. Dieser war vorher als HTML Input Tag definiert worden. Dann rufen wir die Methode reverseText() auf, und senden das Resultat zurück an den Browser mittels out.println().

Wie werden die Daten geschickt?

Nun stellt sich die Frage, wie werden denn die Daten vom Browser zum Server geschickt? Wenn wir uns im letzten Beispiel die Adresszeile im Browser genau ansehen, sehen wir, dass dort folgendes steht:

```
.../reverseTextLogic.jsp?textToReverse=racecar
```

D.h., am Ende des Dateinamens *reverseTextLogic.jsp* wurde einfach ein Fragezeichen, dann der Name des Input-Tags gefolgt von einem Gleichheitszeichen, und schließlich dem Wert den der Benutzer eingegeben hat. Also die Daten werden einfach über die Adresszeile, die URI geschickt!

Man nennt das Ganze einen Uniform Resource Identifier, kurz URI (ähnlich zu URL) [1], und die Details dazu sind in der RFC 3986 spezifiziert [4]. Dort findet man auch die allgemeine Form einer URI:

```
scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]
```

Alles was in eckigen Klammern ist, ist optional. Ein Beispiel wäre also:

```
http://localhost:8080/Ch3_JSP2/reverseTextLogic.jsp?textToReverse=otto
```

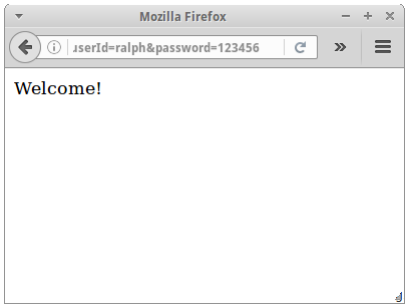
Hat man mehrere Input-Tags, dann werden die einzelnen Key-Value Paare, auch Parameter genannt, noch durch ein Und-Zeichen (Kaufmanns-Und: &) getrennt.

Login

Betrachten wir als zweites Beispiel ein Login Formular, im Prinzip identisch mit dem aus Kapitel eins:

```
<html>
   <body>
      <h1>Login</h1>
      <form action="loginLogic.jsp" method="GET">
          UserID:  <input type="text" name="userId"/><br/>
          Password:<input type="password" name="password"/><br/>
          <input type="submit" value="Login"/>
      </form>
   </body>
</html>
```

Die Daten werden an *login Logic.jsp* Seite geschickt, wo diese dann auf Korrektheit geprüft werden:




```
<%
    String id = request.getParameter("userId");
    String passwd =
        request.getParameter("password");
    if ((id != null) && (passwd != null)) {
        if ((id.equals("ralph")) &&
            (passwd.equals("123456"))) {
            out.println("Welcome!");
        } else {
            out.println("Please, try again!");
        }
    }
%>
```

Interessant ist auch hier wieder die Adresszeile,

```
.../loginLogic.jsp?userId=ralph&passowrd=123456
```

die beinhaltet nämlich Benutzername und Passwort im KLARTEXT! Und beide werden sogar im Browserverlauf gespeichert! Offensichtlich keine so gute Sache.

GET vs POST

Kann man da was machen? Die Lösung ist überraschend einfach: man muss lediglich in der Zeile

```
<form action="loginLogic.jsp" method="GET">
```

das Wort *GET* durch das Wort *POST* ersetzen:

```
<form action="loginLogic.jsp" method="POST">
```

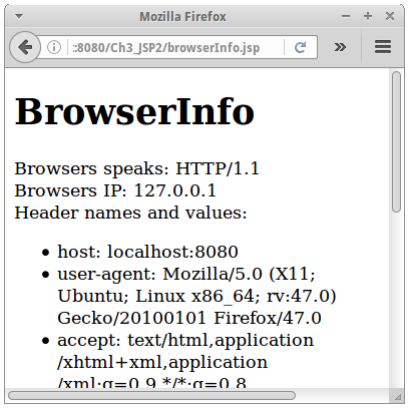
Im ersten Fall spricht man von einem HTTP *GET* Request, im zweiten von einem HTTP *POST* Request. Vergleichen wir die beiden kurz:

- Ein *GET* Request kann durch einen Link oder ein Formular ausgelöst werden, ein *POST* Request nur durch ein Formular.
- Die Daten (Parameter) werden bei einem *GET* Request über die Adresszeile gesendet, sind also im Klartext zu lesen, und werden im Browserverlauf gespeichert. Bei einem *POST* Request werden die Parameter als Teil des HTTP Headers geschickt, sind also nicht so einfach zu sehen, und werden nicht im Browserverlauf gespeichert. Besonders sicher sind die Daten aber auch beim *POST* Request nicht.
- Die Menge der Daten ist bei einem *GET* Request auf ein bis zwei Kilobyte begrenzt, bei einem *POST* Request kann sie aber mehrere Gigabyte betragen.

Am besten wir prüfen das selbst mal nach, idealerweise sollten wir aber vorher den Browserverlauf löschen, oder einen anderen Browser verwenden.

SEP: Wenn möglich sollte man immer den *POST* Request bevorzugen.

BrowserInfo



Über das *request* Objekt erhalten wir Zugriff auf die Parameter, aber das *request* Objekt enthält zusätzliche Informationen über den Browser der gerade auf unseren Server zugreift. Dazu gehört das Protokoll das der Browser versteht, seine IP Adresse und sein Port, aber auch andere interessante Informationen:

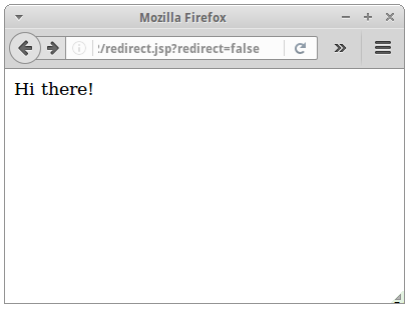
- user-agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:47.0) Gecko/20100101 Firefox/47.0
- accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
- accept-language: en-US,en;de
- accept-encoding: gzip, deflate
- referer: http://localhost:8080/Ch3_JSP2/
- cookie: cook=ie; JSESSIONID=c819f064abc3432b265c0bbf3a20
- connection: keep-alive

Wir können u.a. feststellen, dass der Nutzer den Mozilla Browser verwendet und dass auf seinem Rechner ein 64-bit Ubuntu läuft. Außerdem scheint der Nutzer amerikanisches Englisch und Deutsch zu sprechen. Wir sehen auch, von woher (Referer) der Nutzer auf unsere Seite gekommen ist. Und wir können vermuten, dass er gerne Weihnachts-Plätzchen (Cookies) mag.

Response

Kommen wir zum zweiten großen Thema dieses Kapitels, dem *response* Objekt. Es entspricht der HTTP Response, ist also die Antwort des Servers an den Browser. Der Server sendet alle seine Daten als HTTP Response an den Browser. Auch hier werden die Informationen mit einem JSP Objekt, dem *response* Objekt, gesammelt bevor sie abgeschickt werden.

Redirect



Aus JSP Sicht ist das *response* Objekt einfach wieder ein vordefiniertes Objekt. Sehr häufig wird das *response* Objekt für Redirects verwendet:

```
<%
    String redirect =
        request.getParameter("redirect");
    if ( (redirect != null) &&
        (redirect.equalsIgnoreCase("true")) ) {
        response .sendRedirect("index.jsp");
        return;  // this is very important!
    } else {
        out.println("Hi there!");
    }
%>
```

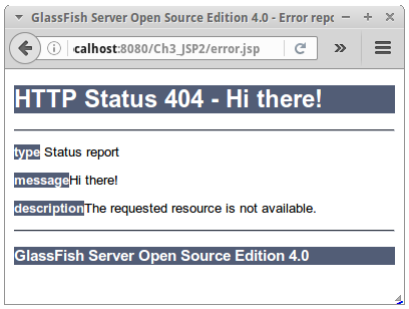
Wir überprüfen also ob der Parameter redirect auf true gesetzt wurde, und falls ja schicken wir den Nutzer einfach zur *index.jsp* Seite. Andernfalls senden wir ihm ein "Hi there!". So etwas ist ganz nützlich für Logins z.B..

Weder vor noch nach dem Redirekt dürfen wir irgendwelche Daten an den Browser schicken, also z.B. out.println() oder ähnliches. Denn der Redirekt funktioniert nicht serverseitig sondern clientseitig. Was ein Redirekt tut, er schickt dem Browser einen 302er Status Code im HTTP Header:

```
response.setStatus(response.SC_MOVED_TEMPORARILY);  // HTTP status
code 302
response.setHeader("Location", "index.jsp");
```

Das veranlasst den Browser dann dazu die neue Seite anstelle der alten zu laden. Der Server sagt also dem Browser, dass er noch mal einen HTTP Request schicken soll, dieses mal aber an die neue Seite. Wenn wir nicht diesen Umweg über den Browser gehen wollen, gibt es den RequestDispatcher. Den verwendet man allerdings eher selten, wir werden ihn im Datenbank Kapitel kennen lernen.

Error



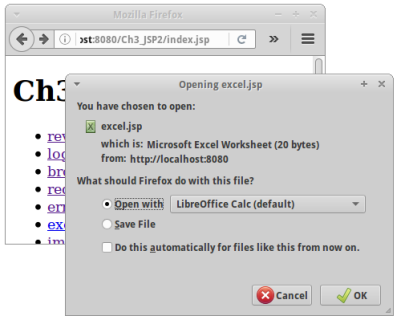
Manchmal geht etwas schief. Dann können wir dem Browser einen Fehlercode schicken. Der bekannteste ist der '404' oder auch "file not found":

```
<%
    response.sendError(response.SC_NOT_FOUND,
        "Hi there!");
%>
```

Wir tun das auch über das *response* Objekt. Effektiv wird dadurch im HTTP Header ein 404 Status Code gesendet. Es gibt noch ganz viele andere Fehler, Details findet man in der RFC 2616 zum HTTP Protokoll.

Es sei noch angemerkt, dass nicht alle Browser die genaue Fehlermeldung weitergeben, der Internet Explorer hatte die Angewohnheit, einfach zu sagen, dass ein Fehler aufgetreten ist, welcher Fehler genau, hat er allerdings verschwiegen.

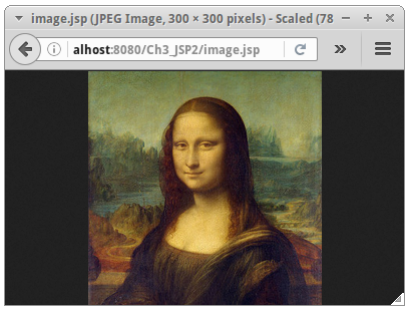
Excel



Eine anderer interessante Anwendung für das *response* Objekt, ist der Content Type. Darüber können wir dem Browser mitteilen, was für Daten im Anhang stecken. Normalerweise ist das "text/html". Wir können das aber ändern:

```
<%
    response.setContentType("application/vnd.ms-excel");
    for (int i = 0; i < 10; i++) {
        out.print("" + i + ",");
    }
%>
```

In dem Beispiel erklären wir dem Browser was jetzt kommt ist ein Excel Spreadsheet. Der Browser fragt den Benutzer dann was dieser mit den Daten zu tun wünscht. Eine Option wäre die Daten direkt mit OpenOffice Calc oder Microsoft Excel zu öffnen.



Image

Genauso können wir auch Bild Dateien im "Rohformat" an den Browser schicken. Wir sagen dem Browser, was jetzt kommt is ein "image/jpeg":

```
<%@page import="java.io.*" %>
<%@page import="java.net.*" %>
<%
    response.setContentType("image/jpeg");
    OutputStream os = response.getOutputStream();
    String filePath = getServletContext().getRealPath("/");
    String imagePath = filePath + "Mona_Lisa.jpg";
    InputStream is = new FileInputStream(imagePath);
    byte[] buffer = new byte[32 * 1024]; // 32k buffer
    int bytesRead = 0;
    while ((bytesRead = is.read(buffer)) != -1) {
        os.write(buffer, 0, bytesRead);
    }
    os.flush();
    os.close();
%>
```

Der Browser erwartet dann ein Bild im Rohformat. Wir sollten uns über den Unterschied im Klaren sein: es handelt sich hier nicht um HTML oder einen Link. Im Allgemeinen, ist dieser Ansatz nicht zu empfehlen, da er relativ langsam ist. Manchmal macht er aber durchaus Sinn: z.B. wenn die Bilder dynamisch generiert werden, oder wenn die Bilder geschützt werden sollen, also nicht für jeden zugänglich sein sollen.

Review

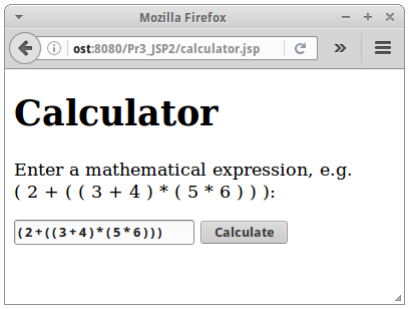
In diesem Kapitel haben wir das Request und das Response Objekt kennen gelernt. Sie sind Abbildungen der unterliegenden HTTP Requests und Responses. Mit ihnen können wir z.B.

- auf Formular Daten zugreifen,
- Informationen über den Browser erfahren,
- den Nutzer woanders hinschicken und
- den Content-Type ändern.

Wir werden gleich noch viel mehr praktische Beispiele sehen wofür die beiden gut sein können.

Projekte

Nachdem wir gesehen haben wie die Request und Response Objekte funktionieren, wollen wir uns jetzt ein paar interessante Beispiele ansehen.



Calculator

Im zweiten Semester (Algorithmen) haben wir gelernt wie man mittels der Stack Klasse mathematische Ausdrücke berechnen kann. Den Code von damals können wir ganz einfach in eine Webanwendung überführen.

Die Anwendung soll ähnlich wie bei unserer Login Applikation aus zwei Teilen bestehen: dem ersten, *calculator.jsp*, in dem der Nutzer den mathematischen Ausdruck eingibt, also einfach ein Formular mit einem Textfeld. Im zweiten Teil, *calculatorLogic.jsp*, deklarieren wir dann die beiden Methoden *convertFromInfixToPostfix()* und *evaluate()* im Deklarationsteil: <%! ... %> (Die beiden Methoden sollten noch irgendwo in dem Ordner vom zweiten Semester rumliegen). Der Rest ist dann trivial:

```
<%
    String infix = request.getParameter("mathExpression");
    String postfix = convertFromInfixToPostfix(infix);
    int result = evaluate(postfix);
    out.println("The result is: " + result);
%>
```

Captcha

Captcha's werden benutzt um Bots von Menschen zu unterscheiden. Man findet diese häufig wenn man sich neu irgendwo anmeldet. Das ist deswegen notwendig, weil es überraschend viele böse Buben im Internet gibt, die versuchen mittels Bots (in der Regel irgendwelche Skripten) sich so viele Accounts zu ergaunern wie möglich, um damit dann wieder irgendwelchen Unsinn zu treiben (Spam verschicken z.B.).

Unsere Catpcha's sind etwas trivial, und verdienen eigentlich den Namen gar nicht, aber die echten funtionieren nach dem gleichen Prinzip. Die Anwendung besteht wieder aus zwei Teilen, *captcha.jsp* und *captchaLogic.jsp*. Im ersten Teil stellen wir dem Kandidaten (Roboter oder Mensch) eine Frage. Die muss natürlich zufällig sein, und sich jedesmal ändern, sonst könnte der Roboter ja die Antwort erraten.

```
<%
    int a = (int) (Math.random() * 9) + 1;
    int b = (int) (Math.random() * 9) + 1;
%>
<html><body>
        <h1>Captcha</h1>
        <p>Please verify that you are a not a robot: <br />
            What is the sum of <%= a%> + <%= b%>?</p>
        <form action="captchaLogic.jsp" method="GET">
            <input type="number" name="sum"/>
            <input type=" hidden " name="result" value="<%= a + b%>"/>
            <input type="submit" value="Verify"/>
        </form>
    </body>
</html>
```

Im zweiten Teil müssen wir dann lediglich feststellen, ob der Kandidat richtig gerechnet hat. Dazu müssen wir allerdings wissen, was die richtige Lösung ist. Dafür können wir das "hidden" Tag verwenden.

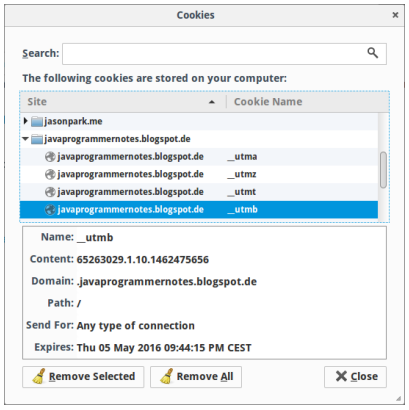
Besonders gut ist unsere Captcha App nicht gerade, denn wenn wir via "View Sourcecode" uns das generierte HTML ansehen, dann steht da die Antwort, und ein paar schlaue Script-Kiddies haben unser Captcha dann auch sofort geknackt.

Cookies

Hier handelt es sich nicht um Chocolate Chip Cookies, oder um Vanillekipferl. Bei den Cookies die wir verwenden handelt es sich um kleine Stücke Text (max. 4096 Zeichen) die wir dem Browser schicken können. Der Browser zeigt diesen Text aber nicht an, sondern speichert ihn. Das folgende Beispiel zeigt wie wir Cookies setzen, und wie wir alle Cookies ausgeben können:

```
<html>
    <body>
<%
    // first add the new cookie:
    String name = request.getParameter("name");
    String value = request.getParameter("value");
    if ((name != null) && (name.length() > 0)) {
        Cookie cookie = new Cookie(name, value);
        cookie.setMaxAge(365 * 24 * 60 * 60);   // one year
        response.addCookie(cookie);
    }

    // then list all cookies:
    Cookie[] cookies = request.getCookies();
    out.println( "<ul>");
    if (cookies != null) {
        for (int i = 0; i < cookies.length; i++) {
            out.println("<li>" + cookies[i].getName() + " = " +
            cookies[i].getValue() + "</li>");
        }
    }
    out.println( "</ul>");
%>
    </body>
</html>
```

Bei jedem Request schickt der Browser alle Cookies die er von uns bekommen hat mit. Das können bis zu 50 Cookies sein. Er schickt aber keine Cookies von anderen Websites, normalerweise.



Erst einmal scheinen Cookies relativ nutzlos, es stellt sich aber heraus, dass sie eine der nützlichsten Erfindungen im Zusammenhang mit dem Internet sind. Die Browser verstecken die Cookies gerne, aber wenn man lange genug sucht, findet man sie immer. Rechts ist ein Beispiel von Cookies die der Website "blogspot.de" auf meinem Rechner gespeichert hat. Die Cookies 'umta', 'umtb' usw. sind recht bekannt, da sie von Google Analytics kommen. Generell werden diese Art von Cookies verwendet um Leute zu trecken. Man kann aber alles mögliche in den Cookies speichern, so lange es weniger als 4 Kilobyte ist.

Cookies sind nicht besonders sicher. Obwohl eigentlich nur der Website der die Cookies gesetzt hat, diese auch auslesen können sollte, finden sich immer wieder Tricks das zu umgehen. Deswegen ist es nicht besonders schlau Passwörter in Cookies zu speichern.

SI: Man sollte nie Passwörter oder andere sensitive Informationen in Cookies speichern.

Editor

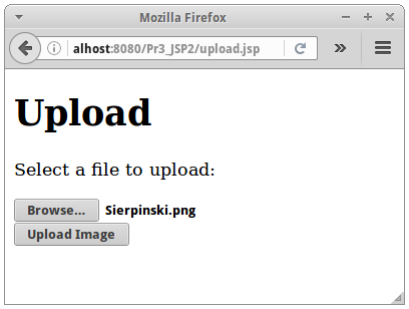
Als nächstes möchten wir weiter an unserem Texteditor schreiben. Auch dieses Beispiel besteht wieder aus zwei Teilen, einer *editor.jsp* und einer *editorLogic.jsp* Seite. Die Seite *editor.jsp* haben wir ja schon erstellt. Deswegen kommen wir gleich zur *editorLogic.jsp* Seite:

```
<%
    String text = request.getParameter("text");
    if (text != null) {
        try {
            FileWriter fw = new FileWriter("editor.txt");
            fw.write(text);
            fw.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    response.sendRedirect("editor.jsp");
%>
```

Wir versuchen also den Text den der User eingegeben hat in der Datei "editor.txt" zu speichern, und danach schicken wir den Nutzer einfach zur *editor.jsp* Seite zurück. Sollte allerdings etwas schief gehen, dann schreiben wir eine Fehlermeldung in den "System.out" Stream, besser wäre aber den Logger zu verwenden.

Es stellen sich hier die interessante Frage: wohin wird die Datei "editor.txt" geschrieben? Was passiert mit der Datei wenn der Server neu gestartet wird? Und was wenn die Applikation neu deployed wird?

Upload



Wenn wir es unseren Nutzern ermöglichen wollen Daten auf unseren Server hochzuladen, z.B. ein Profilbild, dann ist der erste Schritt relativ einfach:

```
<html>
    <body>
        <h1>Upload</h1>
        <p>Select a file to upload:</p>
        <form action="uploadLogic.jsp"
              method="POST"
              enctype="multipart/form-data">
            <input type="file" name="file"/><br/>
            <input type="submit" value="Upload Image"/>
        </form>
    </body>
</html>
```

Wir benutzen zum einen das "file" Inputfeld. Zum anderen müssen wir im "form" Tag noch mitteilen, dass die Daten vom Typ "multipart/form-data" sind.

Der zweite Schritt ist allerdings etwas komplizierter. Das hat damit zu tun, dass u.U. nicht nur eine Datei hochgeladen wird, sondern mehrere. Die muss man dann auseinanderfieseln. Da wir nicht die ersten sind die dieses Problem haben, gibt es in der Apache Commons Bibliothek [2] drei Klassen die uns dabei unterstützen:

```
<%
    int maxFileSize = 5000 * 1024;  // 5 MB
    int maxMemSize = 1000 * 1024;   // 1 MB

    boolean isMultipart =
        ServletFileUpload.isMultipartContent(request);
    if (isMultipart) {
        ServletContext servletContext =
            this.getServletConfig().getServletContext();

        // Create a factory for disk-based file items
        DiskFileItemFactory factory = new DiskFileItemFactory();
        factory.setSizeThreshold(maxMemSize);
        File repository = (File)
servletContext.getAttribute("javax.servlet.context.tempdir");
        factory.setRepository(repository);

        // Create a new file upload handler
        ServletFileUpload upload =
            new ServletFileUpload(factory);
        upload.setSizeMax(maxFileSize);

        // Parse the request
        List<FileItem> items = upload.parseRequest(request);
        Iterator<FileItem> iter = items.iterator();
        while (iter.hasNext()) {
            FileItem item = iter.next();
            if (!item.isFormField()) {
                String filePath = servletContext.getRealPath("/");
                File uploadedFile =
                    new File(filePath + item.getName());
                item.write(uploadedFile);
            }
        }
        out.println("Upload successful!");

    } else {
        out.println("No file uploaded");
    }
%>
```

Die Variable *filePath* enthält den Pfad wo unsere hochgeladenen Dateien gespeichert werden. Dateien auf unseren Server hochzuladen ist nicht ganz unproblematisch, deswegen haben wir einige Schranken eingebaut, was die Größe der Dateien angeht die wir akzeptieren. Unvorsichtige Nutzer oder Hacker können unseren Website in echte Schwierigkeiten bringen, wenn zu viele oder zu große Dateien hochgeladen werden.

SI: Hacker nutzen gerne Buffer Overflows um Zugriff auf unseren Server zu erlangen. Dies gelingt Ihnen manchmal durch das Hochladen sehr großer Dateien.

StateURL

Sehr häufig möchten wir Nutzer unseres Websites unterscheiden können, z.B. wenn die bei uns was kaufen wollen. Man könnte es mit der IP Adresse versuchen. Die taugt aber nicht, weil z.B. bei Firmen es so aussieht wie wenn alle Mitarbeiter der Firma die gleiche IP Adresse haben. Also brauchen wir einen andere Lösung.

Eine ganz einfache Lösung für das Problem ist das sogenannte "URL-Rewrite": dazu hängen wir an jede URL unseres Websites einen Zusatz, z.B.

```
...stateURL.jsp?sessionId=227
```

Wenn jetzt alle URLs die von Nutzer "227" kommen diesen Zusatz haben, können wir Nutzer "227" wiedererkennen. Im Code sieht das so aus:

```
<%
    String sessionId = request.getParameter("sessionId");
    if ( sessionId == null ) {
        sessionId = "" + (int)(Math.random()*1000);
    }
%>
<html>
    <body>
        <h1>State with URL-Rewrite</h1>
        <p>Your sessionId is <%= sessionId %></p>
        <p><a href="stateURL.jsp?sessionId=<%= sessionId %>">
            some link</a></p>
    </body>
</html>
```

Wir stellen also zunächst fest, ob der Nutzer bereits eine *sessionId* hat. Falls nein, generieren wir eine neue. Danach fügen wir an allen Links in unserem Website den Zusatz "?sessionId=227" an. Das kann man auch automatisch machen, aber uns geht es ja darum das Prinzip zu verstehen.

Wir können testen ob unser Beispiel auch wirklich funktioniert, in dem wir zwei verschieden Browser öffnen und unsere Seite besuchen. Die verschiedenen Browser sollten verschiedene *sessionIds* haben.

Ein paar Dinge die wir noch checken sollten:

- Was passiert wenn wir die gleiche Seite in verschiedenen Browsertabs öffnen?
- Könnte es passieren, dass zufällig zwei Nutzer die gleiche *sessionId* bekommen (collision)?
- Könnte sich ein Nutzer einfach als ein anderer ausgeben, indem er die *sessionId* des anderen übernimmt (session hijacking)?

StateCookie

Eine etwas einfachere Methode Nutzer eindeutig zu identifizieren ist mit Hilfe von Cookies. Das erste Mal wenn ein Nutzer auf unsere Seite kommt, kreieren wir ein Cookie mit dem Name "sessionId" und initialisieren es mit einem Zufallswert. Danach, können wir den Nutzer über sein Cookie eindeutig identifizieren.

```
<%
    String sessionId = null;
    // first check if cookie was set already
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
        for (int i=0; i<cookies.length; i++) {
            if (cookies[i].getName().equals("sessionId")) {
                sessionId = cookies[i].getValue();
            }
        }
    }
    // if no cookie was found create one
    if ( sessionId == null ) {
        sessionId = "" + (int)(Math.random()*1000);
        Cookie cookie = new Cookie("sessionId", sessionId);
        cookie.setMaxAge(1 * 60);   // expire in 1 min
        response.addCookie(cookie);
        response.sendRedirect("stateCookie.jsp");
        return;
    }
%>
```

```
<!DOCTYPE html>
<html>
    <body>
        <h1>State with Cookie</h1>
        <p>Your sessionId is <%= sessionId %></p>
        <p><a href="stateCookie.jsp">some link</a></p>
    </body>
</html>
```

Obwohl der JSP Code etwas komplizierter aussieht, ist dieser Ansatz einfacher, denn wir müssen nicht an jede URL etwas anhängen. D.h. unser HTML wird einfacher. Auch hier sollten wir erst mal wieder testen ob unser Beispiel auch wirklich funktioniert, in dem wir zwei verschieden Browser öffnen und unsere Seite besuchen.

Ein paar Dinge die wir noch checken sollten:

- Was passiert wenn wir die gleiche Seite in verschiedenen Browsertabs öffnen?
- Könnte es passieren, dass zufällig zwei Nutzer die gleiche *sessionId* bekommen (collision)?
- Könnte sich ein Nutzer einfach als ein anderer ausgeben (session hijacking)?
- Was passiert wenn der Nutzer keine Cookies zulässt?

NumberGuess

NumberGuess sollte aus dem ersten Semester noch bekannt sein. In dem Spiel "denkt" sich der Computer eine Zahl zwischen 0 und 99, und wir müssen diese erraten. Falls wir falsch raten, sagt uns der Computer ob seine Zahl höher oder niedriger als unsere geratene war.

Der Hauptunterschied zum ersten Semester ist, dass wir im Web nicht alleine sind. Soll heißen, es könnten ja mehr als nur eine Person auf unser Spiel zugreifen. Sollen die anderen dann die gleiche Zahl erraten, oder bekommt jede Person ihre eigene Zahl? Ersteres wäre ganz einfach, deswegen beschäftigen wir uns mit Zweiterem.

Mit unserem momentanen Kenntnisstand, gibt es zwei Möglichkeiten dies zu tun: mit Cookies oder mit dem Hidden-Tag. Da unsere Nutzer sicherheitsbewusst sind und keine Cookies zulassen, verwenden wir das Hidden-Tag.

```
<html>
    <body>
        <h1>NumberGuess</h1>
<%
    ...
%>
        <p>Enter your guess:</p>
        <form action="numberGuess.jsp" method="POST" >
            <input type="number" name="number" required="true"/>
            <input type="hidden" name="numberToGuess"
                                 value="<%= numberToGuess %>" />
            <input type="submit" value="Guess" />
        </form>
    </body>
</html>
```

Als erstes müssen wir feststellen, ob unsere Seite das erste Mal aufgerufen wurde. Das können wir feststellen indem wir checken ob der Parameter "number" schon gesetzt wurde. Falls nicht, dann generieren wir einfach eine Zufallszahl zwischen 0 und 99:

```
<%
    int numberToGuess = 0;
    if ( request.getParameter("number") == null ) {
        // we are here the first time
        Random rand = new Random();
        numberToGuess = rand.nextInt(99);
```

Falls ja, dann müssen wir die Zahl des Nutzers mit der Zahl des Computers vergleichen.

```
    } else {
        // we have been here before
        numberToGuess =
            Integer.parseInt( request.getParameter("numberToGuess") );
        int guess =
            Integer.parseInt( request.getParameter("number") );
        if ( numberToGuess == guess ) {
            out.println("You are the greatest!");
        } else if ( guess < numberToGuess ){
            out.println("Your guess is less than the number.");
        } else {
            out.println("Your guess is higher than the number.");
        }
    }
%>
```

Was noch fehlt ist ein Zähler, der mitzählt wie häufig die Person raten musste bis sie die richtige Antwort hatte. Das könnte man über ein zusätzliches Hidden-Tag lösen. Wollen wir aber nicht, weil wir im nächsten Kapitel eine bessere Lösung sehen werden (Session).

Bevor wir zum nächsten Projekt gehen, gibt es noch einige Dinge über die wir uns kurz Gedanken machen sollten:

- Wie kann man das Spiel abbrechen und neu starten?
- Was müsste man machen, damit das Formular am Ende, also wenn der Nutzer richtig geraten hat, nicht mehr gezeigt wird?
- Was passiert wenn der Nutzer gar nichts, oder Unsinn eingibt?
- Was würde passieren wenn der Nutzer "View Source" kennt, also sich den HTML Quelltext anzeigen lässt?

GuestBook

Bei der GuestBook Anwendung geht es darum, dass Besucher unserer Website Kommentare hinterlassen können. Die Anwendung besteht aus zwei Teilen, dem ersten, *guestBook.jsp*, in dem die Nutzer ihre Kommentare schreiben können, das ist einfach ein Formular mit einer Textarea.

Im zweiten Teil, *guestBookLogic.jsp*, schreiben wir den Kommentar erst Mal in eine Textdatei:

```
<%@page import="java.io.*"%>
<%
    // write comment to file:
    String text = request.getParameter("comment");
    if (text != null) {
        try {
            FileWriter fw = new FileWriter("guestBook.txt", true);
            fw.write("<li>" + text + "</li>");
            fw.close();
        } catch (Exception e) {
            // should log error to log file
        }
```

```
    }

    // show guest book:
%>
```

Anschließend zeigen wir den Inhalt dieser Textdatei an, damit wir alle Kommentare lesen können:

```
<html>
    <body>
        <h1>GuestBook</h1>
        <ul>
<%
    try {
        BufferedReader br = new BufferedReader(
                new FileReader("guestBook.txt") );
        while (true) {
            String line = br.readLine();
            if ( line == null ) break;
            out.println( line );
        }
        br.close();
    } catch (Exception e) {
        // should log error to log file
    }
%>
        </ul>
    </body>
</html>
```

Wichtig ist, dass wir beim FileWriter das "true" Flag setzen, das bedeutet, dass er eine existierende Datei nicht löscht, sondern die neuen Einträge anhängt.

Auch in dieser einfachen Anwendung gibt es wieder einiges zu bedenken:

- Was passiert wenn der Nutzer so Unsinn wie "<script>alert('hi');</script>" eingibt (XSS)?
- Manche Nutzer lieben Schimpfwörter und haben nichts besseres zu tun, als den ganzen Tag Webformulare mit selbigen zu befüllen. Wie könnte man das verhindern (Stopwords)?
- Wo wird die Datei "guestBook.txt" eigentlich gespeichert?
- Was würde passieren wenn ein Nutzer sehr viel Text (Gigabytes oder Terrabytes) eingibt?

SI: Wir sollten niemals Daten trauen die vom Nutzer kommen und immer mit dem Schlimmsten rechnen.

HighScore

Für viele Spiele benötigt man eine Highscore Liste. Die sollte natürlich idealerweise sortiert sein, und zwar derart, dass der Spieler mit dem höchsten Score als erstes gelistet wird. Wenn wir uns an unser zweites Semester erinnern, dann sollte eigentlich die TreeMap die Datenstruktur unserer Wahl sein:

```
TreeMap<Integer,String> map =
    new TreeMap<Integer,String>(Collections.reverseOrder());
```

Der Key ist der Score und der Value ist die UserId des Nutzers.

Auch hier benötigen wir wieder zwei Seiten: über die *highScore.jsp* fügen wir einen neuen Score hinzu. Die *highScoreLogic.jsp* Seite liest zunächst die alten Scores von einer Datei namens "highScores.txt", fügt den neuen Wert hinzu, speichert die Scores, und abschliessend werden sie angezeigt. Unsere Seite besteht aus drei Teilen: der Definition der Methoden,

```
<%!
    private static final String HIGH_SCORE_FILE_NAME =
        "highScores2.txt";

    private TreeMap<Integer, String> readScoresFromFile() {
        TreeMap<Integer, String> map = null;

        if (!(new File(HIGH_SCORE_FILE_NAME)).exists()) {
            map =
            new TreeMap<Integer, String>(Collections.reverseOrder());

        } else {
            try {
                ObjectInputStream ois = new ObjectInputStream(
                        new FileInputStream(HIGH_SCORE_FILE_NAME));
                map = (TreeMap<Integer, String>) ois.readObject();
                ois.close();
            } catch (Exception e) {}
        }

        return map;
    }

    private void writeScoresToFile(TreeMap<Integer, String> map) {
        try {
            ObjectOutputStream oos = new ObjectOutputStream(
                    new FileOutputStream(HIGH_SCORE_FILE_NAME));
            oos.writeObject(map);
            oos.close();
        } catch (Exception e) {}
    }
%>
```

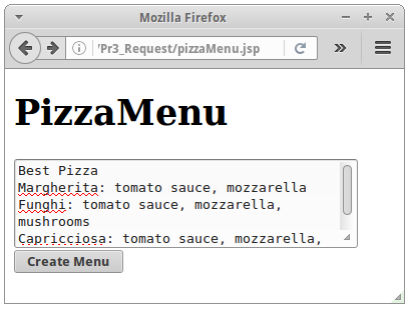
dem JSP das die Parameter einliest,

```
<%
    // save score to file:
    String userId = request.getParameter("userId");
    String score = request.getParameter("score");
    TreeMap<Integer, String> sortedScoreMap = readScoresFromFile();
    if ((userId != null) && (score != null)) {
        sortedScoreMap.put(Integer.parseInt(score), userId);
        writeScoresToFile(sortedScoreMap);
    }
%>
```

und dem HTML, dass die HighScores anzeigt:

```
<html>
    <body>
        <h1>HighScore</h1>
        <ol>
<%
    for (Integer scr : sortedScoreMap.keySet()) {
        out.println("<li>" +scr+": "+sortedScoreMap.get(scr) +
                    "</li>");
    }
%>
        </ol>
    </body>
</html>
```

PizzaMenu



Im letzten Kapitel haben wir uns schon darum gekümmert, dass das Menü unseres Pizza Restaurants gut aussieht. Was wir jetzt noch machen wollen ist, dass das Menü einfach zu ändern ist, was ja in guten Restaurants häufiger vorkommen soll.

Die Anwendung besteht wieder aus zwei Seiten, einer *pizzaMenu.jsp* und einer *pizzaMenuLogic.jsp*. Erstere besteht einfach aus einer Textarea und einem Submit Knopf. Und Zweitere haben wir eigentlich fast schon geschrieben, wir müssen eigentlich nur noch den String *menu* aus dem Parameter holen:

```
<%
    String menu = request.getParameter("menu");
    if (text != null) {
        ...
    }
```

Wir wollen aber noch einen Schritt weitergehen: anstelle wie wir es im letzten Kapitel getan haben, das generierte Menu einfach an den Browser zu schicken, wollen wir es in eine Datei namens *pizzaMenu.html* generieren:

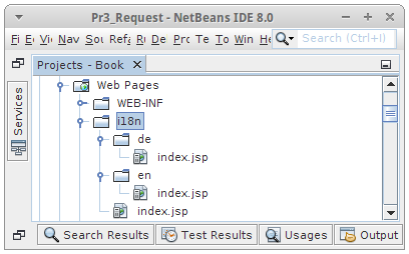
```
<%
    ...
        try {
            String path =
                request.get ServletContext().getRealPath("/");
            FileWriter fw = new FileWriter(path+"pizzaMenu.html");
            fw.write("<!DOCTYPE html><html><body>");
            fw.write("<h1>" + br.readLine() + "</h1>");
            fw.write("<ul>");

            while (true) {
                String line = br.readLine();
                if (line == null) break;
                String[] pizz = line.split(":");
                fw.write("<li><strong>" + pizz[0] +":</strong> "+
                    pizz[1] + "</li>");
            }

            fw.write("</ul>");
            fw.write("</body></html>");
            fw.close();
        } catch (Exception e) {
            // should log error to log file
        }
    }

    response.sendRedirect("pizzaMenu.html");
%>
```

Anschließend machen wir einen Redirect auf die *dynamisch generierte* HTML Seite.

Dies ist unser erstes Beispiel für "dynamic code generation", in diesem Fall HTML. Es ist eine beliebte Technik für High-Volume Websites bei denen sich die Daten nicht zu häufig ändern. Craigslist [3] verwendet diesen Trick.

Im Allgemeinen gilt allerdings, dass man es vermeiden sollte in Webanwendung Dateien zu benutzen. Alternativen sind das *session* oder *application* Objekt oder eine Datenbank. Beides werden wir in den kommenden Kapiteln sehen.

i18n



Das "Inter" in Internet steht ja für international. Es gibt jetzt manche Leute (zu denen ich gehöre) die sagen, dann machen wir eben alles auf Englisch. Es gibt aber viel mehr Leute die ihre Webseiten aber lieber in ihrer eigenen Sprache lesen. Das ist das Problem das man "Internationalization" nennt, kurz i18n.

Wie wir ja vor kurzem gelernt haben, sagt uns der Browser welche Sprache er spricht. Und wir können diese Information verwenden um Besucher unserer Website je nach Sprache umzuleiten:

```
<%
   String userLocale = request.getHeader("Accept-Language");
   Locale locale = request.getLocale();
   String userlanguage = locale.getDisplayLanguage();

   if ( "de".equals(locale.getLanguage())) {
       response.sendRedirect("de/");
       return;
   }
   response.sendRedirect("en/");
   return;
%>
```

In unserem Beispiel haben wir also eine zentrale *index.jsp* Seite, die den Code oben enthält. Zusätzlich gibt es aber für jede Sprache die wir unterstützen wollen ein Unterverzeichnis, idealerweise mit dem Landeskürzel als Namen, und darin befinden sich dann die sprachspezifischen Seiten. Wenn wir die Anwendung testen wollen, dann müssen wir die Spracheinstellungen in unserem Browser ändern, das ist manchmal ganz einfach (bei Opera) und manchmal nicht.

Der Nachteil dieses Verfahrens ist, dass wir quasi für jede Sprache die wir unterstützen wollen eine komplette Kopie unseres Websites anlegen müssen. Das sorgt für einen gewaltigen Wartungsaufwand. Es gibt hier auch andere Lösungen, in der Regel sind das Frameworks wie Struts [5], JSF [6] oder Spring [7].

EscapeXml

Kommen wir zu einem Problem mit dem wir es immer wieder zu tun haben werden: trauen wir dem Input unserer Nutzer? Die Antwort ist ganz klar: Nein! Allerdings sollten wir zwischen zwei Nutzertypen unterscheiden: denen die unabsichtlich etwas falsch machen, also den Trotteln, und denen die absichtlch versuchen unserem Website Schaden zuzufügen, den Hackern.

Zwei Standardverfahren die Hacker gerne anwenden sind Cross-Site Scripting (XSS) und SQL Injection. Wir wollen ersteres demonstrieren und dann zeigen wie man sich dagegen schützt. Als erstes schreiben wir eine Seite *escapeXML.jsp* die aus einem Formular mit einer Textarea einer Checkbox und einem Submit Knopf besteht. In der Textarea soll folgendes JavaScript stehen, dass hypothetisch ein böser Hacker da reingeschrieben hat:

```
<script type="text/javascript">
alert("Hi from JavaScript!");
</script>
```

Das Ziel dieses Formulars ist die Seite *escapeXmlLogic.jsp*:

```
<%@page import="de.variationenzumthema.internet.Utility"%>
<%
    String text = request.getParameter("text");
    if (request.getParameter("escape") != null) {
        out.println( Utility.escapeXml(text) );
    } else {
```

```
            out.println(text);
        }
    %>
```

Je nachdem, ob wir die Checkbox markiert haben oder nicht, wird einmal der Userinput einfach ausgegeben mittels *out.println(),* und das andere Mal wird der Userinput noch durch folgende Methode gejagt:

```
    public static String escapeXml(String in) {
        in = in.replace("&", "&amp;");
        in = in.replace("\"", "&#034");
        in = in.replace("'", "&#039");
        in = in.replace("<", "&lt;");
        in = in.replace(">", "&gt;");
        return in;
    }
```

Das Resultat ist, dass im einen Fall der Browser JavaScript ausführt (kann gefährlich sein), im anderen Fall aber nicht. Ganz ähnlich funktioniert auch die *esacpeSQL()* Mehode. Da wir diese Methoden so häufig brauchen werden (und auch die weiter unten), haben wir diese in der *Utility* Klasse zusammengefasst.

SI: Wir sollten dem Input unserer Nutzer niemals trauen.

Register**

Wir werden es immer wieder mit längeren Formularen zu tun haben, aus denen wir dann ein Java Objekt (POJO) befüllen sollen. Ein Möglichkeit ist dies von Hand zu tun, so wie wir das bisher gemacht haben. Eine andere ist sich eine Konvention zu überlegen und dann die Magie von Reflection zu verwenden.

Die Konvention ist folgende: nehmen wir an wir haben ein Formular mit folgenden Input Feldern:

```
<form action="registerLogic.jsp" method="POST">
    <p>
        <label>Email:</label>
        <input type="text" name="emailId"/>
    </p>
    <p>
        <label>Password:</label>
        <input type="password" name="password"/>
    </p>
    ...
</form>
```

Dann deklarieren wir eine entsprechende Klasse, die für jedes der Input Felder ein Attribut mit gleichem Namen hat:

```
public class User {
    private String emailId;
    private String password;
    ...
}
```

Das ist unsere Konvention.

Und jetzt möchten wir, dass es eine Methode namens *extractObjectFromRequest()* gibt, der wir einfach das *request* Objekt übergeben, und die uns dann ein fertiges *User* Objekt zurückgibt:

```
<%
    User usr =
        (User)Utility.extractObjectFromRequest(User.class, request);
%>
```

und natürlich soll die Methode das für jede mögliche Klasse tun. Und überraschenderweise geht das mit eine paar Zeilen Java Code und Reflection Magic:

```
public static Object extractObjectFromRequest(Class c,
    HttpServlet Request request) {
    Object o = null;
    try {
        o = c.newInstance();
        Field[] flds = c.getDeclaredFields();
        for (Field f : flds) {
            String param = request.getParameter(f.getName());
            if ((param != null) && (param.length() > 0)) {
                f.setAccessible(true);
                Object par = null;
                if (f.getType().getName().equals(
                                        "java.lang.String")) {
                    par = param;
                } else if ...
                    ...

                } else {
                    System.err.println("Unknown type: " +
                                        f.getType().getName());
                }

                if (par != null) {
                    f.set(o, par);
                }
            }
        }
    } catch (Exception e) {
        System.err.println(e);
    }
    return o;
}
```

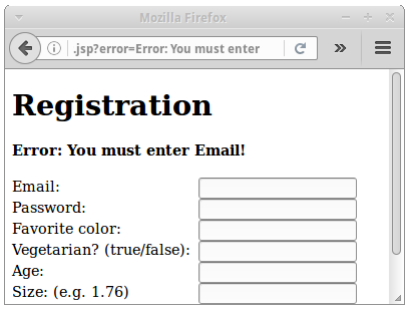
Cool, oder?

Nun wenn wir das auch noch mit der *escapeXML()* und e*scapeSQL()* Methode kombinieren, sind wir schon auf der ziemlich sicheren Seite.

SEP: Wir sollten bei der Wahl einer Programmiersprache darauf achten, dass diese Reflection unterstützt.

Feedback

Manchmal brauchen unsere Nutzer Feedback wenn sie etwas falsch eingegeben haben. Das kann man sehr einfach lösen wenn man das Feedback einfach beim Redirekt an die URL anhängt:




```
<%
    response.sendRedirect(
        "register.jsp?error=Error: You must enter Email!");
    return;
%>
```

und dann an passender Stelle im Formular ausgibt, evtl. sogar in roter Farbe:

```
<%
    if ( request.getParameter("error") != null ) {
        out.println("<p><strong>"+
            request .getParameter("error")+
            "</strong></p><br/>");
    }
%>
```

Man kann das auch über den RequestDispatcher machen, wenn man nicht möchte das die Fehlermeldung Teil der URI ist. Später mehr zum RequestDispatcher.

Pre-Fill**

Was wirklich ätzend ist, wenn man ein langes Formular ausgefüllt hat, einen klitze-kleinen Fehler gemacht hat, eine komische Fehlermeldung bekommt (wie oben) und dann das ganze Formular noch einmal ausfüllen muss. Das kann doch nicht sein.

Hier gibt es zwei Möglichkeiten: man könnte das serverseitig machen. Das ist aber überraschend unschön. Die andere Möglichkeit ist das clientseitig mit JavaScipt zu tun. Der Trick ist JavaScript dynamisch zu generieren und diese dann in die Seite einzufügen:

```
<html>
    <head>
<%
    User usr = new User("ralph@lano.de", Color.GREEN, true, 42, 1.73,
new Date());
    String javaScript = Utility.createJavaScriptFromObject(usr);
    out.println(javaScript);
%>
    </head>
    <body onload="fillFormFromJS()">
        <h1>Pre-Fill</h1>
        ...
    </body>
</html>
```

Wir gehen davon aus, dass die generierte JavaScript Methode *fillFormFromJS()* heißt. Nach dem Laden der Seite, wird diese aufgrund des *onload* Events im <body> Tag ausgeführt.

Die Methode *fillFormFromJS()* generieren wir dynamisch aus dem POJO, in diesem Fall der Klasse User. Wir definieren also einen String *js* in dem wir das JavaScript generiern:

```
public static String createJavaScriptFromObject(Object o) {
    String js = "";
    js += "<script>";
    js += "function fillFormFromJS() {";
    js += "var map = {";

    try {
        Field[] flds = o.getClass().getDeclaredFields();
        for (Field f : flds) {
            f.setAccessible(true);
            String key = f.getName();
            Object value = f.get(o);
            if (value instanceof String) {
                js += "'" + key + "': '" + (String) value + "',";
            } else if (value instanceof Integer) {
```

```
                    js += "'" + key + "': '" + (Integer) value + "',";
                } else if (value instanceof Double) {
                    js += "'" + key + "': '" + (Double) value + "',";
                } else if (value instanceof Boolean) {
                    js += "'" + key + "': '" + (Boolean) value + "',";
                } else if (value instanceof Color) {
                    js += "'" + key + "': '" +
                          convertColorToName((Color) value) + "',";
                } else if (value instanceof Date) {
                    //2016-08-17
                    DateFormat format =
                        new SimpleDateFormat("yyyy-MM-dd");
                    js += "'" + key + "': '" +
                        format.format((Date) value) + "',";
                }
            }

        } catch (Exception e) {
            System.err.println(e);
        }

        js += "};";
        js += "for(var key in map) {";
        js += "document.getElementsByName(key)[0].value = map[key];";
        js += "}";
        js += "}";
        js += "</script>";
        return js;
    }
```

Dann gehen wir alle Felder des übergebenen Objekts o durch, benutzen etwas Reflection-Magic (rot markiert), und füllen eine JavaScript Map mit key-value Paaren. Hier ist der Key der Name des jeweiligen Form Elements und value ist der Wert den es haben soll. Mit der kleinen Schleifen am Ende dann,

```
        for(var key in map) {
            document.getElementsByName(key)[0].value = map[key];
        }
```

setzen wir die Input Felder auf die gewünschten Werte. Damit das Ganze funktioniert, müssen die Form Elemente die gleichen Namen haben wie die Attribute der User Klasse. Man nennt das auch "coding by convention". Außerdem, ist das unser zweites Beispiel für "generative coding", in diesem Fall generieren wir JavaScript Code.

Research

Manche Themen haben wir wieder mal nur sehr oberflächlich behandelt. Man könnte sich aber das eine oder andere noch etwas detailierter ansehen.

Cookies

Zu Cookies gibt es immer ganz viele Fragen. Z.B. was sind den "Secure Cookies" oder "Super-Cookies"? Auch interessant, wo werden die Cookies den gespeichert? Interessant ist auch ein Firefox Plugin namens Lightbeam.

RFC 3986

RFC 3986 definiert u.a. den Uniform Resource Identifier (URI). Auch diese Spezifikation sollten wir uns mal ansehen, und auch darin nach bekannten Namen unter den Autoren suchen. Außerdem sollten wir uns den Unterschied zwischen URI, URL und URN klar machen.

Cross-Site Scripting und SQL Injection

Wir sollten im Internet mal nachlesen was es denn mit "Cross-Site Scripting (XSS)" und "SQL Injection" so auf sich hat.

Reflection

Leider wird nirgendwo die Magie der Reflection ordentlich unterrichtet, ausser vielleicht in Hogwarts. Aber mit Sicherheit gibt es im Internet irgendwo ein gutes Tutorial. Bitte Bescheid geben, falls Sie eines finden!

Generative Coding und Coding by Convention

Im letzten Abschnitt haben wir die Begriffe "Generative Coding" und "Coding by Convention" mehrmals gesehen. Auch hier macht es Sinn sich mal schlau zu machen was denn dahinter steckt.

Fragen

1. Nennen Sie drei Unterschiede zwischen einer HTTP GET und POST Anfrage.
2. Welche Informationen enthält der HTTP Header? Nennen Sie fünf.
3. Welche Informationen kann man aus dem JSP "request" Objekt beziehen? Nennen Sie drei Beispiele.

Referenzen

Anbei die Referenzen die in diesem Kapitel verwendet wurden.

[1] Uniform Resource Identifier, https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

[2] Apache Commons Bibliothek, https://commons.apache.org/

[3] Craigslist, https://nuremberg.craigslist.de/

[4] RFC 3986, https://www.ietf.org/rfc/rfc3986.txt

[5] Apache Struts, https://struts.apache.org/

[6] JavaServer Faces Technology,, JSF, www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

[7] spring, https://spring.io/

Session and Application

Bisher sind unsere Webanwendungen sehr dumm, sie haben kein Gedächtnis, weder Kurzzeit noch Langzeit. Mit den Session und Applikation Objekten erhalten wir auf einmal ein Kurzzeitgedächtnis, d.h. wir können uns auf einmal Dinge merken. Und damit ergeben sich viele neue Anwendungsszenarien.

Session

Die wichtigsten Anwendungen im Internet haben mit Shopping zu tun. Und das zentrale Konzept hinter jeder Shopping Anwendung ist der Warenkorb. Der Warenkorb ist einfach eine Liste von Artikeln, z.B. Büchern, die unser Kunde gerne kaufen möchte.

Ganz wichtig ist natürlich, dass jeder Kunde seinen eigenen Warenkorb hat. Und genau für diesen Zweck wurde das *session* Objekt erfunden. Eine "Session" ist sozusagen eine "Einkaufs-Session". Jeder Nutzer hat seine eigene Session. Und so wie es in JSP die Objekte *out*, *request* und *response* bereits gibt, gibt es auch das *session* Objekt bereits vordefiniert.

Das *session* Objekt ist einfach eine HashMap mit beliebigen key-value Paaren. Die Benutzung ist denkbar einfach:

```
<%
    session.setAttribute("key", "value");
    String v = (String) session.getAttribute("key");
%>
```

Das wirklich schöne ist, dass wir von jeder unserer JSP Seiten auf das *session* Objekt zugreifen können. Bisher war es immer kompliziert wenn wir Daten zwischen zwei Seiten transferieren wollten. Im letzten Kapitel haben wir drei Varianten kennengelernt:

- das "Reverse Text" Beispiel benutzte einem Link,
- das "Captcha" Beispiel benutzte ein Hidden Tag,
- das "Cookies" Beispiel benutzte Cookies.

Aber wenn wir kurz nachdenken, sehen wir dass eigentlich alle immer über das *request* Objekt gingen. Außerdem mussten die Daten immer über den Browser gehen, was z.B. bei unserer Captcha Anwendung dazu führte, dass sie ganz einfach zu knacken war.

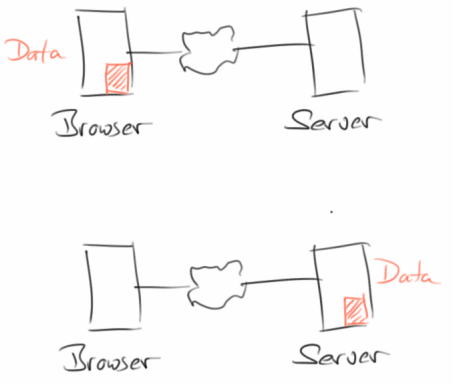
Captcha

Betrachten wir das Captcha Beispiel aus dem letzten Kapitel noch einmal. Anstelle das Hidden-Tag zu verwenden, würden wir die richtige Antwort in der Session speichern:

```
<%
    int a = (int) (Math.random() * 9) + 1;
    int b = (int) (Math.random() * 9) + 1;
    session.setAttribute("correctAnswer", a+b);
%>
```

um dann im *captchaLogic.jsp* das Resultat wieder aus der Session zu holen:

```
<%
    String sum = request.getParameter("sum");
    Integer result = (Integer) session.getAttribute("correctAnswer");
    ...
%>
```



Der Rest des Codes bleibt gleich. Versuchen wir jetzt aber mal diese neue Variante von Captcha zu knacken. Mit View Source sehen wir nichts. Und auch sonst lässt sich diese neue Version nicht knacken. Das hat damit zu tun, dass das Resultat, also die *correctAnswer*, nie zum Browser geschickt wird, sie verlässt den Server nie. Es ist ganz wichtig das zu verstehen!

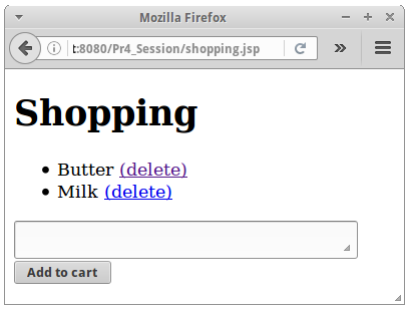
SEP: Daten die wir im *session* Objekt speichern werden immer nur auf dem Server gespeichert.

Expiration

Daten die im *session* Objekt gespeichert sind werden also auf dem Server gehalten, der Browser bekommt die nie zu sehen. Das kann aber auch zu einem Problem werden, und zwar wenn wir zu viele Nutzer haben, oder zu große Datenmengen in der *session* Hashmap speichern. Deswegen haben Sessions immer einen *Timeout*: wenn eine Session für eine bestimmte Zeit (meist 15 Minuten) nicht genutzt wird, wird sie einfach gelöscht. Und deswegen sollten wir auch nicht zu große Datenmengen in Sessions speichern.

Wie funktioniert das mit der Session? Wir könnten das auch selbst machen, wie wir in den Beispielen StateURL und StateCookie im letzten Kapitel gesehen haben, aber warum kompliziert wenn es auch einfach geht.

Shopping



Der Klassiker für den Einsatz von Sessions ist die Shopping Anwendung. In unserer sehr einfachen Shopping Anwendung gibt es einen Warenkorb (einfach eine Liste), und der Kunde kann Artikel zum Warenkorb hinzufügen oder wieder aus dem Warenkorb entfernen. Und natürlich wollen wir den Inhalt des Warenkorbs auflisten.

```
<html>
    <body>
        <h1>Shopping</h1>
<%
    Vector<String> cart =
        (Vector<String>) session.getAttribute("Cart");
    if ( cart == null ) {
        cart = new Vector<String>();
        session.setAttribute("Cart", cart);
    }

    out.println( "<ul>" );
    for (int i = 0; i < cart.size(); i++) {
        String item = cart.get(i);
        out.println( "<li>" );
        out.println( item );
        out.println(
            "  <a href='shoppingLogic.jsp?id="+i+"'>(delete)</a>" );
        out.println( "</li>" );
    }
    out.println( "</ul>" );
%>
    <form action="shoppingLogic.jsp" method="POST">
        <textarea name="item" rows="1" cols="40"></textarea>
        <input type="submit" value="Add to cart" />
    </form>
    </body>
</html>
```

Das Löschen und Hinzufügen passiert in der *shoppingLogic.jsp* Seite.

```
<%
    // add a new item
    String item = request.getParameter("item");
    if ( (item != null) && (item.length() > 3) &&
         (item.length() < 1000) ) {
        Vector<String> cart =
            (Vector<String>) session.getAttribute("Cart");
        cart.add( item );
    }
```

```
        // delete existing item
        String idToDelete = request.getParameter("id");
        if ( idToDelete != null) {
            Vector<String> cart =
                (Vector<String>) session.getAttribute("Cart");
            int id = Integer.parseInt( idToDelete );
            if ( ( id >= 0 ) && ( id < cart.size() ) ) {
                cart.remove( id );
            }
        }
        response.sendRedirect("shopping.jsp");
        return;
    %>
```

Wenn wir fertig sind schicken wir den Nutzer einfach zurück zur *shopping.jsp* Seite.

Login

Kommen wir zu unserem nächsten Klassiker: der Login Seite. Praktisch jeder Website hat eine. Im Prinzip ist das Schema immer das Gleiche: Es gibt ein paar Seiten, die kann jeder sehen, es gibt aber auch einen geschützten Bereich, protected, den kann man nur sehen, wenn man sich vorher eingeloggt hat.

Wir brauchen also mindestens drei Seiten: die *login.jsp* Seite, die jeder sehen kann, dann eine *protected.jsp* Seite, die kann man nur sehen wenn man sich erfolgreich eingeloggt hat, und eine *loginLogic.jsp* Seite, die checkt, ob man die richtige Benutzernamen-Passwort Kombination eingegeben hat. Um das Ganze etwas einfacher testen zu können fügen wir noch eine Navigationsleiste in jede Seite mit ein. Ach ja, und ausloggen sollten wir uns auch noch können, also eine Seite *logout.jsp* brauchen wir noch.

Die *login.jsp* Seite ist trivial, da brauchen wir gar nichts mehr dazu zu sagen. Interessanter wird die *loginLogic.jsp* Seite.

```
    <%
        String id = request.getParameter("userId");
        String passwd = request.getParameter("password");
        if ((id != null) && (passwd != null)) {
            if ((id.equals("ralph")) && (passwd.equals("123456"))) {
                session.setAttribute("User", id);
                response.sendRedirect("protected.jsp");
                return;
            }
        }
        session.removeAttribute("User");
        response.sendRedirect("login.jsp");
        return;
    %>
```

Wir checken also ob der Nutzer den richtigen Benutzernamen und das richtige Passwort eingegeben hat. Wenn ja, dann fügen wir einen neuen Eintrag in das Session Map mit dem Key "User" ein und schicken ihn zur Seite *protected.jsp*. Falls nein, dann löschen wir das Attribute "User" sicherheitshalber, und schicken den Nutzer zurück zu *login.jsp* Seite. Es ist ganz wichtig, nach dem *response.sendRedirect()* immer ein *return* zu setzen, denn ohne, würden im Beispiel oben die Zeilen danach noch ausgeführt werden.

Interessant ist jetzt wie wir die Seite *protected.jsp* schützen. Das geht überraschend einfach: wir müssen nur nachschauen, ob das Attribute "User" gesetzt wurde oder nicht:

```
<%
    String user = (String) session.getAttribute("User");
    if ( user == null ) {
        response.sendRedirect("login.jsp");
        return;
    }
%>
<!DOCTYPE html>
<html>
    <body>
        <%@include file="navigation.jsp" %>
        <h1>Protected</h1>
        <p>Welcome <%= user %>.</p>
    </body>
</html>
```

Falls es nämlich nicht gesetzt wurde, dann darf der Nutzer die Seite nicht sehen, wir schicken ihn einfach zur *login.jsp* Seite zurück. Ansonsten darf er die Seite sehen. Wir müssten das natürlich mit jeder Seite tun die wir schützen wollen. Das ist zwar etwas umständlich, aber funktioniert ganz gut. In zwei Kapiteln werden wir sehen wie man das auch weniger umständlich machen kann.

Bleibt noch zu klären wie das mit dem logout funktioniert. Das ist eigentlich ein Einzeiler:

```
<%
    session.removeAttribute("User");
    response.sendRedirect("login.jsp");
    return;
%>
```

Wir löschen also das "User" Attribut aus der Session und schicken den Nutzer zurück auf Los.

SEP: Nach einem sendRedirect() sollte immer ein *return* Statement folgen.

Include Directive

Noch eine kurze Anmerkung zur "include directive", die haben wir schon mal kurz im ersten Kapitel benutzt ohne zu wissen wie das wirklich funktioniert.

```
<html>
    <body>
        <%@include file="navigation.jsp" %>
        <h1>Login</h1>
        ...
    </body>
</html>
```

Wir verwenden sie zur Vermeidung von doppelten Code. Was das Include macht, es fügt die einzufügende Datei, in der Regel eine JSP Datei, an der Stelle ein, an der das Tag steht. Es ist also so, wie wenn der Inhalt von navigation.jsp an der Stelle stehen würde. Das passiert bevor aus der JSP Seite ein Servlet generiert wird. Das führt dazu, dass das Inkludieren zur "compile time" passiert, also die Seite wird schon beim Kompilieren eingefügt, macht also unseren Website nicht langsamer. Deswegen können wir soviele Dateien einfügen wie wir lustig sind. Anbieten tut es sich natürlich ganz speziell für Header und Footer, die auf allen Seiten gleich sein sollen.

Application

Mit dem *session* Objekt können wir Daten zwischen verschiedenen Seiten eines Nutzers austauschen. Wie können wir aber Daten zwischen verschiedenen Nutzern austauschen, oder besser wie können wir Daten zentral mehreren Nutzern zur Verfügung stellen? Dafür gibt es das *application* Object. Es funktioniert genauso wie das *session* Objekt, nur dass eben jeder von überall darauf zugreifen kann. Sozusagen eine Art "globale" Hashmap.

```
<%
    application.setAttribute("key", "value");
    String v = (String) application.getAttribute("key");
%>
```

Visitor

Schauen wir unser Visitor Beispiel aus dem zweiten Kapitel noch einmal an. Damals haben wir Instanzvariablen verwendet um einen globalen Zähler zu implementieren. Die Lösung hatte aber Probleme wie wir damals bereits angedeutet haben.

Eine besser Lösung einen Visitor Counter zu implementieren ist mit Hilfe des *application* Objekts. Der Code ist ganz einfach, wir speichern den Zähler im *application* Objekt. Das einzige was noch zu checken ist, dass der Zähler beim allerersten Aufruf auf eins gesetzt wird.

```
<%
    Integer counter =
        (Integer)application.getAttribute("visitorCounter");
    if ( counter == null ) {
        counter = 1;
    } else {
        counter++;
    }
    application.setAttribute("visitorCounter", counter);
%>
<html>
    <body>
        <h2>Welcome, Visitor Nr. <%= counter %></h2>
    </body>
</html>
```

Diese neue Version löst fast alle Probleme unserer alten Version, bis auf eines: wenn der Server neu gestartet wird, dann wird der Zähler wieder zurück gesetzt. Die Lösung dafür sind *jspInit()* und *jspDestroy()*. Für unseren "new and improved" Visitor Counter bedeutet das, dass wir wenn die *jspDestroy()* Methode aufgerufen wird, den momentanen Wert des Zählers in eine Datei (oder später Datenbank) speichern, und wenn die *jspInit()* Methode aufgerufen wird, den Wert des Zählers aus der Datei lesen.

Guestbook

Ein weiteres typisches Anwendungsszenario für das *application* Objekt ist das klassische Guestbook. Im Guestbook können Nutzer unseres Sites Kommentare hinterlassen. Foren funktionieren auch ganz ähnlich. Da jeder auf dem Guestbook Kommentare hinterlassen kann, bietet es sich an diese im *application* Objekt zu speichern.

In unsere Guestbook Anwendung sollen also Nutzer Kommentare abgeben können, die Kommentare aller Nutzer sollen aufgelistet werden, und es soll auch die Möglichkeit geben Kommentare zu löschen. Unsere Anwendung besteht wieder aus zwei Teilen, der *guestbook.jsp*, die das Anzeigen der Kommentare übernimmt, die Links zum Löschen vom Kommentaren enthält und einer Textarea um neue Kommentare abzugeben. Interessant ist, dass der Code identisch mit dem der Shopping List ist, lediglich *session* wird durch *application* ersetzt.

```
<html>
    <body>
        <h1>Guestbook</h1>
<%
    Vector<String> comments =
        (Vector<String>) application.getAttribute("Guestbook");
    if (comments == null) {
        comments = new Vector<String>();
        application.setAttribute("Guestbook", comments);
    }
    out.println("<ul>");
    for (int i = 0; i < comments.size(); i++) {
        String msg = comments.get(i);
        out.println("<li>");
        out.println(msg);
        out.println(
        "  <a href='guestbookLogic.jsp?id=" + i + "'>(delete)</a>");
        out.println("</li>");
    }
    out.println("</ul>");
%>
        <form action="guestbookLogic.jsp" method="POST">
            <textarea name="comment" cols="40" rows="1"></textarea>
            <input type="submit" value="Post new comment" />
        </form>
    </body>
</html>
```

Außerdem gibt es wieder eine Seite, *guestbook Logic.jsp*, für die Logik, die also neue Kommentare zum *application* Objekt hinzufügt, oder exisitierende löscht.

```
<%
    // add a new guestbook entry
    String comment = request.getParameter("comment");
    if ( (comment != null) && (comment.length() > 5) &&
         (comment.length() < 1000) ) {
        Vector<String> comments =
            (Vector<String>)application.getAttribute("Guestbook");
        comments.add( comment );
    }

    // delete existing guestbook entry
    String idToDelete = request.getParameter("id");
    if ( idToDelete != null) {
        Vector<String> comments =
            (Vector<String>)application.getAttribute("Guestbook");
        int id = Integer.parseInt( idToDelete );
        if ( ( id >= 0 ) && ( id < comments.size() ) ) {
            comments.remove( id );
        }
    }
    response.sendRedirect("guestbook.jsp");
    return;
%>
```

Wie wir im Code sehen, nehmen wir ein paar einfach Checks des User-Inputs vor. Allerdings vor XSS Attacken bewahrt uns das nicht, wenn wir aber unsere *escapeXml()* Methode verwenden würden, wären wir auch dagegen gefeit.

Dictionary

Ein sehr schönes Beispiel sowohl für die Nutzung des *application* Objekts, als auch die Nutzung der *jspInit()* Methode ist unsere Dictionary Anwendung. Es geht darum Wörter aus dem Englischen ins Deutsche zu übersetzen.

Das Wörterbuch selbst ist in einer Datei gespeichert. Wir könnten also her gehen und jedes Mal wenn eine Übersetzung ansteht, das gesamte Wörterbuch vom Dateisystem einlesen, nach dem gesuchten Wort suchen und es übersetzen. Dass das weder besonders effektiv noch sehr schnell sein wird dürfte jedem klar sein.

Also machen wir folgendes: in der *jspInit()* Methode laden wir das gesamte Wörterbuch und speichern es in einer Hashmap. Diese Hashmap fügen wir dann einfach dem *application* Objekt hinzu. Dieser Vorgang passiert nur einmal, wenn die JSP Seite das erste Mal aufgerufen wird.

```
<%!
    public void jspInit() {
        String sPath = getServletContext().getRealPath("/") +
                                              "dictionary_en_de.txt";
        Map<String,String> dictionary = loadDictionaryFromFile(sPath);
        ServletContext application =
            getServletConfig().getServletContext();
        application.setAttribute("Dictionary", dictionary);
    }

    private Map<String,String> loadDictionaryFromFile(String fileName)
    {
        // look at example from first semester...
    }
%>
<%
    Map<String,String> dictionary =
        (Map<String,String>)application.getAttribute("Dictionary");

    String english = request.getParameter("englishWord");
    String german = dictionary.get(english);
    out.println("The German translation of '" + english +
                "' is '" + german + "'");
%>
```

Danach holen wir uns einfach die *dictionary* Hashmap aus dem *application* Objekt, suchen nach dem gewünschten Wort und seiner Übersetzung und geben diese aus.

SEP: Wir sollten vermeiden Instanzvariablen in JSP Seiten zu verwenden.

SEP: Wir sollte das Dateisystem so wenig wie möglich verwenden.

Review

Was haben wir in diesem Kapitel gelernt? Wir haben auf einmal ein Kurzzeitgedächtnis in den Formen

- session: für individuelle Informationen die nur einen Nutzer betreffen, Stichwort Warenkorb und
- application: für globale Informationen die von allen Nutzern geteilt werden, Stichwort Guestbook.

Dass wir damit sehr interessante Projekte realisieren können werden wir gleich sehen.

Projekte

Wir haben jetzt das nötige Handwerkszeug, dass wir viele Beispiele aus dem ersten Semester "webifizieren" können. Wir werden dabei sehen, dass wir sehr viel vom Code aus dem ersten Semester wiederverwenden können, man nennt das auch "Re-Use".

NumberGuess

Ähnlich wie beim Visitor Counter Beispiel, haben wir beim NumberGuess Beispiel aus dem letzten Kapitel, das Hidden-Tag verwendet. Das ist natürlich nicht so schlau, denn jeder der weiß wie das mit dem "View Source" funktioniert, kann die Zahl sehr schnell erraten. Deswegen wollen wir in unserem ersten Projekt das NumberGuess Beispiel so abändern, dass es die zu ratende Zahl im *session* Objekt speichert und nicht im Hidden-Tag, ganz in Analogie zu dem Visitor Counter Beispiel.

Hangman

Wir haben Hangman ja bereits im ersten Semester programmiert, und es wäre natürlich Unsinn jetzt wieder von vorne zu beginnen. Anstelle nehmen wir einfach die Klasse Hangman aus dem ersten Semester und machen ein paar kleine Modifikationen:

- zuerst einmal entfernen wir das "extends ConsoleProgram",
- dann verschieben wir die Klasse in ein Paket, z.B. "de.variationenzumthema.internet",
- wir nehmen den Inhalt der run() Methode und kopieren ihn in den Konstuktor
- und wir machen Methoden auf die wir von außen zugreifen müssen public.

Nach diesen Vorbereitungen, überlegen wir uns kurz wie die Webanwendung funktionieren soll: erst einmal, soll jeder Nutzer sein eigenes Hangman Spiel bekommen. Das bedeutet, dass wir mit dem *session* Objekt arbeiten müssen. Beim ersten Laden der Seite, muss Hangman initialisiert werden, also ein neues Wort soll erzeugt werden. Das passiert im Konstuktor der Hangman Klasse. Die fügen wir dann zum *session* Objekt hinzu.

```
<%@page import="de.variationenzumthema.internet.Hangman"%>
<%
    // get handle to Hangman object
    Hangman hangman = (Hangman)session.getAttribute("Hangman");
    if ( (hangman == null) ||
         (request.getParameter("reset") != null) ) {
        hangman = new Hangman();
        session.setAttribute("Hangman", hangman);
    }

    // check if a guess was made
    String guess = request.getParameter("guess");
    if ( (guess != null) && (guess.length() == 1) ) {
        hangman.checkGuess(guess);
        if ( hangman.areWeDone() ) {
            out.println("Congratulations!");
        }
    }
%>
<!DOCTYPE html>
<html>
    <body>
        <h1>Hangman</h1>
        <p>The word looks like this:
            <strong><%= hangman.getHintWord() %></strong>.<br/>
            You used <%= hangman.getCounter() %> guesses.
        </p>
        <form action="hangman.jsp" method="POST" >
        Your guess:
            <input type="text" name="guess" />
            <input type="submit" value="Guess" />
        </form>
        (<a href="hangman.jsp?reset=true">Reset</a>)
    </body>
</html>
```

Wir haben hier auch noch gleich eine "Reset" Funktion mit eingebaut, falls das Wort zu schwer war.

TicTacToe

TicTacToe ist ein anderes sehr schönes Beispiel, wie wir ein Programm aus dem ersten Semester webifizieren können. In unserem TicTacToe soll Mensch gegen Maschine spielen. Die Maschine fängt an mit dem ersten Zug. Da wir das Problem ja schon im ersten Semester gelöst haben, machen wir das Gleiche wie oben: wir nehmen den Code aus dem ersten Semester und folgen den Schritten von Hangman um aus dem GraphicsProgram eine allgemeine Klasse zu machen.

Natürlich haben wir in einer Webanwendung keine MouseEvents, da müssen wir uns was überlegen. Eine Lösung sind Links: z.B. könnten wir die Position x=2 und y=0 wie folgt in einem Link kodieren:

```
<a href='ticTacToe.jsp?posI=2&posJ=0'>_</a>
```

Das TicTacToe Feld selbst stellen wir dann als HTML Tabelle dar. Und die Logik haben wir uns ja im ersten Semester schon ausgedacht.

```
<%@page import="de.variationenzumthema.internet.TicTacToeLogic"%>
<%
    // get handle to TicTacToe object
    TicTacToeLogic ttt =
        (TicTacToeLogic)session.getAttribute("TicTacToe");
    if ( (ttt == null) || (request.getParameter("reset") != null) ) {
        ttt = new TicTacToeLogic();
        session.setAttribute("TicTacToe", ttt);
    }

    // check if human made a move
    if ( request.getParameter("posI") != null ) {
        int posI = Integer.parseInt(request.getParameter("posI"));
        int posJ = Integer.parseInt(request.getParameter("posJ"));
        ttt.setNewPosition(posI,posJ);
    }

%>
<!DOCTYPE html>
<html>
    <body>
        <h1>TicTacToe</h1>
<table border="1">
<%
    for (int i = 0; i < 3; i++) {
        out.print("<tr>");
        for (int j = 0; j < 3; j++) {
            out.print("<td style='width:20px;text-align:center'>");
            if ( ttt.getBoardAt(i, j) == '_') {
                out.print(
                    "<a href='ticTacToe.jsp?posI="+i+"&posJ="+j+"'>");
            }
            out.print( ttt.getBoardAt(i, j) );
            if ( ttt.getBoardAt(i, j) == '_') {
                out.print("</a>");
            }
            out.print("</td>");
        }
        out.print("</tr>");
    }

    // check for game over:
    if ( ttt.isGameOver() ) {
        out.println("Game over!");
    }
%>
</table>
    (<a href="ticTacToe.jsp?reset=true">Reset</a>)
    </body>
</html>
```

Wie bei Hangman, instanziieren wir beim ersten Aufrufen der Seite ein Objekt der Klasse *TicTacToeLogic*. Dieses wird dann einfach im *session* Objekt gespeichert. Was ein klein bisschen Arbeit macht, ist die Tabelle zu generieren und die Links richtig zu setzen.

Adventure

Machen wir weiter mit dem Adventure Spiel aus dem ersten Semester. Inzwischen kennen wir den Drill: wir nehmen wieder unsere Adventure ConsoleProgram Klasse aus dem ersten Semester und machen daraus eine ganz normale Klasse mit Konstruktor. Etwas müssen wir allerdings beachten: die Daten für das Adventure Spiel kommen ja aus einer Datei, und den Pfad zu dieser Datei (die ja irgendwo auf dem Server liegt) müssen wir dem Konstruktor übergeben:

```
public class Adventure {
    public Adventure(String filePath) {
        loadWorld(filePath);
        currentRoom = "kitchen";
    }
    ...
}
```

Die Klasse wird wie immer beim ersten Laden der JSP Seite instanziiert.

```
<%
    Adventure advntr = (Adventure) session.getAttribute("Adventure");
    if (advntr == null) {
        String filePath = config.getServletContext().getRealPath("/")
                        + "adventure.txt";
        advntr = new Adventure(filePath);
        session.setAttribute("Adventure", advntr);
    }
%>
```

Wir beginnen unser Adventure in der Küche. Von da können wir in eine Liste von Räumen, die wir mittels *getAvailableRooms()* erhalten. Wir könnten diese Liste einfach aufzählen, viel besser ist es hier aber das HTML Select-Tag zu verwenden. Der Vorteil des Select-Tags ist, dass der Nutzer eigentlich keine falschen Eingaben machen kann.

```
<form action="adventureLogic.jsp" method="POST" >
    You can go to:
    <select name="room">
        <%
            for (String room : advntr.getAvailableRooms()) {
                out.println("<option value='" + room + "'>" +
                            room + "</option>");
            }
        %>
    </select>
    <input type="submit" value="Go" />
</form>
```

Die Logik um zum nächsten Raum zu gelangen haben wir in die *adventureLogic.jsp* Datei ausgelagert:

```
<%@page import="de.variationenzumthema.internet.Adventure"%>
<%
    if (request.getParameter("reset") != null) {
        session.removeAttribute("Adventure");
    } else {
        String room = request.getParameter("room");
        Adventure advntr =
            (Adventure) session.getAttribute("Adventure");
        advntr.setCurrentRoom(room);
    }
    response.sendRedirect("adventure.jsp");
%>
```

Das ist zwar nicht nötig, macht unseren Code aber viel lesbarer und wartbarer.

Sowohl in Hangman als auch TicTacToe verwendeten wir nur eine Datei für Logik und für View. Aber eigentlich sollte man die beiden trennen, das ist was wir hier gemacht haben: der View ist in *adventure.jsp* und die Logik in *adventureLogic.jsp*. Später wird uns das in Richtung Model-View-Controller Pattern führen, aus unserer Logik wird der Controller.

SEP: Je weniger Optionen ein Nutzer hat, desto weniger kann schief gehen.

Chat

Bisher haben wir nur mit dem *session* Objekt gearbeitet, es wird Zeit dass wir auch ein Beispiel mit dem *application* Objekt machen. Ein Chat in dem mehrere Menschen miteinander chatten können ist eine schöne Anwendung für das *application* Objekt. Wir können das Formular aus dem TwoPlayer Chat aus Kapitel eins verwenden, wir brauchen die Seiten *chat.jsp* und *conversation.jsp*.

In der *chatLogic.jsp* Seite nehmen wir die einzelnen Messages und fügen sie zur Liste der Konversation hinzu. Die Konversations-Liste speichern wir im *application* Objekt. Damit wir zwischen den verschiedenen Nutzern unterscheiden können verwenden wir die letzten zwei Ziffern der *sessionId*, und fügen die vor die eigentliche Message.

```
<%
    String msg = request.getParameter("msg");
    if (msg != null) {
        Vector<String> conversation =
            (Vector<String>) application.getAttribute("Chat");
        if (conversation == null) {
            conversation = new Vector<String>();
            application.setAttribute("Chat", conversation);
        }
        String id = session.getId();
        id = id.substring(id.length()-2);
        conversation.add(id + ": " + msg);
    }
    response.sendRedirect("chat.jsp");
%>
```

Kommen wir zu der Geschichte mit dem IFrame-Tag: das Problem mit dem Web ist, dass es nach dem Poll-Prinzip funktioniert. Soll heißen, der Browser muss beim Server nachfragen ob sich irgendetwas geändert hat. Der Server kann nicht von sich aus irgendwelche Daten an den Browser schicken (das geht erst mit HTML5's WebSockets). In unserem Beispiel bedeutet das, dass wir nicht mitbekommen, wenn ein anderer Nutzer eine neue Message gepostet hat. Wir müssten also eigentlich einmal alle paar Sekunden auf den Reload Knopf drücken um festzustellen, ob irgendjemand was Neues gepostet hat.

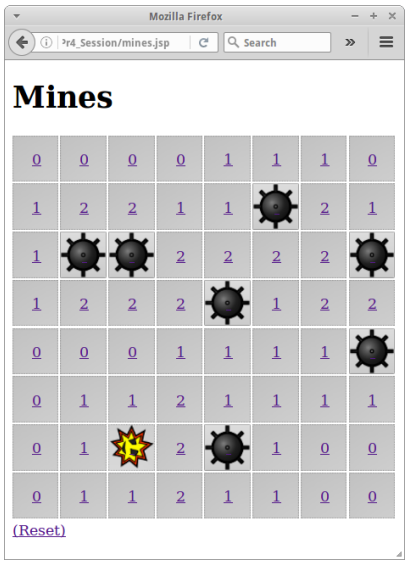
Das können wir aber auch automatisch machen, mit Hilfe des Refresh Meta-Tags. Wir schreiben also eine Seite *chatConversation.jsp*, die nichts anderes macht, als sich selbst alle 5 Sekunden neu zu laden. Alles was die Seite macht, ist einfach den Inhalt der Konversation zu listen. Und hier kommt der IFrame-Tag ganz gelegen, er erlaubt es uns nämlich eine andere Seite in unsere Hauptseite einzubetten.

```
<%
    Vector<String> conversation =
        (Vector<String>)application.getAttribute("Chat");
    if ( conversation == null ) {
        conversation = new Vector<String>();
        application.setAttribute("Chat", conversation);
    }
%>
```

```
<html>
    <head>
        <meta http-equiv="refresh" content="5" />
    </head>
    <body>
<%
    for (int i=conversation.size()-1; i>=0; i--) {
        out.println( conversation.get(i) +"<br/>" );
    }
%>
    </body>
</html>
```

Die Lösung funktioniert. Allerdings, und das sollten wir nicht aus den Augen verlieren, generieren wir mit dem Refresh-Tag eine konstante Hintergrundlast auf unserem Server. Alle fünf Sekunden kommt von jedem Browser der an dem Chat teilnimmt ein Request. Bei vielen Benutzern kann das zu einem Problem werden. Eleganter wäre wenn man das Ganze mit HTML5's WebSockets machen würde. Aber erstens benötigt man JavaScript dazu, und zweitens ist das etwas komplizierter.

SEP: Man sollte regelmäßige Refreshs vermeiden, da sie eine unnötige Last auf dem Server erzeugen.



Mines

Bisher waren alle unsere Beispiele eher textlastig. Wie wäre es mit einem kleinen Graphikbeispiel? Da wir ja weder JavaScript, Applets oder geschweige denn Flash verwenden wollen, werden wir auf Animationen verzichten müssen. Aber ein Spiel wie Mines aus dem ersten Semester oder Kartenspiele sind kein Problem.

Wir nehmen den Mines Code aus dem ersten Semester und machen aus dem GraphicsProgram wieder eine allgemeine Klasse, wie wir das auch bei Hangman gemacht haben. Die Bilder laden wir auf den Server hoch, und wir verwenden das img-Tag und eine HTML Tabelle um das Spielfeld grafisch im Browser darzustellen. Inzwischen dürfte das schon fast zur Routine geworden sein:

```
<%
    MinesClone mc =
        (MinesClone)session.getAttribute("MinesClone");
    if ( mc == null ) {
        mc = new MinesClone();
        session.setAttribute("MinesClone", mc);
    }
%>
<!DOCTYPE html>
<html>
    <body>
        <h1>Mines</h1>
        <%= mc.drawWholeField() %>
        <a href="minesLogic.jsp?reset=true">(Reset)</a>
    </body>
</html>
```

Das Zeichnen des Spielfeldes überlassen wir der MinesClone Klasse, das macht unser HTML übersichtlicher, und löst auch die Kopplung, soll heißen wir müssen weniger Methoden der MinesClone Klasse public machen. Die *drawWholeField()* Methode sieht wie folgt aus:

```
public String drawWholeField() {
    String html = "<table style='border-spacing: 0;" +
                  "border-collapse: collapse;'>";
    for (int i = 0; i < FIELD_SIZE; i++) {
        html += "<tr>";
        for (int j = 0; j < FIELD_SIZE; j++) {
            html += drawOneTile(i, j);
        }
        html += "</tr>";
    }
    html += "</table>";
    return html;
}
```

Bleibt nur noch die *minesLogic.jsp* Seite, und die simuliert eigentlich nur den MouseClick, ähnlich wie bei TicTacToe:

```
<%@page import="de.variationenzumthema.internet.MinesClone"%>
<%
    if ( request.getParameter("reset") != null ) {
        session.removeAttribute("MinesClone");
    } else {
        int i = Integer.parseInt( request.getParameter("i") );
        int j = Integer.parseInt( request.getParameter("j") );
        MinesClone mc =
            (MinesClone)session.getAttribute("MinesClone");
        mc.mouseClicked(i, j);
    }
    response.sendRedirect("mines.jsp");
%>
```

Eigentlich überraschend einfach. Falls wir uns jetzt inspiriert fühlen, können wir auch unserem TicTacToe Spiel einen grafischen Facelift verpassen, und natürlich wartet Schiffeversenken.

Messenger

Bei Messanger handelt es sich um eine kleine Anwendung in der sich verschiedene Nutzer Messages senden können, ähnlich zu Email. Die Details zu dieser Anwendung haben wir ja bereits im ersten Kapitel beschrieben und auch das UI haben wir dort schon umgesetzt. Hier wollen wir uns um die Datenhaltung und Businesslogik kümmern.

Die Login Seite ist reine UI, sie sendet lediglich den Alias des Nutzers an die Seite *messengerHome.jsp*.

Die zentrale Frage die sich stellt bevor wir weitermachen können, verwendet man das *session* oder das *application* Objekt? Oder vielleicht beide? Da wir Messages zwischen verschiedenen Personen hin und her schicken wollen, ist klar das wir etwas brauchen, auf das alle Nutzer zugreifen können. Deswegen ist auf jeden Fall das *application* Objekt notwendig. In ihm speichern wir eine Map, die den Nutzer-Alias als Key hat und die Messages für diesen Nutzer als Value:

```
Map<String, List<String>> users =
    (Map<String, List<String>>) application.getAttribute("Messenger");
```

Wie üblich müssen wir beim allerersten Aufruf dafür sorgen, dass die Map initialisiert wird.

Die nächste Frage die sich stellt, handelt es sich um einen neuen Nutzer, oder hat sich der Nutzer schon mal bei uns angemeldet. Dies können wir einfach mittels

```
if (!users.containsKey(alias)) {
    List<String> msgs = new ArrayList<String>();
    msgs.add("Welcome to Messenger!" + "; System");
    users.put(alias, msgs);
}
```

klären. Gibt es den Nutzer noch nicht, müssten wir eine neue Liste für Messages anlegen.

Danach können wir dem Nutzer das Formular zum Versenden von Messages zeigen:

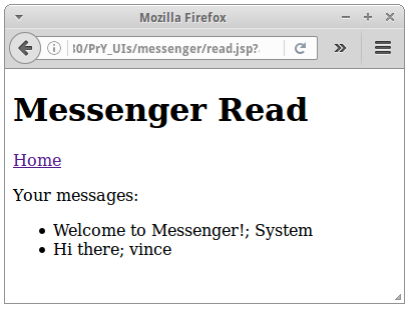
```
<html>
    <body>
        <h1>Messenger Home</h1>
        <p>
          <a href='messengerRead.jsp?alias=<%= alias %>'>
            Read my messages.
          </a>
        </p>
        <form action="messengerSend.jsp" method="GET">
            Enter message to send:
            <textarea name="message" rows="2" cols="40"></textarea>
            <input type="hidden" name="senderId"
                                 value="<%= alias %>"/>
            <br/>
            to user
            <select name="receiverId">
                <%
                    for (String name : users.keySet()) {
                        out.println("<option value='" + name + "'>" +
                                    name + "</option>");
                    }
                %>
            </select>
            <input type="submit" value="Send"/>
        </form>
    </body>
</html>
```

Als kleinen Bonus können wir dem Nutzer noch die möglichen Adressaten mit einem Select-Tag auflisten. Wir verwenden hier das Hidden-Tag um mitzuteilen wer der Sender der Nachricht ist. Alternativ, könnte man wie im Captcha Beispiel auch den Sender Alias im *session* Objekt speichern.

Die *messengerSend.jsp* Seite holt sich die Parameter und fügt die neue Message in der *users*-Map für den Empfänger ein:

```
<%
    ... get parameter ...

    Map<String, List<String>> users =
     (Map<String, List<String>>)application.getAttribute("Messenger");
    List<String> msgs = users.get(receiverId);
    msgs.add(message + "; "+senderId);
%>
```

Die *messengerRead.jsp* Seite




```
<%
    String alias = request.getParameter("alias");
    Map<String, List<String>> users =
        (Map<String, List<String>>)
        application.getAttribute("Messenger");
    List<String> msgs = users.get(alias);
%>
<!DOCTYPE html>
<html>
    <body>
        <h1>Messenger Read</h1>
        <p><a href='messengerHome.jsp?alias=<%= alias %>'>Home</a></p>
        <p>Your messages:</p>
        <ul>
        <%
            for ( int i=msgs.size()-1; i>=0; i-- ) {
                out.println("<li>"+msgs.get(i) +"</li>");
            }
        %>
        </ul>
    </body>
</html>
```

listet einfach alle Messages die für einen bestimmten Nutzer bestimmt sind.

SI: Wir sollten den Messanger auf Sicherheitsprobleme hin untersuchen.

TwoPlayer

Nach dem Multiplayer-Chat wollen wir uns hier mit dem Two-Player Chat beschäftigen. Die genauen Anforderungen und die UI haben wir ja schon im ersten Kapitel abgehandelt. Kommen wir zum interessanten Teil, der Logik.

Um das Problem zu lösen benötigen wir sowohl das *application* als auch das *session* Objekt. Überlegen wir uns wie die Logik funktionieren soll: wenn der erste Spieler kommt, dann muss dieser erst einmal warten. Er bekommt eine *waitingId*, die der Einfachheit halber gleich seine *sessionId* ist. Deswegen brauchen wir das *session* Objekt. Und natürlich müssen wir uns irgendwo global (application) merken, dass jemand wartet. Außerdem sollte der erste Spieler gelegentlich nachfragen, ob denn schon ein anderer Spieler aufgetaucht ist, das machen wir wie üblich mit dem Refresh-Tag:

```
<%
    // check logic...
%>
<html>
    <head>
        <meta http-equiv="refresh" content="5" />
    </head>
    <body>
        <h1>TwoPlayer</h1>
        <p>Waiting for other player to join...</p>
    </body>
</html>
```

Kommt der zweite Spieler, dann kann der ja an unserem globalen Merker erkennen, dass schon jemand wartet, und wir können die beiden verbinden. Wir vergeben dann eine *pairId*, die auch wieder der Einfachheit halber einfach die beiden *sessionIds* verknüpft ist. Der zweite Spieler kennt jetzt seine *pairId*, dem ersten Spieler müssen wir sie noch mitteilen. Da diseer aber alle 5 Sekunden fragt, können wir sie ihm beim nächsten Fragen mitteilen.

Sobald beide die *pairId* haben ist es ganz einfach: wir fügen zum *application* Objekt eine HashMap namens *messageMap*, die als Key die *pairId* hat und als Wert eine Liste mit der Konversation zwischen den beiden. Jedes mal wenn einer was sagt, wird das einfach an die Liste angehängt. Der Ansatz ist dann genauso wie bei unserer Chat Applikation weiter oben.

Da die Logik doch etwas komplizierter ist, macht es Sinn diese in eine eigene Klasse auszulagern. Das hat mehrere Vorteile: unsere JSP Seiten werden nicht so messy und bleiben übersichtlicher. Wir können die Klasse *TwoPlayerLogic* auch in anderen Anwendungen leicht wiederverwenden. Und wir können die Klasse unabhängig testen, z.B. mit JUnit.

```
package de.variationenzumthema.internet;

public class TwoPlayerLogic {

    private String waitingId = null;
    private Map<String, String> pairMap;
    private Map<String, Vector<String>> messageMap;

    public TwoPlayerLogic() {
        pairMap = new HashMap<String, String>();
        messageMap = new HashMap<String, Vector<String>>();
    }

    public void sendMessage(String pairId, String msg) {
        Vector<String> msgs = messageMap.get(pairId);
        if (msgs != null) {
            msgs.add(msg);
        }
    }

    public Vector<String> receiveMessages(String pairId) {
        Vector<String> msgs = messageMap.get(pairId);
        return msgs;
    }

    public String findPartner(String mySessionId) {
        // check if I am paired already:
        String pairId = pairMap.get(mySessionId);
        if (pairId == null) {
            if ((waitingId != null) && (waitingId != mySessionId)) {
                // somebody is already waiting:
                pairId = waitingId + mySessionId;
                pairMap.put(waitingId, pairId);
                pairMap.put(mySessionId, pairId);
                waitingId = null;

                // add a first welcome message
                Vector<String> msgs = new Vector<String>();
                msgs.add("Welcome!");
                messageMap.put(pairId, msgs);
```

```
                } else {
                    // nobody is waiting:
                    pairId = null;
                    waitingId = mySessionId;
                }
            }
            return pairId;
        }

        public void removeFromPairMap(String pairId) {
            // check if I am paired already:
            if (pairId != null) {
                int len = pairId.length() / 2;
                String id1 = pairId.substring(0, len);
                String id2 = pairId.substring(len);
                pairMap.remove(id1);
                pairMap.remove(id2);
                messageMap.remove(pairId);
            }
        }
    }
```

Was noch ein bisschen Kopfzerbrechen bereitet ist, wie wir den Chat wieder beenden. Das ist subtil kompliziert, aber auch machbar.

Obwohl unser Beispiel sich auf Zwei-Spieler beschränkt, dürfte es relativ klar sein, dass Gruppen-Chats oder Gruppen-Spiele ganz ähnlich gelöst werden können.

Wie sicher ist unsere Anwendung? Ein Hacker (oder die NSA) müsste an die *sessionId* gelangen, um einen Chat abzuhören. D.h. von der Sicherheit her ist die Anwendung so gut wie jede Online-Banking Anwendung, vorausgesetzt man verwendet HTTPS.

Eine Sache die uns evtl. noch stört, der ganz Datenverkehr läuft über unseren Server und produziert dort natürlich eine Last für die wir bezahlen müssen. Manchmal möchte man das, so wie bei WhatsApp. Wenn man aber nicht alles mit anhören möchte, was sich die Leute so erzählen, dann könnte man einen Peer-to-Peer Ansatz wählen, wie Skype das macht. Die Technologie dazu heißt WebRTC.

Research

Zu diesem Kapitel gibt es nicht ganz so viel zu erforschen. Aber die Themen "Session HIjacking" und auch wie funktioniert das mit den Sessions z.B. in Tomcat sind schon interessant.

Session Hijacking

Als erstes sollten wir nachlesen was "Session Hijacking" überhaupt ist. Dann sollten wir verstehen warum "Session Hijacking" mit HTTPS viel schwieriger ist. Lange haben sich auch große Website wie Facebook und eBay geweigert HTTPS zu verwenden (warum?). Erst mit dem Firefox Plugin Firesheep [1] wurde Session Hijacking aber auf einmal so einfach, dass fast alle Großen jetzt HTTPS verwenden.

Tomcat

Tomcat ist ein sehr populärer Servlet Engine. Auch GlassFish verwendet ihn, wir merken das nur nicht. Interessant wäre jetzt mal herauszufinden, ob Tomcat Cookies oder URL-Rewrite für sein Session Management verwendet. Und eine andere Frage wäre, kann man das vielleicht ändern?

Fragen

1. Wir sollten nicht zu große Datenmengen in Sessions speichern. Nehmen wir an wir wollen das Profilbild unserer User mit 1 MegaByte in der Session speichern. Wieviel Speicher (RAM) braucht unser Server wenn 1000 Leute pro Minute gerade den Server besuchen. Wichtig ist hier auch zu beachten, dass eine Session erst nach 15 Minuten Inaktivität wieder gelöscht wird! (15 GByte)

2. Session vs Application: Gehen Sie durch die Projekte im Anhang und entscheiden Sie ob für die Lösung des Projektes ein Session Objekt, ein Applikation Objekt, evtl. beide oder evtl. keines von beiden notwendig ist.

3. In einem der Assignments sollten Sie einen Dictionary Service implementieren. Also ein deutsch-englisches Wörterbuch mittels JSP. Beschreiben Sie kurz wie Sie dabei vorgegangen sind.

4. Betrachten Sie die folgenden fünf Web-Anwendungen. Bei der Entwicklung der Web-Anwendung müssen Sie entscheiden, ob Sie ein “session” Objekt, ein “application” Objekt, vielleicht beides, vielleicht aber auch keines brauchen. Für die unten aufgeführten Anwendungen entscheiden Sie welche nötig sind und begründen Sie Ihre Entscheidung.
   - Calendar: Persönlicher Terminkalender.
   - Roulette: Einfache Version von Roulette in der ein Spieler gegen den Computer spielt.
   - Personal Phone Book: Ein Service um seine persönlichen Telefonnummern zu verwalten.
   - Corporate Phone Book: Das Corporate Phone Book enthält die Telefonnummern aller Mitarbeiter einer Firma. Ist für alle Mitarbeiter zugänglich.
   - BattleShip: Auch bekannt als "Schiffeversenken", hier spielen zwei Spieler gegeneinander.

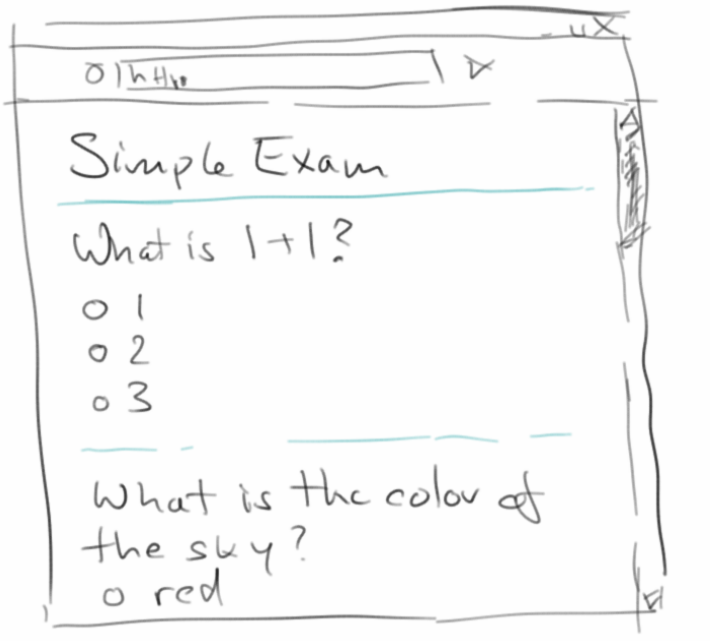
5. Betrachten Sie die folgenden sechs Web-Anwendungen. Bei der Entwicklung der Web-Anwendung müssen Sie entscheiden, ob Sie ein “session” Objekt, ein “application” Objekt, vielleicht beides, vielleicht aber auch keines brauchen. Für die unten aufgeführten Anwendungen entscheiden Sie welche nötig sind und begründen Sie Ihre Entscheidung.
   - Doodle: Doodle ist eine Web Anwendung, die es ermöglicht freie Zeitfenster für gemeinsame Treffen zu finden.
   - ToDo: ToDo hat ein einfaches Textfeld in dem ein Benutzer eine ToDo-Liste eintragen kann. Jeder Benutzer sollten seine eigene ToDo-Liste sehen.
   - Chat: Chat implementiert einen privaten Chat zwischen zwei Personen. TicTacToe: Schreiben Sie eine TicTacToe Web-App für einen Spieler, also Spieler gegen Computer.
   - NumberGuess: Der Computer wählt eine Zufallszahl zwischen 0 und 100. Der Benutzer soll dann erraten welche Zahl das war, und erhält Feedback ob diese kleiner, größer oder gleich der Zufallszahl war, bis der Benutzer die Zahl richtig hat.
   - Dictionary: Dictionary besteht aus einem HTML Formular in dem der Benutzer ein Wort eingibt und das dann übersetzt wird.

6. JSP hat 9 vordefinierte Objekte. Nennen Sie vier davon und erklären Sie wofür diese gut sind.

Referenzen

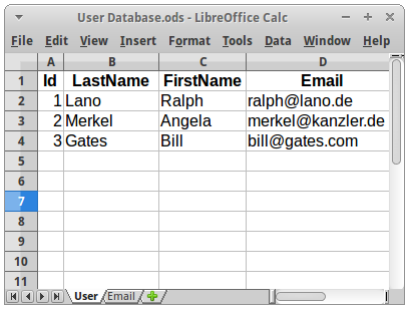
[1] Firesheep, codebutler.github.io/firesheep/

Database



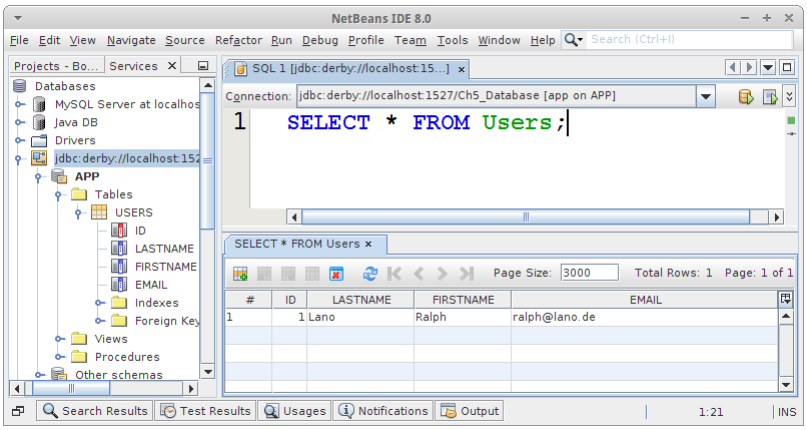
Fast hinter jeder Webanwendung steckt eine Datenbank. Datenbanken sind unser Langzeit-Gedächtnis. Deswegen werden wir in diesem Kapitel Webanwendungen schreiben, die ihre Daten in Datenbanken ablegen. Wir werden allerdings relative wenig mit Datenbanken direkt zu tun haben, da wir nach einer kurzen Einführung eigentlich nur Object-Relational-Mapping verwenden werden. Dies erlaubt es uns, uns auf das Wesentliche zu konzentrieren: die Webanwendung.

Database



Obwohl viele Leute höllischen Respekt vor Datenbanken haben, sind die eigentlich gar nicht so schwer. Für uns ist eine Datenbank erst einmal ein paar Excel Tabellen. Beginnen wir mit einer Tabelle für Nutzer unserer Webanwendung, wir nennen sie mal "User". Ein User hat einen Nachnamen, einen Vornamen und eine Email Adresse. In der Tabelle werden daraus Spalten (Columns). Die einzelnen Einträge für die Leute sind dann die Reihen (Rows). Bei Datenbank Tabellen fügt man dann meist noch eine Spalte für die "Id" ein: eine Id ist einfach ein Zähler, der bei 1 anfängt, und für jeden neuen Eintrag um eins erhöht wird. Damit hat jeder Eintrag eine eigene Id, und es kann nie doppelte Ids geben. Was aber passieren könnte, dass es zwei Leute mit dem gleichen Namen gibt. Gibt's ja auch in echt.

JavaDB



Das schöne an Netbeans ist, dass es mit einer einfachen Datenbank kommt, der JavaDB, manchmal auch Derby genannt. JavaDB ist eine OpenSource Datenbank, in Java geschrieben, die für unsere Zwecke vollauf genügt. Sie ist ziemlich schnell, da sie alles im RAM macht, das ist aber auch ihr größter Nachteil, man kann nicht so viele Daten in ihr speichern, also ein paar hundert Megabytes Maximum. Da wir gar nicht soviel Daten haben, ist das kein Problem.

Wenn man eine neue Datenbank anlegen will, klickt man einfach auf JavaDB mit der rechten Maustaste und sagt "Create Database". Danach verbindet man sich mit ihr in dem man einfach auf die neue Verbindung doppel-klickt. Will man dann SQL Kommandos ausführen (und wir wollen das), dann macht man wieder einen Rechtsklick und sagt "Execute Command", das öffnet dann den SQL Query Editor, so wie in dem Bild rechts.

SQL

Die "Structured Query Language", kurz SQL, ist die Sprache die die meisten Datenbanken sprechen. Es ist eine etwas ältere Sprache (so wie Altgriechisch, sieht man daran, dass alles groß geschrieben ist und die Befehle immer weniger als acht Buchstaben haben), aber sie funktioniert immer noch recht gut.

Als erstes ist es immer eine gute Idee evtl. existierende Tabellen zu löschen. Das geht mit

```
DROP TABLE Users;
```

ganz einfach. Sollte man natürlich nur machen, wenn da keine wichtigen Daten drin waren, die sind nämlich sonst futsch.

Danach legen wir eine neue Tabelle an. Hierfür gibt es das "CREATE TABLE" Kommando:

```
CREATE TABLE Users (
    Id INTEGER not null,
    LastName VARCHAR(255) not null,
    FirstName VARCHAR(255),
    Email VARCHAR(255),
    PRIMARY KEY (Id)
);
```

Wir sagen also, dass die Tabelle "Users" heißen soll, dass es eine Spalte Id geben soll die eine Ganzzahl sein soll und die immer einen Wert haben muss (not null). Dann deklarieren wir noch die anderen Spalten. Dabei heißt "VARCHAR(255)" soviel wie ein String der Länge 255 auf Altgriechisch. Am Ende sagen wir noch, dass "Id" der Primary Key sein soll.

Nachdem wir die Tabelle angelegt haben, wollen wir auch ein paar Daten einfügen, und das geht folgendermaßen:

```
INSERT INTO Users (Id, LastName, FirstName, Email)
    VALUES (1,'Lano','Ralph','ralph@lano.de');
```

Ganz wichtig sind die Apostrophen, auch *Single Quotes* genannt. Das sind also die geraden Striche die von oben nach unten gehen, und sich weder leicht nach links, noch leicht nach rechts lehnen.

Und schließlich wollen wir mal schauen was inzwischen in unserer Tabelle alles drin steht, und das geht mit:

```
SELECT * FROM Users;
```

Es gibt dann noch ein DELETE und ein UPDATE Kommando, aber im Prinzip war's das schon.

ORM

Wir könnten jetzt im 20. Jahrhundert bleiben und fleißig unsere SQL Statements tippen, und die Welt wäre auch in Ordnung. Da wir aber schon seit längerem im 21. Jahrhundert sind werden wir das nicht tun. Stattdessen werden wir ORM machen, also Object-Relational-Mapping. Das hört sich kompliziert an, ist aber eigentlich ganz einfach.

Java ist eine objekt-orientierte Sprache, und wenn wir Java schreiben, dann denken wir immer in Objekten. Alle unsere Daten sind in Form von Objekten. Bisher war es so, wenn wir unseren Rechner ausgeschaltet haben, und unsere Objekte nicht in Dateien gespeichert hatten, dann waren sie weg. Was wir jetzt machen wollen, wir wollen unsere Objekte in einer Datenbank speichern. Und genau das macht ORM für uns.

POJOs

POJO steht für "Plain Old Java Object", also einfach ein Java Objekt. Nehmen wir unseren User und machen daraus ein POJO:

```
package de.variationenzumthema.internet;

public class User {
    private Long id;
    private String lastName;
    private String firstName;
    private String email;
}
```

Meistens haben POJOs noch ein oder zwei Konstuktoren und natürlich Getters und Setters. Eine *toString()* Methode schadet auch nichts. Kann man ja alles autogenerieren lassen, deswegen kümmern wir uns der Einfachheit halber nur um die Attribute (Instanzvariablen).

Wenn wir jetzt dieses Objekt in die Datenbank speichern wollen, müssen wir es mit *Annotationen* versehen. Annotationen sagen dem ORM Engine (in unserem Fall Hibernate [1]) wie er denn das POJO in der Datenbank speichern soll. Nehmen wir unser *User* POJO:

```
package de.variationenzumthema.internet;

@Entity
@Table(name = "Users")
public class User {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String lastName;

    private String firstName;

    private String email;

    // Hibernate needs a default constructor.
    public User() {
        super();
    }
}
```

Die erste Annotation, *@Entity*, sagt einfach dass die Klasse *User* in der Datenbank gespeichert werden soll. Die zweite Annotation, *@Table*, sagt wie die Datenbanktabelle heißen soll in der das POJO gespeichert wird. Normal wäre "User". Da aber "User" bei den meisten Datenbanken schon existiert, nennen wir sie "Users". "Loosers" geht auch, wäre aber problematisch wenn unsere Nutzer mal unseren Code zu sehen bekommen. Normalerweise sollte aber der Tabellenname identisch mit dem POJO Namen sein.

Danach kommen die Annotationen für die Attribute. Aus Attributen werden Spalten in der Tabelle. Wenn wir nichts sagen, dann macht Hibernate das automatisch. Wir können aber auch sagen, dass eine Spalte nicht leer sein darf mit *@Column(nullable = false)*. Und wir sollten immer eines der Attribute als Primary Key deklarieren, das machen wir mit der *@Id* Annotation. Wenn wir noch wollen, dass die Datenbank die Ids automatisch vergibt, dann fügen wir noch *@GeneratedValue* nach der @Id hinzu.

hibernate.cfg.xml

Nachdem wir unsere POJOs mit Annotationen versehen haben, müssen wir unseren ORM Engine konfigurieren. Wir verwenden Hibernate. Deswegen müssen wir einmal die nötigen Libraries zum Projekt hinzufügen (macht man meist gleich am Anfang wenn man das Projekt neu anlegt), und Hibernate konfiguieren in der *hibernate.cfg.xml* Datei:

```
< hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">
            org.hibernate.dialect.DerbyDialect</property>
        <property name="hibernate.connection.driver_class">
            org.apache.derby.jdbc.ClientDriver</property>
        <property name="hibernate.connection.url">
            jdbc:derby://localhost:1527/Ch5_Database</property>
        <property name="hibernate.connection.username">app</property>
        <property name="hibernate.connection.password">app</property>

        <!-- e.g. validate | update | create | create-drop -->
        <property name="hibernate.hbm2ddl.auto">create-drop</property>
        <property name="show_sql">true</property>

        <!-- Persistent classes -->
        <mapping class="de.variationenzumthema.internet.User"/>

    </session-factory>
</hibernate-configuration>
```

Der erste Abschnitt erklärt Hibernate wie es sich mit der Datenbank verbinden kann. Der zweite Teil sagt Hibernate, dass es bei jedem Neustart alle Tabellen in der Datenbank löschen und neu anlegen soll. So etwas sollte man natürlich nicht in einer Produktionsumgebung machen, denn sonst sind alle Daten weg. Beim Debuggen hilft es manchmal sich das SQL das Hibernate generiert anzeigen zu lassen, deswegen setzt man *show_sql* auf true. Und schließlich müssen wir Hibernate noch erzählen welches POJO denn in die Datenbank geschrieben werden soll. In unserem Fall ist es das *User* POJO.

User

Nach all der Vorarbeit, ist es jetzt so weit: wir wollen unser *User* POJO in der Datenbank speichern. Zunächst brauchen wir ein ganz einfaches HTML Formular für die Eingabe der User Daten:

```
<html>
    <body>
        <h1>User</h1>
        <p>Please enter new user information:</p>
        <form action="userLogic.jsp" method="GET">
            Last name:  <input type="text" name="lastName"/><br/>
            First name: <input type="text" name="firstName"/><br/>
            Email: <input type="text" name="email"/><br/>
            <input type="submit" value="Create new user"/>
        </form>
    </body>
</html>
```

In der *userLogic.jsp* speichern wir diese mit Hilfe von Hibernate in der Datenbank. Das geht über die Hibernate Session:

```
<%
    // write user to database
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String email = request.getParameter("email");
    if (lastName != null) {
        Session hibSession =
            HibernateUtil.getSessionFactory().openSession();
        hibSession.beginTransaction();
        User usr = new User(lastName,firstName,email);
        hibSession.merge(usr);
        hibSession.getTransaction().commit();
        hibSession.close();
    }
%>
```

Wir verwenden das *merge()* Kommando, was wie ein Update funktioniert: also falls es den Eintrag schon gibt wird er geändert, falls nicht wird er neu angelegt. Bei allen Operationen die etwas in der Datenbank verändern, müssen wir den ganzen Code immer zwischen einem *beginTransaction()* und einem *commit()* stecken, denn es könnte ja etwas schief gehen. Falls etwas schief geht, würde man einen *rollback()* machen, sehen wir später noch. Am Ende immer schön brav seine Sessions zu machen mit *close()*.

Wenn wir dann alle User auflisten wollen geht das folgendermaßen:

```
<%
    Session hibSession =
        HibernateUtil.getSessionFactory().openSession();

    Query q = hibSession.createQuery(
        "from User usr order by usr.lastName desc");
    Collection<User> allUsrs = q.list();

    for (User usr : allUsrs) {
        out.println(usr.toString()+"<br/>");
    }

    hibSession.close();
%>
```

Wir verwenden hier eine Query um die Datenbankabfrage zu erledigen. Die Querysprache ist HQL (Hbernate Query Language) die vom Syntax her sehr ähnlich zu SQL ist, später mehr dazu.

HibernateUtil

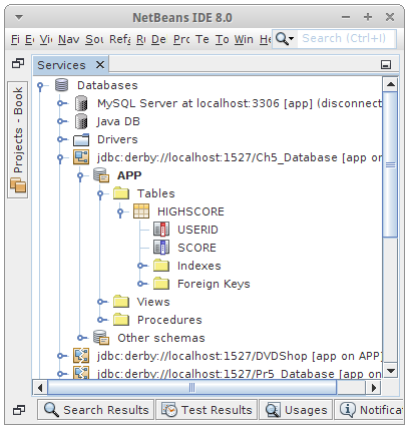
HibernateUtil ist eine Klasse, die man sich durch Netbeans ganz einfach autogenerieren lassen kann, aber man kann sie auch selbst von Hand anlegen:

```
package de.variationenzumthema.internet;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new
AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Die HibernateUtil ist ein Singleton Pattern, und gibt uns Zugriff auf die Hibernate SessionFactory. Die SessionFactory benötigen wir um neue Hibernate Sessions zur Datenbank aufzumachen. Haben wir ja oben schon verwendet.

HighScore

Beginnen wir mit ein paar Beispielen um uns mit ORM anzufreunden. Viele Spiele im Internet haben eine HighScore Liste. Die muss natürlich irgendwo gespeichert werden, und wir werden eine kleine Anwendung dafür schreiben. Zu einem HighScore gehört eine UserId, z.B. ein Alias oder eine Email Adresse, und ein Score. Damit lässt sich auch schon unser HighScore definieren:



```
@Entity
@Table(name = "HighScore")
public class HighScore {

    @Id
    private String userId;

    @Column(nullable = false)
    private int score;

    ... constructors, getters, setters, toString
}
```

Dieses Mal haben wir bei unserem Primary Key die *@GeneratedValue* Annotation weggelassen, was Sinn macht, da *wir* ja die UserId wählen wollen und nicht die Datenbank das machen soll. Es bedeutet aber auch, dass die UserId eindeutig sein muss, es kann also keine zwei Personen mit gleicher UserId geben. Was wir nicht vergessen dürfen, wir müssen noch einen Eintrag für unser neues POJO in der *hibernate.cfg.xml* Datei machen.

Zum Testen der Anwendung schreiben wir wieder ein Formular, wie rechts zu sehen. In diesem Fall wollen wir den GET Request verwenden, da wir später ja evtl. von unserem Spiel aus über einen ganz einfachen Aufruf wie etwa

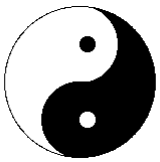
```
http://localhost:8080/.../highScoreLogic.jsp?userId=ralphlano&score=42
```

einen neuen Eintrag in die HighScore Liste generieren wollen (kann man mit der Java *URL* Klasse ganz einfach machen).

Das Schreiben der Daten in die Datenbank erfolgt wieder vollkommen analog wie in unserem User Beispiel. Wir tun das in der *highScoreLogic.jsp* Datei:

```
<%
    // write score to database
    String userId = request.getParameter("userId");
    String score = request.getParameter("score");
    if ((userId != null) && (score != null)) {
        Session hibSession =
            HibernateUtil.getSessionFactory().openSession();
        hibSession.beginTransaction();
        HighScore hs = new HighScore(userId, Integer.parseInt(score));
        hibSession.merge(hs);
        hibSession.getTransaction().commit();
        hibSession.close();
    }

    // show high scores:
%>
<!DOCTYPE html>
<html>
    <body>
        <h1>HighScore</h1>
        <ol>
<%
    Session hibSession =
        HibernateUtil.getSessionFactory().openSession();

    Query q = hibSession.createQuery (
        "from HighScore hs order by hs.score desc");
    Collection<HighScore> highScrs = q.list();
```

```
        for (HighScore hs : highScrs) {
            out.println(hs.toString()+"<br/>");
        }

        hibSession.close();
    %>
            </ol>
        </body>
    </html>
```



DAO

Wenn wir die beiden Dateien *userLogic.jsp* und *highScoreLogic.jsp* vergleichen sehen wir, dass diese fast identisch sind. Immer wenn etwas fast identisch ist, dann deutet das auf eine mögliche Vereinfachung hin. Im Falle von Datenbanken (und auch Hibernate) läuft das fast immer auf die sogenannten DAOs hinaus, die *Data Access Objects*.

Ein DAO sollte die standard Datenbank Operationen, Create, Read, Update und Delete, auch CRUD genannt, beherrschen. Für unsere HighScore Anwendung würde die folgende HighScoreDao Klasse genau das machen (wir haben der Lesbarkeit halber das Exception Handling weggelassen):

```
package de.variationenzumthema.internet;

public class HighScoreDao {

    private Session hibSession = null;

    public HighScoreDao() {
        this.hibSession =
            HibernateUtil.getSessionFactory().openSession();
    }

    public HighScore merge(HighScore entity) {
        hibSession.beginTransaction();
        HighScore e = (HighScore) hibSession.merge(entity);
        hibSession.getTransaction().commit();
        return e;
    }

    public void delete(HighScore entity) {
        hibSession.beginTransaction();
        hibSession.delete(entity);
        hibSession.getTransaction().commit();
    }

    public HighScore findById(String id) {
        return (HighScore) hibSession.get("HighScore", id);
    }

    @SuppressWarnings("unchecked")
    public List<HighScore> findAll() {
        return hibSession.createQuery(
            "Select e From HighScore e").list();
    }
}
```

Die Methode *merge()* ist sowohl für Create als auch für Update zuständig, die Methode *delete()* kümmert sich um das Delete, und mit *findById()* und *findAll()* wird das Read abgedeckt.

Unsere *highScoreLogic.jsp* Datei wird dann um einiges einfacher:

```
<%
    // write score to database
    HighScoreDao dao = new HighScoreDao();
    String userId = request.getParameter("userId");
    String score = request.getParameter("score");
    if ((userId != null) && (score != null)) {
        HighScore hs = new HighScore(userId, Integer.parseInt(score));
        dao.merge(hs);
    }

    // show high scores:
%>
<!DOCTYPE html>
<html>
    <body>
        <h1>HighScore</h1>
        <ol>
<%
    Collection<HighScore> highScrs = dao.findAll();

    for (HighScore hs : highScrs) {
        out.println(hs.toString()+"<br/>");
    }
%>
        </ol>
    </body>
</html>
```

Unser Datenbank Code reduziert sich auf drei Zeilen.

Mensa

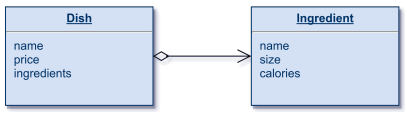
Als nächstes wollen wir uns Datenbanken widmen mit mehr als nur einer Tabelle. Wir nehmen dazu unser Mensa Beispiel aus dem ersten Semester:

"Die Mensa hat Gerichte und Zutaten. Ein Gericht hat einen Namen, einen Preis und eine Liste von Zutaten. Eine Zutat hat einen Namen, einen Preis und Kalorien."

Mensa POJOs

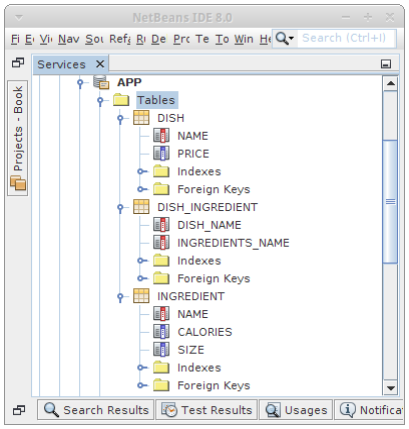
Wir beginnen mit unseren POJOs, in diesem Fall *Dish*:

```
@Entity
@Table(name = "Dish")
public class Dish {

    @Id
    private String name;

    @Column(nullable = false)
    private double price;

    @ OneToMany
    private Set<Ingredient> ingredients;

    ... constructors, getters, setters, toString
}
```



und *Ingredient*:

```
@Entity
@Table(name = "Ingredient")
public class Ingredient {

    @Id
    private String name;

    private String size;

    private double calories;

    ... constructors, getters, setters, toString
}
```



Das Einzige was hier neu ist, ist die Verknüpfung zwischen den beiden POJOs: eine *OneToMany* Beziehung. Soll heißen, ein Dish kann mehrere Ingredients haben. Wir können als Datenstruktur ein Set oder eine Liste verwenden. Wenn uns die Reihenfolge der Ingredients egal ist, nehmen wir ein Set, ansonsten eine Liste.

Interessant ist zu sehen was hier in der Datenbank passiert: Hibernate generiert jeweils eine Tabelle für die *Dish* und *Ingredient* POJOs. Zusätzlich wird aber noch eine Tabelle DISH_INGREDIENT angelegt, eine sogenannte Intersection Table, mit der die *OneToMany* Beziehung in der Datenbank abgebildet wird.

Mensa DAOs

Nachdem wir unsere POJOs definiert haben, und auch zur *hibernate.cfg.xml* Datei hinzugefügt haben, definieren wir die DAOs. In diesem Fall zwei, weil wir ja zwei POJOs haben. Wenn wir uns die beiden DAOs ansehen, dann sehen wir, dass die faktisch identisch sind. Sowas ruft natürlich förmlich nach Vererbung, und wenn man ein klein bisschen was von Generics versteht, denn sehen unsere beiden DAOs sehr einfach aus:

```
public class MensaDishDao extends GenericDao<String, Dish> {
    public MensaDishDao() {
        super();
    }
}
```

und

```
public class MensaIngredientDao extends GenericDao<String, Ingredient>
{
    public Mensa IngredientDao () {
        super();
    }
}
```

Generics

Wir haben bisher Generics eigentlich nur indirekt gesehen, wenn wir Listen, Maps oder Sets verwendet haben. Hier bietet sich jetzt die Gelegenheit Generics einmal selbst anzuwenden:

```
public class GenericDao<K extends Serializable, E> {

    private Class<E> entityClass;
    private Session hibSession = null;

    public GenericDao() {
        this.hibSession =
            HibernateUtil.getSessionFactory().openSession();
        // some reflection magic:
        ParameterizedType genericSuperclass =
            (ParameterizedType) getClass().getGenericSuperclass();
        this.entityClass =
            (Class<E>) genericSuperclass.getActualTypeArguments()[1];
    }

    public void delete(E entity) {
        hibSession.beginTransaction();
        hibSession.delete(entity);
        hibSession.getTransaction().commit();
    }

    public E merge(E entity) {
        hibSession.beginTransaction();
        E e = (E) hibSession.merge(entity);
        hibSession.getTransaction().commit();
        return e;
    }

    public E findById(K id) {
        return (E) hibSession.get(entityClass, id);
    }

    @SuppressWarnings("unchecked")
    public List<E> findAll() {
        return hibSession.createQuery ("Select e From " +
            entityClass.getCanonicalName() + " e").list();
    }

    protected Session getHibernate Session() {
        return hibSession;
    }
}
```

Obwohl man anfangs vielleicht etwas überwältigt ist, wenn man sich den Code lange genug ansieht und ihn mit der HighScoreDao vergleicht, beginnt man das Licht am Ende des Tunnels zu erblicken ("I have seen the light!").

Mensa Test

Etwas später werden wir die Mensa Web Anwendung komplett implementieren. Aber nur zum Testen wollen wir eine kleine JSP Seite schreiben, *mensa.jsp*, die ein paar Ingredients und Dishes anlegt, und dann alle auflistet, damit wir sehen, dass auch alles funktioniert.

Das Anlegen von Test Objekten macht man am besten in der DAO. Wir fügen zu jeder der beiden DAOs eine Methode *initDatabase()* hinzu. Einmal bei den Ingredients:

```
public class MensaIngredientDao extends
    GenericDao<String, Ingredient> {

    public Mensa IngredientDao () {
        super();
    }

    public void init Database () {
        save( new Ingredient("Egg", "large", 80) );
        save( new Ingredient("Butter", "1 table spoon", 100) );
        save( new Ingredient("Milk", "1 cup", 150) );
        save( new Ingredient("Flour", "100 g", 364) );
    }
}
```

und dann bei den Dishes:

```
public class MensaDishDao extends GenericDao<String, Dish> {

    public MensaDishDao() {
        super();
    }

    public void init Database ( Mensa IngredientDao daoIng) {
        Set<Ingredient> ingrds = new HashSet<Ingredient>();
        ingrds.add(daoIng.findById("Egg"));
        ingrds.add(daoIng.findById("Butter"));
        ingrds.add(daoIng.findById("Milk"));
        ingrds.add(daoIng.findById("Flour"));
        save(new Dish("Pancake", 2.50, ingrds));
    }
}
```

Mit diesen Modifikationen wird die *mensa.jsp* Seite relativ einfach:

```
<%!
    public void jspInit() {
        ServletContext application =
            getServletConfig().getServletContext();

        MensaIngredientDao daoIng = new MensaIngredientDao();
        daoIng.initDatabase();
        application.setAttribute("Mensa.IngredientDao", daoIng);

        MensaDishDao daoDish = new MensaDishDao();
        daoDish.initDatabase(daoIng);
        application.setAttribute("Mensa.DishDao", daoDish);
    }
%>
```

In der *jspInit()* Methode kreieren wir je ein *MensaIngredientDao* und ein *MensaDishDao* Objekt, initialisieren jede Tabelle mit Daten, und fügen die DAO-Objekte zum *application* Objekt hinzu. Das hat den Vorteil, dass wir später nicht immer wieder den Konstruktor neu aufrufen müssen, bedeutet aber auch ein gewisses Bottleneck, denn es existiert jeweils nur ein Objekt von jeder DAO.

Kommen wir zum zweiten Teil der *mensa.jsp* Seite:

```
<html>
    <body>
        <h2>Mensa</h2>
        <p>Available dishes:</p>
        <ul>
<%
    MensaDishDao daoDish =
        (MensaDishDao)application.getAttribute("Mensa.DishDao");
    List<Dish> dishs = daoDish.findAll();
    for (Dish di : dishs) {
        out.println("<li>"+di+"</li>");
    }
%>
        </ul>
        <p>Available ingredients:</p>
        <ul>
<%
    MensaIngredientDao daoIng =
(MensaIngredientDao)application.getAttribute("Mensa.IngredientDao");
    List<Ingredient> ings = daoIng.findAll();
    for (Ingredient ing : ings) {
        out.println("<li>"+ing+"</li>");
    }
%>
        </ul>
    </body>
</html>
```

Wir holen uns das jeweilige DAO Objekt, und listen einfach alle Dishes und Ingredients auf.

OneToOne, ManyToOne, OneToMany und ManyToMany

Was ORM ein klein wenig kompliziert macht sind die Beziehungen zwischen POJOs. Im Mensa Beispiel haben wir unsere erste *OneToMany* Beziehung gesehen. Es gibt aber noch andere Beziehungen. Um ein bisschen ein Gefühl für die verschiedenen X-To-Y Beziehungen zwischen Tabellen zu bekommen, betrachten wir das Beispiel mit Usern und ihren Emails. Nehmen wir an, es gibt eine Klasse User und eine Klasse Email:

```
public class User {

    @Id
    private String lastName;

    private Email email;
}
```

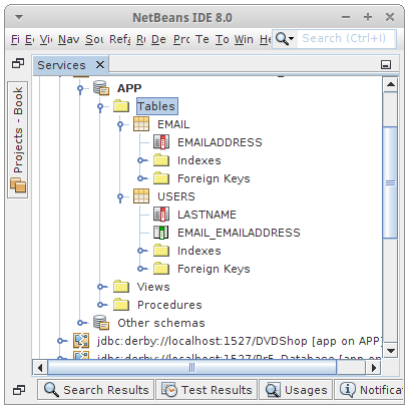
```
public class Email {

    @Id
    private String emailAddress;
}
```

Je nach Anwendung kann es die folgenden vier Szenarien geben:

- OneToOne
- ManyToOne
- OneToMany
- ManyToMany

Außerdem müssen wir noch zwischen uni-direktional und bi-direktional unterscheiden: im ersten Fall, kennt der User seine Email, die Email weiß aber nicht zu welchem User sie gehört. Im bi-direktional Fall, kennt der User seine Email, und die Email weiß zu welchem User sie gehört. Der uni-direktionale Fall ist immer einfacher und auch performanter. Nur wenn absolut nötig, sollte man den bi-direktionalen Fall verwenden. Wir werden fast ausschließlich uni-direktional arbeiten.

OneToOne



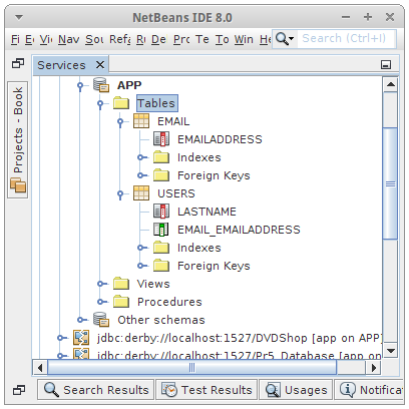
Die einfachste Beziehung ist die OneToOne Beziehung: Ein User hat eine Email.

In dieser Beziehung hat ein User eine Email. Wenn wir allerdings möchten, dass eine Email nicht zu zwei Usern gehören darf, müssen wir noch den Uniqueness Constraint angeben. Denn ohne ist es möglich, dass die gleiche Email zu zwei Usern gehört.

```
public class User {

    @Id
    private String lastName;

    @ OneToOne
    @JoinColumn(unique=true)
    private Email email;
}
```

```
public class Email {

    @Id
    private String emailAddress;
}
```

ManyToOne



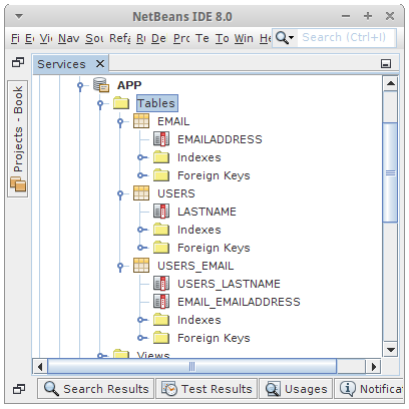
Die nächst einfachere Beziehung ist die ManyToOne Beziehung: Auch, ein User hat eine Email.

Wenn wir uns das Datenbank Schema ansehen, bemerken wir, dass es identisch zur OneToOne Beziehung ist. Wenigstens im uni-direktionalen Fall. Deswegen wird die ManyToOne Beziehung eigentlich nie alleine verwendet, sondern immer zusammen mit einer OneToMany Beziehung auf der anderen Seite.

```
public class User {

    @Id
    private String lastName;

    @ManyToOne
    private Email email;
}
```

```
public class Email {

    @Id
    private String emailAddress;
}
```

OneToMany

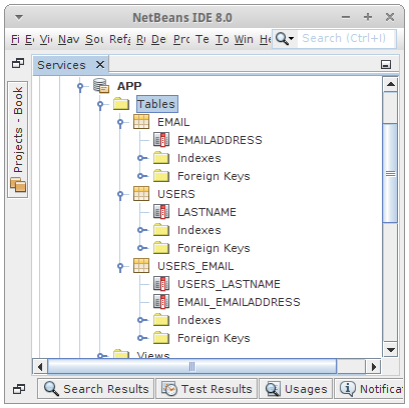


Das ist eigentlich die Beziehung die am häufigsten vorkommt: Ein User hat mehrere Emails.

In dieser Beziehung kann ein User mehrere Emails haben. Interessant ist allerdings, dass eine Email nicht zu zwei Usern gehören kann.

```
public class User {

    @Id
    private String lastName;

    @OneToMany
    private Set<Email> emails;
}
```

```
public class Email {

    @Id
    private String emailAddress;
}
```



ManyToMany

Schließlich kommen wir zur komplexesten Beziehung: Ein User hat mehrere Emails und eine Email kann zu mehreren Usern gehören.

In dieser Beziehung kann ein User mehrere Emails haben. Und eine Email kann auch zu mehreren Usern gehören.

```
public class User {

    @Id
    private String lastName;

    @ManyToMany
    private Set<Email> emails;
}
```

```
public class Email {

    @Id
    private String emailAddress;
}
```

Review

Was haben wir in diesem Kapitel gelernt? Wir haben

- eine einfache Datenbank kennengelernt,
- die Grundkommandos von SQL gesehen und das Akronym CRUD,
- unsere ersten Schritte mit ORM unternommen
- und die Unterschiede zwischen den verschiedenen X-To-Y Beziehungen.

Gleich werden wir einige Beispiele für unser neues Langzeitgedächtnis sehen.

Projekte

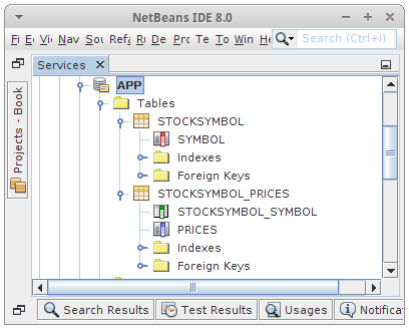
Jetzt wird es Ernst: wir können in Prinzip fast jede existierende Webanwendung nachbauen. Natürlich macht es Sinn sich langsam vorzuarbeiten. Machen wir hier.

Stocks

Im ersten Semester haben wir eine kleine Anwendung geschrieben um Aktienkursen anzuzeigen. Wir wollen das Beispiel jetzt webifizieren. Wichtig ist uns dabei, dass die Daten nicht aus der Datei kommen, sondern aus der Datenbank.

Für uns besteht ein Aktienkurs aus dem Aktienkürzel, oder Symbol, und den Kursen, einfach einer Liste von Double. Das Ganze fassen wir in dem POJO StockSymbol zusammen:

```
@Entity
@Table(name = "StockSymbol")
public class StockSymbol {

    @Id
    private String symbol;

    @ElementCollection
    @Column(name = "prices", nullable = false)
    private List<Double> prices;

    ... constructors, getters, setters, toString
}
```

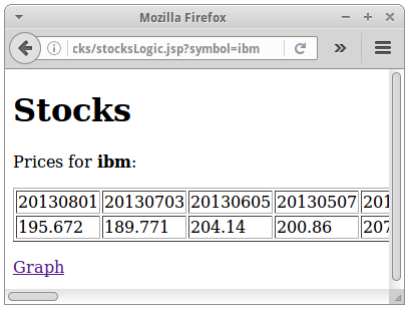


Was hier neu ist ist das Attribut *@ElementCollection*: es ist effektiv wie eine OneToMany Beziehung für primitive Datentypen. Wie das in der Datenbank abgebildet wird sehen wir rechts. Es gibt also eine Tabelle für die Symbole und eine für die Aktienkurse.

Nachdem unser POJO steht, sehen wir uns die DAO an. Wir machen nicht viel mit der Datenbank: wir initialisieren sie einmal, danach greifen wir nur lesend darauf zu, deswegen brauchen wir eigentlich nicht die GenericDao zu bemühen.

```
public class StockDao {

    private Session hibSession = null;

    public StockDao() {
        this.hibSession =
            HibernateUtil.getSessionFactory().openSession();
    }

    public StockSymbol getStockPrice(String symbol) {
        return (StockSymbol)hibSession.get(StockSymbol.class,symbol);
    }
```

```
public List<String> initStockPriceTable(String path) {
    List<String> dates = null;
    try {
        hibSession.beginTransaction();
        BufferedReader br =
            new BufferedReader(new FileReader(path));

        // first line contains dates:
        String line = br.readLine();
        dates = readDates(line);

        // other lines contain data:
        while (true) {
            line = br.readLine();
            if (line == null) {
                break;
            }
            StockSymbol entry = new StockSymbol(line);
            hibSession.merge(entry);
        }

        br.close();
        hibSession.getTransaction().commit();

    } catch (Exception e) {
        e.printStackTrace();
    }
    return dates;
}

private List<String> readDates(String line) {
    List<String> dates = new ArrayList<String>();
    String[] datesArray = line.split(",");
    for (int i = 1; i < datesArray.length; i++) {
        dates.add(datesArray[i]);
    }
    return dates;
}
```



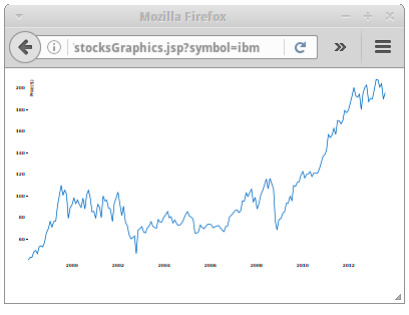
In der Methode *initStockPriceTable()* lesen wir die Aktienkurs aus der Datei und speichern sie in der Datenbank. Interessant ist dabei, dass wir nicht jedesmal einen Commit machen, sondern erst nachdem wir alle Daten gespeichert haben. Das ist ungefähr dreimal so schnell. Trotzdem dauert das Ganze ca. eine halbe Minute. Nachdem die Daten aber in der Datenbank sind, ist die Anwendung sehr schnell.

In der *stocksLogic.jsp* Seite initialisieren wir alles in der *jspInit()* Methode. Wir kreieren das Dao Objekt, rufen die *initStockPriceTable()* Methode auf, und fügen es dem *application* Objekt hinzu, damit wir später ganz einfach darauf zugreifen können. Über den Request Parameter erhalten wir das gewünschte Symbol, für das wir die Daten aus der Datenbank abfragen, und schließlich in einer HTML Tabelle anzeigen.

```
<%!
    public void jspInit() {
        String path = getServletContext().getRealPath("/") +
            "stocks/SP500_HistoricalStockDataMonthly.csv";
        ServletContext application =
            getServletConfig().getServletContext();
```

```
            // init database with data from file:
            StockDao dao = new StockDao();
            List<String> dates = dao.initStockPriceTable(path);
            application.setAttribute("Stocks.dao", dao);
            application.setAttribute("Stocks.dates", dates);
        }
%>
<%
    // query database for symbol
    String symbol = request.getParameter("symbol");
    StockDao dao = (StockDao) application.getAttribute("Stocks.dao");
    List<String> dates =
        (List<String>) application.getAttribute("Stocks.dates");
    StockSymbol sp = dao.getStockPrice(symbol);

    // show stock prices:
%>
<!DOCTYPE html>
<html>
    <body>
        <h1>Stocks</h1>
        <p>Prices for <strong><%= sp.getSymbol()%></strong>:</p>
        <table border="1">
            <tr>
<%
    for (int i = 0; i < dates.size(); i++) {
        out.println("<td>"+dates.get(i)+"</td>");
    }
%>
            </tr>
            <tr>
<%
    List<Double> prices = sp.getPrices();
    for (int i = 0; i < prices.size(); i++) {
        out.println("<td>"+prices.get(i)+"</td>");
    }
%>
            </tr>
        </table>
        <p><a href="stocksGraphics.jsp?symbol=<%= symbol %>">
            Graph</a></p>
    </body>
</html>
```

Challenge



Eigentlich sind wir mit unserer Hauptaufgabe, Aktienkurse in eine Datenbank zu schreiben und dann daraus zu lesen fertig. Nur die Darstellung als HTML Tabelle ist nicht besonders sexy. Mit ein klein wenig JavaScript und der JavaScript Library D3JS [2] ist es aber ein Leichtes die Aktienkurse auch grafisch darzustellen. Wir müssen lediglich die Kursdaten in eine Form bringen, die die D3JS Bibliothek versteht, in diesem Fall "TabSeperatedValues" auch TSV genannt. Das macht die Datei *stocksTSV.jsp*:

```
<%
    // query database for symbol
    String symbol = request.getParameter("symbol");
    StockDao dao = (StockDao) application.getAttribute("Stocks.dao");
    List<String> dates =
        (List<String>) application.getAttribute("Stocks.dates");
    StockSymbol sp = dao.getStockPrice(symbol);

    // from 20121011 to 11-Oct-12
    DateFormat formater1 = new SimpleDateFormat("yyyyMMdd");
    DateFormat formater2 = new SimpleDateFormat("dd-MMM-yy");

    // show stock prices in TSV form:
    out.println("date    close");
    List<Double> prices = sp.getPrices();
    for (int i = 0; i < dates.size(); i++) {
        Date dte = (Date)formater1.parse(dates.get(i));
        out.println(formater2.format(dte) +"    "+prices.get(i));
    }
%>
```

Diese Daten können dann mit Hilfe der JavaScript Datei *stocksGraphics.jsp* grafisch dargestellt werden. Der JavaScript Code stammt von Mike Bostock's Line Chart Beispiel [3].

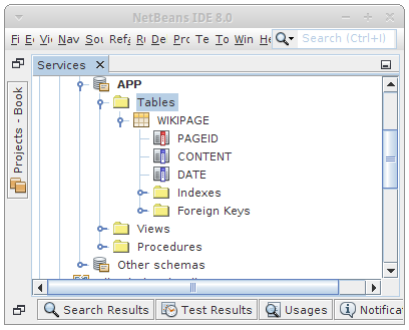
SI: Wenn wir versuchen die Datei *SP500_HistoricalStockDataMonthly.csv* direkt runter zu laden, werden wir feststellen, dass dies möglich ist. Die Frage ist ob wir das wollen?

Wiki

Wir sind jetzt soweit unser eigenes kleines Wiki zu schreiben. Die Anforderungen dazu und die UI haben wir ja bereits im ersten Kapitel beschrieben und umgesetzt. Was noch fehlt ist der Wiki-Markup. Wir lassen uns von Wikipedia's Markup inspirieren, beschränken uns aber auf folgenden:

- '=' für Überschriften (evtl auch '==' und '==='),
- '----' für eine horizontale Trennlinie,
- eine leere Zeile für eine Leerzeile,
- '*' für eine Aufzählungsliste,
- und natürlich [[...]] für Links.

POJOs



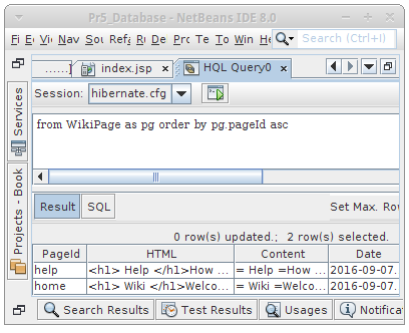
Kümmern wir uns als nächstes um das Datenmodel, die POJOs. In diesem Fall genügt ein POJO:

```
public class WikiPage {

    @Id
    private String pageId;

    @Column(length = 32000, nullable = false)
    private String content;

    @Column(nullable = false)
    private Date date;

    ... constructors, getters, setters, toString
}
```

Nichts besonders Kompliziertes. Auch die DAO ist eigentlich trivial:

```
public class WikiDao extends GenericDao<String, WikiPage> {

    public WikiDao() {
        super();
    }

    public List<WikiPage> search(String searchTerm) {
        List<WikiPage> pageList = null;
        searchTerm = Utility.escapeSql(searchTerm);
        Query q = getHibernateSession().createQuery (
                "from WikiPage as pg where lower(pg.content) "
                        + "like lower('%"+searchTerm+"%') "
                        + "order by pg.pageId asc"
             );
        pageList = (List<WikiPage>) q.list();
        return pageList;
    }

    /**
     * we want pages to be sorted, hence override default behavior
     */
    public List<WikiPage> findAll() {
        return getHibernateSession().createQuery(
            "from WikiPage as pg order by pg.pageId asc").list();
    }

    public void initDatabase() {
        String homePage
                = "= Wiki =\n"
                + "Welcome to 'Wiki'.\n"
                + "\n"
                + "Help can be found under [[help]].\n"
                + "----\n";
        merge(new WikiPage("home", homePage));
        String helpPage
                = "= Help =\n"
                + "How to use this simple Wiki.  "
                + "The following markup exists:\n"
                + "* '=' for heading (=, ==, === exist)\n"
                + "* '----' for horizontal line\n"
                + "* empty line for new line\n"
                + "* '*' for bulleted list\n"
                + "* \\[\\[...\\]\\] for links\n"
                + "That's it.";
        merge(new WikiPage("help", helpPage));
    }
}
```

In der *initDatabase()* Methode initialisieren wir wie üblich die Datenbank, in diesem Fall mit zwei Wikiseiten.



HQL

Interessanter sind allerdings die *findAll()* und die *search()* Methoden. In der *findAll()* Methode überschreiben wir das normale Verhalten der Elternklasse, wir wollen nämlich dass die Seiten in alphabetischer Reihenfolge gelistet werden. Das erreichen wir mit dem "order by" Argument, und "asc" steht für *ascending*, also aufsteigend.

Noch interessanter ist die *search()* Methode: die Sprache die wir hier verwenden ist die Hibernate Query Language (HQL), die vom Syntax her stark an SQL erinnert:

```
"from WikiPage as pg where lower(pg.content) "
+ "like lower('%"+searchTerm+"%') "
+ "order by pg.pageId asc"
```

Eigentlich ist die Query selbsterklärend.

In Netbeans gibt es einen HQL Editor, der es einem erlaubt HQL Queries zu testen. Man erreicht ihn, in dem auf die *hibernate.cfg.xml* Datei mit der rechten Maustaste klickt und dann "Run HQL Query" auswählt. Der HQL Editor hat allerdings zwei kleine Quirks: zum einen müssen alle POJOs einen Default Konstruktor haben. Und zum anderen, müssen wir bevor wir den HQL Editor aufrufen, in der *hibernate.cfg.xml* Datei das "create-drop" durch "validate" ersetzen. Aus irgendeinem Grund, werden beim Start des HQL Editor anscheinend alle Tabellen erst einmal gedropped. Aber ansonsten funktioniert der HQL Editor "like a charm".

Prepared Statements

Im vorletzten Kapitel haben wir ganz kurz das Wort "SQL Injection" verwendet als wir über Cross-Site Scripting (XSS) und *escapeXML()* gesprochen haben. Was ist "SQL Injection"? Nehmen wir an wir fragen einen Nutzer nach *userName* und *password*, und würden dann daraus folgendes SQL konstruieren:

```
String sql = "SELECT id FROM users WHERE username='"+userName+"' AND
    password='"+password+"'";
```

Wenn der Nutzer so etwas wie "ralph" und "123456" eingibt, dann funktioniert das auch super. Wenn aber ein Schlaumeier für *password* das folgende eingibt:

```
222222' OR '1' = '1
```

dann passiert etwas Interessantes: obwohl das Passwort falsch ist, wird er trotzdem eingeloggt. Das hat damit zu tun, dass die Datenbank folgendes sieht:

```
false AND false OR true
```

und das ergibt immer *true*. Und das ist "SQL Injection". Es ist überraschend wieviele echte Webseiten sich mit diesem Trick überrumpeln lassen.

Wir wollen das natürlich nicht, und deswegen verwenden wir zum einen immer die *escapeSql()* Methode, aber zum anderen sollten wir auch Prepared Statements verwenden. In Hibernate Speek würde das so aussehen:

```
createQuery("from User where userName=:userName")
    .setParameter("userName", userName).getSingleResult();
```

Im Beispiel für die *search()* Methode gerade haben wir das fälschlicherweise nicht so gemacht, aber jetzt können wir es ja richtig machen.

Regular Expressions

Kommen wir zum letzten interessanten Thema in diesem Projekt: reguläre Ausdrücke. Die Frage ist nämlich wie macht man aus dem Wiki Markup

```
= Wiki =
Welcome to 'Wiki'.

Help can be found under [[help]].
----
```

das folgende HTML?

```
<h1> Wiki </h1>
Welcome to 'Wiki'.
<br/>
Help can be found under <a href='index.jsp?page=help'>help</a>.
<hr/>
```

Wenn wir uns an unsere reguläre Ausdrücke aus dem zweiten Semester erinnern, dann ist das fast trivial. Was unser Leben einfach macht ist, dass die Methode *replaceAll()* der Klasse String reguläre Ausdrücke verwendet.

```
private String parseLine(String line) {
    String regex;

    // ' ' <br/>
    if (line.length() == 0) {
        line = "<br/>\n";
        return line;
    }

    // ---- <hr/>
    if (line.equals("----")) {
        line = "<hr/>\n";
        return line;
    }

    // escape html tags: '<' and '>'
    line = line.replaceAll("<", "&lt;");
    line = line.replaceAll(">", "&gt;");

    // * <ul><li>
    if (line.startsWith("*")) {
        line = "<ul><li>" + line.substring(1) + "</li></ul>\n";
    }

    // = <h1>
    if (line.startsWith("=")) {
        regex = "(=(.+?)=)";
        line = line.replaceAll(regex, "<h1>$2</h1>\n");

        // == <h2>
        regex = "(==(.+?)==)";
        line = line.replaceAll(regex, "<h2>$2</h2>\n");

        // === <h3>
        regex = "(===(.+?)===)";
        line = line.replaceAll(regex, "<h3>$2</h3>\n");
    }

    // [[]] <a>
    regex = "(\\[\\[(.+?)\\]\\]*)";
    line = line.replaceAll(regex,
            "<a href='index.jsp?page=$2'>$2</a>");

    // \[ -> [
    line = line.replaceAll("\\\\\\[", "[");
    line = line.replaceAll("\\\\\\]", "]");

    return line;
}
```

Langsam werden wir warm mit den regulären Ausdrücken, oder? You gotta love them.

Challenge

Wenn man noch etwas Zeit und Lust hat, kann man das Wiki ein bisschen erweitern:

- sich überlegen wie man mit multi-user Edits umgeht, oder alternativ
- eine Versionierung einführt,
- ein Login hinzufügt, damit nur eingeloggte Nutzer Änderungen vornehmen dürfen oder
- ein Image Upload ermöglicht, natürlich mit dazugehörigem Markup.

SI: Wir sollten es uns angewöhnen immer *escapeSql()* und *escapeXml()* zu verwenden.

Mensa

Kommen wir noch einmal zurück zu unserem Mensa Beispiel. Im ersten Kapitel haben wir ja ausführlich die Requirements gelistet und auch die UI umgesetzt. Am Anfang dieses Kapitels haben wir das Problem aus der Datenbank/POJO Sicht beleuchtet. Allerdings ist uns dabei ein kleiner Fehler unterlaufen. Denn die Beziehung zwischen Dish und Ingredient ist keine OneToMany Beziehung, sondern eine ManyToMany Beziehung. Warum? Weil die Zutat "Milch" durchaus in mehreren Gerichten vorkommen kann. Also, ein *Dish* hat mehrere *Ingredients* und ein *Ingredient* kann zu mehreren *Dishes* gehören, deswegen müssen wir hier eine ManyToMany Beziehung wählen. Wir müssen noch klären ob die Beziehung uni-direktional oder bi-direktional sein soll. In unseren Anforderungen steht nichts, dass wir auflisten sollen in welchen *Dishes* ein bestimmtes *Ingredient* vorkommt, deswegen ist die Beziehung uni-direktional. Also müssen wir im POJO *Dish* lediglich OneToMany durch ManyToMany ersetzen.

Nach diese kleine Änderung kümmern wir uns um die JSP Seiten. Die UI steht ja schon, wir müssen lediglich die Datenbank Logik hinzufügen. Beginnen wir mit der *listIngredients.jsp* Seite:

```
<html>
    <body>
        <%@include file="mensaNavigation.jsp" %>
        <h2>List all Ingredients</h2>
        <ul>
<%
    IngredientDao daoIng =
      (IngredientDao)application.getAttribute("Mensa.IngredientDao");
    List<Ingredient> ings = daoIng.findAll();
    for (Ingredient ing : ings) {
        out.println("<li>"+ing+"</li>");
    }
%>
        </ul>
    </body>
</html>
```

Die Liste der *Ingredients* ist jetzt nicht mehr statisch, sondern kommt aus der Datenbank.

Die *addIngredient.jsp* Seite wird ein klein wenig modifiziert, damit die Logik richtig funktioniert:

```
<html><body>
        <%@include file="mensaNavigation.jsp" %>
        <h2>Add New Ingredient</h2>
        <form action="mensaLogic.jsp" method="GET">
            Name:  <input type="text" name="name"/><br/>
            Size:  <input type="text" name="size"/><br/>
            Calories: <input type="text" name="calories"/><br/>
            <input type="hidden" name="addIngredient"/>
            <input type="submit" value="Add Ingredient"/>
        </form>
</body></html>
```

Denn wir wollen die Logik zentral in der *mensaLogic.jsp* Seite zusammenführen:

```
<%
    // add new ingredient
    if ( request .getParameter("addIngredient") != null ) {
        String name = request.getParameter("name");
        String size = request.getParameter("size");
        double calories =
            Double.parseDouble(request.getParameter("calories"));
        if (name != null) {
            Ingredient ingr = new Ingredient(name,size,calories);
            IngredientDao daoIng =
(IngredientDao)application.getAttribute("Mensa.IngredientDao");
            daoIng.merge(ingr);
        }
        response.sendRedirect("listIngredients.jsp");
        return;
    }

    // delete ingredient
    ...

    // add new dish
    ...

    // delete dish
    ...

    response.sendRedirect("mensa.jsp");
%>
```

Analog müssen wir jetzt nur noch die Seiten

- listDishes.jsp
- addDish.jsp
- deleteDish.jsp
- deleteIngredient.jsp

modifizieren und mit Leben füllen. Ist aber gar nicht so schwer.

Login

Die wohl am häufigsten benötigte Anwendung ist die Login Anwendung. Im ersten Kapitel haben wir unsere Anforderungen aufgestellt und die UI Screen bereits umgesetzt. Was noch fehlt ist die Logik. Die fassen wir in einer *loginLogic.jsp* Seite zusammen.

Die Loginseite, *login.jsp*, haben wir ja schon angelegt. Die Logindaten werden in der *loginLogic.jsp* Seite überprüft. Anstelle eines response.sendRedirect() verwenden wir das erste Mal den *RequestDispatcher*, der es uns erlaubt Daten mitzuschicken. Das ist nicht möglich mit sendRedirect().

```
<%
    // user login
    if (request.getParameter("login") != null) {
        String emailId = request.getParameter("emailId");
        String passwd = request.getParameter("password");
        if ((emailId != null) && (passwd != null)) {
            // try to create a user object
            UserDao usrDao =
                (UserDao)application.getAttribute("Login.UserDao");
            User usr = usrDao.loginUser(emailId,passwd);
```

```
            if ( usr != null ) {
                session.setAttribute("Login.User",usr);
                response.sendRedirect("protected.jsp");
                return;
            } else {
                request.setAttribute("error",
                    "Wrong username and/or password.");
                request.getRequestDispatcher("login.jsp")
                    .forward(request, response);
                return;
            }
        }
        request.setAttribute("error",
            "Please enter username and password.");
        request .get Request Dispatcher (" login .jsp")
            .forward( request , response );
        return;
    }
%>
```

Als nächstes kümmern wir uns um die Registrierung. Die Seite *register.jsp*, existiert schon. Die Daten werden wieder in der *loginLogic.jsp* Seite überprüft. Interessant an diesem Beispiel ist, dass wir das ganze *request* Objekt einfach als Argument an die Dao übergeben. Wir werden weiter unten sehen warum das super-praktisch ist. Wenn die eigentliche Registrierung erfolgreich war, dann bekommt der Nutzer eine Email zugesandt. Da wir noch nicht wissen wie das geht, geben wir einfach den Text der Email im Browser aus. Wenn der Nutzer aber auf den Link klickt, dann funktioniert alles wie gewünscht.

```
<%
    // registration
    if (request.getParameter("registration") != null) {

        UserDao usrDao =
            (UserDao)application.getAttribute("Login.UserDao");
        String errorMsg = usrDao.verifyRegistrationData( request );
        if ( errorMsg == null ) {
            String email = usrDao.createUser( request );
            out.println("Registration success!");
            out.println(email);
            return;

        } else {
            request.setAttribute("error", errorMsg);
            request.getRequestDispatcher("register.jsp")
                .forward(request, response);
            return;
        }
    }
%>
```

Es folgt das Ändern des Passworts. Die Seite *changePassword.jsp* ist schon implementiert. Die Daten gehen wieder an die *loginLogic.jsp* Seite, aber die eigentlich Arbeit macht wieder die UserDao.

```
<%
    // change password
    if (request.getParameter("changePassword") != null) {
        UserDao usrDao =
            (UserDao)application.getAttribute("Login.UserDao");
        User usr = (User)session.getAttribute("Login.User");
        String errorMsg = usrDao.changePassword( request ,usr);
        if ( errorMsg == null ) {
```

```
                out.println("Change of password success!");
                return;

            } else {
                request.setAttribute("error", errorMsg);
                request.getRequestDispatcher("changePassword.jsp")
                    .forward(request, response);
                return;
            }

        }
    %>
```

Es folgt das vergessene Passwort. Die Seite *forgotPassword.jsp* fragt nach EmailId und Lieblingsfarbe, und schickt die Daten an die *loginLogic.jsp* Seite. Wie üblich macht die UserDao die meiste Arbeit.

```
    <%
        // forgot password
        if (request.getParameter("forgotPassword") != null) {
            String emailId = request.getParameter("emailId");
            String favoriteColor = request.getParameter("favoriteColor");
            if ((emailId != null) && (favoriteColor != null)) {
                UserDao usrDao =
                    (UserDao)application.getAttribute("Login.UserDao");
                String email =
                    usrDao.forgotPassword(emailId,favoriteColor);
                if ( email == null ) {
                    request.setAttribute("error",
                        "The information you provided do not match "+
                        "any of our records.");
                    request.getRequestDispatcher("forgotPassword.jsp")
                        .forward(request, response);
                    return;
                } else {
                    out.println("The following email was sent to you: ");
                    out.println(email);
                    out.println("You should change your password at "+
                        "your next login!");
                    return;
                }
            }
        }

        // just make sure we did not forget an if or something
        response.sendRedirect("login.jsp");
    %>
```

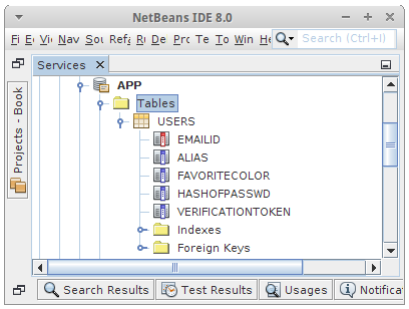
Auch hier ist das Versenden der Emails nur angedeutet, aber das Passwort ist das neue Passwort.

Was noch fehlt ist die *verifyEmail.jsp* Seite. Man hätte sie auch Teil der *loginLogic.jsp* Seite machen können, aber da der Link nach außen hin sichtbar ist, scheint es Sinn zu machen diese separat zu behandeln. Wie üblich macht die UserDao die meiste Arbeit.

```
<%
    String emailId = request.getParameter("emailId");
    String verificationToken =
        request.getParameter("verificationToken");
    if ((emailId != null) && (verificationToken != null)) {

        UserDao usrDao =
            (UserDao) application.getAttribute("Login.UserDao");
        if (usrDao.verifyUser(emailId, verificationToken)) {
            out.println("Your account is now activated!");
            out.println("Try to <a href='login.jsp'>login</a>");
            return;
        }
    }
    out.println("You must provide a valid emailId and a "+
                "valid verificationToken.");
%>
```

Wie wir die *protected.jsp* Seite schützen haben wir im Login Beispiel des letzten Kapitels gesehen.

POJOs



Aus Datenbanksicht ist die Login Anwendung trivial. Wir haben ein POJO für den Nutzer:

```
@Entity
@Table(name = "Users")
public class User {

    @Id
    private String emailId;

    @Column(nullable = false)
    private String alias;

    @Column(length = 256, nullable = false)
    private String hashOfPasswd;

    @Column(nullable = false)
    private String favoriteColor;

    @Column(nullable = true)
    private String verificationToken = "New user, not verified.";

    ... constructors, getters, setters, toString

    public boolean isVerifiedUser() {
        if (verificationToken == null) {
            return true;
        }
        return false;
    }
}
```

Interessant ist die Tatsache, dass es bei vielen Datenbanken bereits eine Tabelle namens "User" gibt, deswegen geben wir unserer Tabelle den Namen "Users", damit es hier nicht zu komischen Fehlermeldungen kommt. Aus Bequemlichkeit fügen wir noch eine Methode *isVerifiedUser()* hinzu.

DAO

Etwas komplizierter wird allerdings die UserDao, hier passiert die ganze Arbeit. Deswegen besprechen wir die verschiedenen Teile separat. Wir beginnen mit dem Konstruktor. UserDao erbt von GenericDao, also alle CRUD Standardmethoden werden übernommen. Was wir allerdings noch benötigen sind Informationen zum Hashen des Passwortes, denn wir wollen das Passwort nicht im Klartext in der Datenbank speichern. Dazu benötigen wir das Salt, und die Anzahl der Iterationen wie häufig gehasht werden soll.

```
public class UserDao extends GenericDao<String, User> {

    private byte[] salt;
    private int iterations;
    private int passwordHashedLength;

    public UserDao() {
        super();
        // TODO: the following should be loaded from config file:
        salt = Utility.hexStringToByteArray(
            "3577b8a868cce281cc76cf859613d5ad");
        iterations = 881;
        passwordHashedLength = 256; // related to User.hashOfPasswd
    }

}
```

Betrachten wir als nächstes die *initDatabase()* Methode: hier wird ein Testnutzer angelegt. Es zeigt wie wir aus dem Klartext-Passwort "123456" ein "gehashtes" Passwort erzeugen.

```
    public void initDatabase() {
        String pwdHash = Utility.generatePasswordHash(
                "123456", salt, iterations, passwordHashedLength);
        merge(new User("ralph@lano.de", "ralphlano", pwdHash,
                "red", null));
    }
```

Wenn sich ein Nutzer einloggen möchte, rufen wir die *loginUser()* Methode auf. Wir prüfen zuerst ob es einen Nutzer mit der gewünschten emailId gibt, dann überprüfen wir ob der Nutzer schon verifiziert wurde, um schließlich zu prüfen ob auch das richtige Passwort eingegeben wurde. Falls alles passt, geben wir eine Instanz des User Objektes zurück, sonst gibts *null*.

```
    public User loginUser(String emailId, String passwd) {
        User usr = findById(emailId);
        if (usr != null) {
            if (usr.isVerifiedUser()) {
                String pwdHash =
                    Utility.generatePasswordHash(passwd, salt,
                        iterations, passwordHashedLength);
                if (pwdHash.equals(usr.getHashOfPasswd())) {
                    return usr;
                }
            }
        }
        return null;
    }
```

Wenn wir es mit einem neuen Nutzer zu tun haben, dann müssen wir erst mal verifizieren, dass es kein Roboter ist (wir wollen keine Roboter als Nutzer, ziemlich fies, ich weiß) und dass alle wichtigen Daten eingegenen wurden:

```
public String verifyRegistrationData( HttpServletRequest request){
  String error = "An unknown error occured.";

  String emailId = request.getParameter("emailId");
  String alias = request.getParameter("alias");
  String passwd = request.getParameter("password");
  String favoriteColor = request.getParameter("favoriteColor");
  String sum = request.getParameter("sum");
  String result = request.getParameter("result");
  String acceptTerms = request.getParameter("acceptTerms");

  // check if robot
  if ((sum !=null) && (result !=null) && (sum.equals(result))) {

    // check if terms accepted.
    if ((acceptTerms != null) && (acceptTerms.equals("on"))) {

      // check if user entered proper data
      if ((emailId!=null) && (passwd!=null) &&
        (alias!=null) && (favoriteColor!=null)) {

        if (Utility.isValidEmail(emailId)) {
          if (Utility.isStrongPassword(passwd)) {
            if ((alias.length()>2) && (favoriteColor.length()>2)) {

              // now check if a user with this emailId already exists:
              if (findById(emailId) == null) {
                // everything is fine!
                error = null;

              } else {
                error =
                    "A user with this email address already exists.";
              }
            } else {
              error = "Your alias and/or favorite color must be at "+
                      "least 3 chars long.";
            }
          } else {
            error = "Your password must be at least 6 chars long "+
                    "and contain a small letter, a capital letter, "+
                    "a digit and a special character.";
          }
         } else {
          error = "Email is not a valid email address.";
         }
      } else {
        error = "All fields are required.";
      }
    } else {
      error = "You must accept the terms.";
    }
  } else {
    error = "Wrong captcha.";
  }
  return error;
}
```

Danach können wir die *createUser()* Methode aufrufen, und übergeben ihr das komplette *request* Objekt. Das ist viel praktischer als sechs Parameter individuell zu übergeben. Wir holen uns die Information die wir brauchen, und legen einen neuen User an. Wir generieren anschließend einen String, den wir als Email an den Nutzer schicken können damit dieser seine Email verifizieren kann. Solange er das nicht getan hat, ist sein Status nicht verifiziert, was wir daran erkennen, das sein Attribut *verificationToken* nicht auf null gesetzt ist.

```
public String createUser( HttpServlet Request request) {
    String emailId = request.getParameter("emailId");
    String alias = request.getParameter("alias");
    String passwd = request.getParameter("password");
    String favoriteColor = request.getParameter("favoriteColor");
    String verificationToken = Utility.createVerificationToken();
    String hashOfPasswd = Utility.generatePasswordHash(passwd, salt,
         iterations, passwordHashedLength);

    User usr = new User( emailId, alias, hashOfPasswd,
         favoriteColor, verificationToken);
    save(usr);

    String email =
         "Click on this link to verify your email identity: ";
    email += "<a href='verifyEmail.jsp?emailId="+emailId+
         "&verificationToken="+verificationToken+"'>link</a>";

    return email;
}
```

Allerdings ist in dem Code ein kleiner Denkfehler: was passiert denn wenn der Nutzer schon existiert?

So, der User existiert jetzt in der Datenbank, als nächstes erwarten wir, dass er auf den Verifizierungslink in seiner Email klickt. Also müssen wir ihn verifizieren:

```
public boolean verifyUser(String emailId,String verificationToken) {
    if ((emailId!=null) && (verificationToken!=null)) {
        User usr = findById(emailId);
        if (usr != null) {
            if (verificationToken.equalsIgnoreCase(
                                usr.getVerificationToken())) {

                usr.setVerificationToken(null);
                merge(usr);
                return true;
            }
        }
    }
    return false;
}
```

Fehlt noch die Möglichkeit sein Passwort zu ändern. Natürlich müssen wir erst checken ob das alte Passwort richtig war, und dann muss das neue Passwort natürlich was taugen, sonst bleibt's beim alten:

```
public String changePassword( HttpServlet Request request, User usr) {
  String error = "An unknown error occured.";

  String password = request.getParameter("password");
  String newPassword1 = request.getParameter("newPassword1");
  String newPassword2 = request.getParameter("newPassword2");
  if ((password != null) && (newPassword1 != null) &&
    (newPassword2 != null)) {
    if ( newPassword1.equals(newPassword2)) {
      if (usr != null) {
        String hashOfPasswd =
           Utility.generatePasswordHash(password, salt, iterations,
           passwordHashedLength);
        if (hashOfPasswd.equals(usr.getHashOfPasswd())) {
          if (Utility.isStrongPassword(newPassword1)) {
            String hashOfNewPasswd1 =
              Utility.generatePasswordHash(
              newPassword1, salt, iterations, passwordHashedLength);
            usr.setHashOfPasswd(hashOfNewPasswd1);
            merge(usr);
            // everything is fine!
            error = null;

          } else {
            error = "Your password must be at least 6 chars long "+
              "and contain a small letter, a capital letter, "+
              "a digit and a special character.";
          }
        } else {
          error = "You entered the wrong password.";
        }
      } else {
        error = "You must login before changing your password.";
      }
    } else {
      error = "Your new passwords do not match.";
    }
  } else {
    error = "You must enter all fields.";
  }
  return error;
}
```

Zum Schluß soll es ja solche Dödels geben die ihr Passwort vergessen. Die sollten dann wenigstens ihre Lieblingsfarbe wissen. Falls sie die wissen, bekommen sie eine Email, ansonsten behandeln wir sie wie Roboter:

```
public String forgotPassword(String emailId, String favoriteColor) {
    User usr = findById(emailId);
    if (usr != null) {

        if (favoriteColor.equalsIgnoreCase(usr.getFavoriteColor())) {
            String verificationToken =
                Utility.createVerificationToken();
            // make sure usr cannot login
            usr.setVerificationToken(verificationToken);

            String passwd = Utility.createStrongPassword();
            String hashOfPasswd = Utility.generatePasswordHash(
                     passwd, salt, iterations, passwordHashedLength);
            usr.setHashOfPasswd(hashOfPasswd);
            merge(usr);

            String email = "Your new password is: " + passwd +"<br/>";
            email +=
              "Click on this link to verify your email identity: ";
            email += "<a href='verifyEmail.jsp?emailId="+emailId+
              "&verificationToken="+ verificationToken+"'>link</a>";
            return email;
        }
    }
    return null;
}
```

Das war jetzt ziemlicher schwerer Tobak, aber mit dem Top-Down Ansatz aus dem ersten Semester haben wir das auch geschafft.

Fassen wir noch einmal zusammen was wir gelernt haben:

- Datenbank Tabellen sollten nie "User" heißen
- die Datenbank initialisieren wir in *jspInit()* (allerdings nur zum Testen, sollte man nicht auf einer Production Maschine machen)
- die Logik ist idealerweise an einer Stelle *loginLogic.jsp* zusammengefasst
- die DAO kann sehr viel Arbeit erledigen
- den Workflow von Registration, Verification über Change Password hinzu Forgot Password
- security issues, hashing, regular expressions, strong passwords, passwordHashed, salt
- wie man dem Nutzer Feedback bei falschen Eingaben geben kann
- wie man die Sessions verwendet um geschützte Seite zu schützen
- wie ein Verification Link funktioniert
- wie man aus einem ByteArray einen String generiert.

Wenn man jetzt noch eine kleine Herausforderung sucht, dann könnte man sich überlegen was noch fehlt damit man zwei verschiedene Arten von Nutzern haben könnte, also z.B. normale Nutzer und solche mit Administratorrechten.

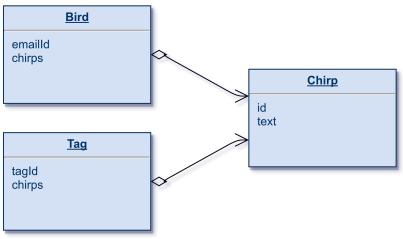
Chirpr

Unser Mini-Soziales-Netzwerk Chirpr haben wir auch schon gegen Ende des ersten Kapitels vorgestellt. Die UI Seiten haben wir schon angelegt, es fehlt nur noch ein wenig Logik. Ein Teil davon passiert in der *chirprLogic.jsp* Seite:

```
<%
    // user login
    if (request.getParameter("login") != null) {
        String emailId = request.getParameter("emailId");
        if ((emailId != null) && (emailId.length() > 3)) {
            // try to create a user object
            BirdDao brdDao =
                (BirdDao)application.getAttribute(" Chirpr.BirdDao");
            Bird brd = brdDao.getBird(emailId);

            if (brd != null) {
                session.setAttribute(" Chirpr.Bird",brd);
                response.sendRedirect("protected.jsp");
                return;
            }
        }
        request.setAttribute("error",
            "Your username must be at least 4 characters long.");
        request.get RequestDispatcher("index.jsp")
            .forward(request, response);
        return;
    }

    // chirp
    if (request.getParameter("chirp") != null) {
        Bird brd = (Bird)session.getAttribute(" Chirpr.Bird");
        if ( brd != null) {
            String text = request.getParameter("text");
            if ((text != null) && (text.length() >= 5) &&
                (text.length() <= 42)) {
              Chirp crp = new Chirp(text);
              ChirpDao crpDao =
                (ChirpDao)application.getAttribute("Chirpr.ChirpDao");
              crpDao.save(crp);
              brd.addChirp(crp);
              BirdDao brdDao =
                (BirdDao)application.getAttribute("Chirpr.BirdDao");
              brdDao.merge(brd);
              TagDao tagDao =
                (TagDao)application.getAttribute("Chirpr.TagDao");
              tagDao .parse Chirp ForTags(crp);
            }
        }
        response.sendRedirect("protected.jsp");
        return;
    }

    // just make sure we did not forget an if or something
    response.sendRedirect("index.jsp");
%>
```

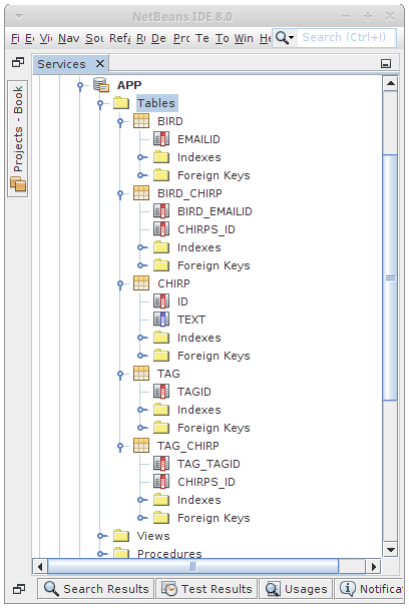
Der andere Teil passiert in den Daos.

POJOs



Bevor wir zu den Daos kommen, kümmern wir uns um die POJOs. Wir beginnen mit den Birds, also den Nutzern. Die Birds setzen Chirps ab, haben also keinen, einen oder mehrere, also eine OneToMany Beziehung. Ob man ein Set oder eine Liste als Container verwendet hängt davon ab, ob einem die Reihenfolge der Chirps wichtig ist. Uns ist sie nicht wichtig.

```
public class Bird {

    @Id
    private String emailId;

    @ OneToMany
    private Set< Chirp > chirp s;

    ... constructors, getters, setters, toString
}
```



Kommen wir zu den Chirps. Jeder Chirp kann Tags enthalten und jedes Tag kann in mehreren Chirps vorkommen, also eigentlich eine ManyToMany Beziehung. Wenn wir unsere Anforderungen aber genau durchlesen, gibt es niemals die Anforderung dass wir die Tags eines Chirps auflisten müssen. D.h. die Chirps müssen gar nichts über ihre Tags wissen. Das erlaubt es uns die viel komplexere ManyToMany Beziehung durch eine OneToMany Beziehung auf der Tag Seite zu ersetzen.

```
public class Chirp {

    @Id
    @GeneratedValue(strategy=
                    GenerationType.IDENTITY)
    private Long id;

    @Column(length = 42, nullable = false)
    private String text;

    ... constructors, getters, setters, toString
}
```

Und damit ergibt sich das Tag POJO:

```
public class Tag {

    @Id
    private String tagId;

    @OneToMany
    private Set<Chirp> chirps;

    ... constructors, getters, setters, toString
}
```

Das ist gar nicht so kompliziert, oder?

DAOs

Kommen wir zu den DAOs. Die BirdDao ist ganz einfach, wir haben nur eine Zusatzanforderung, wenn ein Bird noch nicht existiert, soll einfach ein neuer angelegt werden:

```
public class BirdDao extends GenericDao<String, Bird> {

    public BirdDao() {
        super();
    }

    public Bird getBird(String emailId) {
        Bird brd = findById(emailId);
        if (brd == null) {
            brd = new Bird(emailId);
            save(brd);
        }
        return brd;
    }
}
```

Die ChirpDao übernimmt alle Eigenschaften der GenericDao und initialisiert die Datenbank mit zwei Birds und drei Chiprs:

```
public class ChirpDao extends GenericDao<Long, Chirp> {

    public ChirpDao() {
        super();
    }

    public void initDatabase(BirdDao brdDao,TagDao tagDao) {
        Bird bd1 = new Bird("ralph");
        brdDao.save(bd1);
        Bird bd2 = new Bird("vince");
        brdDao.save(bd2);

        Chirp cp1 = new Chirp("Welcome to # Chirpr");
        save(cp1);
        bd1.addChirp(cp1);
        tagDao.parseChirpForTags(cp1);

        Chirp cp2 = new Chirp("#Java is the greatest!");
        save(cp2);
        bd2.addChirp(cp2);
        tagDao.parseChirpForTags(cp2);

        Chirp cp3 = new Chirp("# Chirpr is written in #Java");
        save(cp3);
        bd2.addChirp(cp3);
        tagDao.parseChirpForTags(cp3);

        brdDao.merge(bd1);
        brdDao.merge(bd2);
    }
}
```

Schließlich kommen wir zur TagDao. Hier übernehmen wir wieder alle Eigenschaften der GenericDao. Dann müssen wir aber noch zwei Dinge tun: erst einmal müssen wir in der Methode *parseChirpForTags()* einen Chirp nach Hash-Tags durchsuchen. Das machen wir mit einem Regulären Ausdruck (got to love them by now). Und wir müssen die Tags mit all ihren Chirps speichern, das machen wir in *addChirpToTag()*.

```
public class TagDao extends GenericDao<String, Tag> {

    public TagDao() {
        super();
    }

    public void parseChirpForTags(Chirp crp) {
        Pattern pat = Pattern.compile("(^|\\s)\\#(\\w+)");
        Matcher mat = pat.matcher(crp.getText());
        while (mat.find()) {
            String tag = mat.group(2);
            //System.out.println("."+tag+".");
            addChirpToTag(tag.toLowerCase(),crp);
        }
    }

    private void addChirpToTag(String tag, Chirp crp) {
        Tag tg = findById(tag);
        if ( tg == null ) {
            tg = new Tag(tag);
            save(tg);
        }
        tg.addChirp(crp);
        merge(tg);
    }
}
```

Und das war's dann eigentlich schon.

Challenge

Das Gefühl, dass alles immer einfach ist, kann manchmal täuschen. Nehmen wir an, wir hätten eine Anforderung die besagt, dass in der Auflistung der Chirps, neben jedem Chirp der Bird (also Nutzer) stehen soll, der diesen Chirp geschrieben hat, z.B.

```
#Java is the greatest! ralph
```

Mit unserem momentanen Datenmodell ist diese Anforderung nicht zu erfüllen! Was müssten wir denn ändern damit das möglich wäre? Was wären denn die Konsequenzen? Ist es das wirklich wert? Das ist wieder unsere berühmte 80-20 Regel, auch Pareto Prinzip genannt.

Quizs

Bei Quizs handelt es sich um eine Webanwendung für die Erstellung und Durchführung von Tests. Die Anforderungen haben wir ja schon im ersten Kapitel gesehen, dort haben wir auch die UI Seiten bereits angelegt.

Kommen wir zur Logik, ein Teil davon passiert in der *quizsLogic.jsp* Seite:

```
<%
    // user login
    if (request.getParameter("login") != null) {
        String emailId = request.getParameter("emailId");
        if ((emailId != null) && (emailId.length() > 3)) {
            if ( emailId.equals("teacher")) {
                session.setAttribute("Quizs.Teacher","Teacher");
                response.sendRedirect("teacher/");
                return;

            } else {
                // try to create a user object
                StudentDao stdntDao =
(StudentDao)application.getAttribute("Quizs.StudentDao");
                Student stdnt = stdntDao.getStudent(emailId);
                if (stdnt != null) {
                    session.setAttribute("Quizs.Student",stdnt);
                    response.sendRedirect("student/");
                    return;
                }
            }
        }
        request.setAttribute("error",
            "Your username must be at least 4 characters long.");
        request.get RequestDispatcher("index.jsp")
            .forward(request, response);
        return;
    }

    // just make sure we did not forget an if or something
    response.sendRedirect("index.jsp");
%>
```

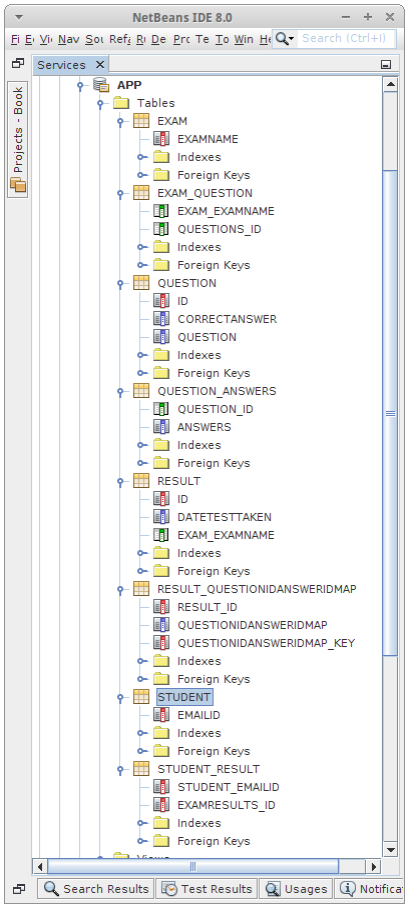
Der Rest der Logik passiert in den Daos.

POJOs

In diesem Beispiel gibt es einige POJOs. Ein Student kann an Examen teilnehmen. Ein Examen besteht aus mehreren Fragen. Und jedes Examen hat Resultate.

Wir beginnen mit dem Question POJO, weil es das einfachste ist. Eine Frage hat mehrere Antworten, und Antworten sind einfache Strings. Bei den Antworten kommt es aber ausnahmsweise mal auf die Reihenfolge an, deswegen verwenden wir heute eine Liste:

```
public class Question {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String question;

    @ElementCollection
    @Column(nullable = false)
    private List<String> answers;

    @Column(nullable = false)
    private Integer correctAnswer;

    ... constructors, getters, setters, toString
}
```



Ein Exam besteht aus mehreren Fragen, und da eine Frage durchaus in verschiedenen Exams sein darf, handelt es sich um eine ManyToMany Beziehung. Auch hier wollen wir, dass die Reihenfolge festgelegt ist, deswegen eine Liste als Datentyp:

```
public class Exam {

    @Id
    private String examName;

    @ ManyToMany
    private List<Question> questions;

    ... constructors, getters, setters, toString
}
```

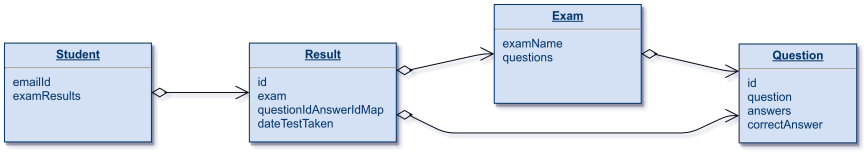
Für die Resultate eines Exams verwenden wir das Result POJO. Resultate gehören immer zu einem Exam, deswegen eine OneToOne Beziehung zwischen Result und Exam. Muss diese *unique* sein? Die Antwort ist nein, da ja zwei Studierende am gleichen Exam teilnehmen können, aber durchaus zwei unterschiedliche Resultate erziehlen können. Kommen wir zu den Antworten: das sind die Antworten die der Studierende abgegeben hat. In diesem Fall ist unsere Collection eine Map, in der die Id der Frage der Key ist und die Antwort, die der Studierende gegeben hat, der Value.

```
public class Result {

    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Long id;

    @ OneToOne
    private Exam exam;

    @ElementCollection
    @Column(nullable = true)
    private Map<Long,Integer> questionIdAnswerIdMap;

    @Column(nullable = false)
    private Date dateTestTaken;

    ... constructors, getters, setters, toString
}
```

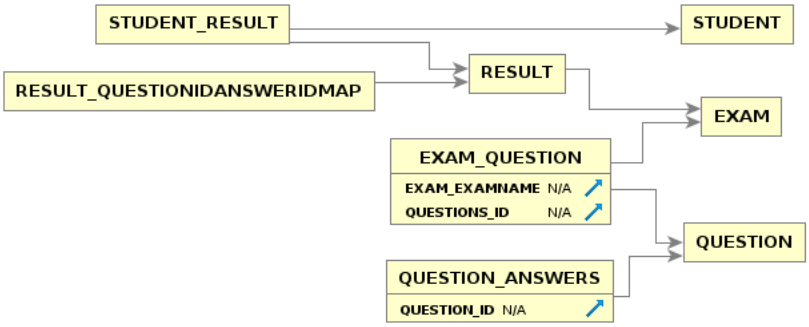
Zum Schluß fehlt noch das Student POJO: ein Studierender muss über seine Resultate Bescheid wissen. Da ein Studierender an mehreren Tests teilnehmen kann ist es entweder eine OneToMany oder eine ManyToMany Beziehung. Fragt sich, ob ein bestimmtes Resultat eines bestimmten Tests zu mehr als einem Studierenden gehören kann. Da wir keine Gruppenarbeit zulassen wollen, ist die Antwort nein, und deswegen handelt es sich um eine OneToMany Beziehung.

```
public class Student {
    @Id
    private String emailId;

    @ OneToMany
    private Set<Result> examResults;

    ... constructors, getters, setters, toString
}
```

Bei so vielen POJOs kann man schon den Überblick verlieren, da hilft ein Klassendiagramm die Übersicht zu bewahren.



Interessanterweise ist es um einiges einfacher als das korrespondierende Entity Relationship Diagramm:



Aber natürlich ist das ERD genauer, es enthält mehr Information.

DAOs

Kommen wir zu den Daos. Die QuestionDao ist trivial:

```
public class QuestionDao extends GenericDao<Long, Question>  {

    public QuestionDao() {
        super();
    }
}
```

ebenso die ResultDao:

```
public class ResultDao extends GenericDao<Long, Result>  {

    public ResultDao() {
        super();
    }
}
```

Die ExamDao ist nur etwas komplizierter, aber nur weil wir die Datenbank mit wenigstens einem Exam initialisieren wollen:

```
public class ExamDao extends GenericDao<String, Exam>  {

    public ExamDao() {
        super();
    }

    public void initDatabase(QuestionDao qstnDao) {
        List<String> anss1 = new ArrayList<String>();
        anss1.add("1");
        anss1.add("2");
        anss1.add("4");
```

```
            Question q1 =
                new Question("What is 1 + 1?", anss1, 2);
            qstnDao.save(q1);

            List<String> anss2 = new ArrayList<String>();
            anss2.add("red");
            anss2.add("black");
            anss2.add("blue");
            Question q2 =
                new Question("Which color is the sky?", anss2, 3);
            qstnDao.save(q2);

            Exam ex = new Exam("Simple Exam");
            ex.addQuestion(q1);
            ex.addQuestion(q2);
            save(ex);
        }
    }
```

Die StudentDao ist auch nicht besonders kompliziert. Auch hier wollen wir zwei Studierende vorinstallieren und wir überschreiben die *getStudent()* Methode, denn wir möchten, dass Studierende einfach neu angelegt werden, falls sie noch nicht exisiterien sollten.

```
public class StudentDao extends GenericDao<String, Student>  {

    public StudentDao() {
        super();
    }

    public Student getStudent(String emailId) {
        Student stdnt = findById(emailId);
        if (stdnt == null) {
            stdnt = new Student(emailId);
            save(stdnt);
        }
        return stdnt;
    }

    public void initDatabase() {
        Student st1 = new Student("ralph");
        save(st1);
        Student st2 = new Student("vince");
        save(st2);
    }
}
```

Nach diesem Projekt sollte uns eines klar geworden sein: dadurch dass wir unsere Webanwendungen in kleine Teilprojekte zerlegen (Top-Down lässt grüßen) und dass wir die UI vom Datenmodell und Businesslogik trennen (View-Model-Controller), lassen sich selbst ursprünglich komplex anmutende Anwendung relativ problemlos implementieren.

Research

Manche Themen habe wir hier nur sehr oberflächlich behandelt. Man könnte sich aber das eine oder andere noch etwas detailierter ansehen.

ORM Engines

Hibernate ist nicht die einzige ORM Engine. Um sicher zu gehen, dass wir mit Hibernate auch eine gute Wahl getroffen haben, sollten wir erst einmal die alternativen ORM Engines für Java finden, und dann die wichtigsten vergleichen. Außerdem sollten wir mal nach ORM Engines für andere Programmiersprachen wie ASP, PHP oder Python suchen. Falls eine Sprache keine ORM Engines hat, sollte man die Finger davon lassen.

NoSql

Unter dem Kürzel "NoSql" sind in den letzten Jahren sehr viele neue Datenbankkonzepte eingeführt worden. Bevor man sich allerdings blindlings auf das "Neue" stürzt, sollte man erst einmal wissen worauf man sich da einlässt und was eigentlich dahinter steckt. Eine Frage die uns natürlich interessiert, kann man ORM auch mit einer NoSql Datenbank verwenden, und macht das überhaupt Sinn?

Fragen

1. Was ist ein POJO? Wofür steht “POJO”? Wofür sind POJOs nützlich?

2. Schreiben Sie eine Klasse Book und fügen Sie Hibernate Annotationen hinzu. Die Klasse Book sollte die folgenden Eigenschaften haben: title, author und price. Achten Sie darauf dass Sie auch die Konstruktoren mit angeben. Bei den Gettern und Settern können Sie davon ausgehen, dass diese autogeneriert wurden. Hinweis, die folgenden Annotationen könnten hilfreich sein:
   @Column(length = 32000, nullable = false)
   @Entity
   @Gene

3. Vergleichen Sie die normale Art und Weise mit Datenbanken zu arbeiten (also SQL) mit der objekt-orientierten Art und Weise (also ORM). Nennen Sie je zwei Vorteile.

4. Was ist beim Speichern von Passwörtern in Datenbanken zu beachten, oder wie würden Sie Passwörter in einer Datenbank speichern?

5. Zeigen Sie anhand eines konkreten Beispiels (z.B. Usern und ihren Email Addressen) den Unterschied zwischen One-to-One, Many-to-One und Many-to-Many Beziehungen. Geben Sie für jede konkrete Beispiele wie die Tabellen in der Datenbank aussehen. Beschreiben Sie auch genau welche evtl. Constraints auf den jeweiligen Spalten (Columns) liegen müssen.

6. Nennen Sie vier Gründe, warum man keine binären Dateien in der Datenbank speichern sollte. Nennen Sie einen warum es evtl. manchmal doch Sinn macht.

7. Angenommen eine Seite fragt Sie nach Benutzernamen und Passwort. Sie haben aber sowohl Ihren Benutzernamen als auch Ihr Passwort vergessen. Was müssten Sie tun um sich mittels SQL Injection trotzdem einloggen zu können? Ist dieses Vorgehen legal?

8. Erklären Sie bitte wie SQL Injektion funktioniert (geben Sie evtl. ein Beispiel) und nennen Sie drei Möglichkeiten wie man SQL Injektion verhindern kann.

9. In diesem Kapitel habe wir viel mit Hibernate gearbeitet. Dabei haben wir die folgenden Dateien generiert oder verwendet. Erklären Sie bitte was diese enthalten, und wofür sie benötigt werden, evtl. mit Beispiel:
   - hibernate.cfg.xml:
   - HibernateUtil.java:
   - User.java:
   - UserDao.java:

Referenzen

Alles was mit Hibernate zu tun hat findet man in Referenz [1]. Einen kleinen Überblick was sich mit D3JS anstellen lässt gibt [2].

[1] HIBERNATE - Relational Persistence for Idiomatic Java, Hibernate Reference Documentation, https://docs.jboss.org/hibernate/orm/3.6/reference/en-US/html/index.html

[2] D3.js - Data-Driven Documents, https://d3js.org

[3] Mike Bostock's Block, Line Chart, https://bl.ocks.org/mbostock/3883245

Services

Mit der Verbreitung von Apps und mobilen Endgeräten hat auch der Bedarf an Web-Services explosionsartig zugenommen. In diesem Kapitel werden wir sehen wie wir Services zur Verfügung stellen, und wie diese im Detail funktionieren. Nächstes Semester können wir dann basierend auf diesen Services mobilen Anwendungen realisieren.

SimpleServlet

Wir haben Servlets bereits im zweiten Kapitel kennengelernt: damals haben wir gesehen, dass aus jeder JSP Seite erst einmal ein Servlet generiert wird, das dann kompiliert und später ausgeführt wird. Was sind Servlets? Servlets sind einfach Java Klassen die die Klasse *HttpServlet* als Elternklasse haben:

```
public class SimpleServlet extends HttpServlet {

    @Override
    protected void doGet( HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello from SimpleServlets doGet() method.");
    }

    @Override
    protected void doPost( HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello from SimpleServlets doPost() method.");
    }

    @Override
    public String getServletInfo() {
        return "Demonstration of servlets.";
    }
}
```

Es gibt eine *doGet()* und eine *doPost()* Methode, die wir überschreiben können, wie im obigen Beispiel. Die erstere wird aufgerufen wenn wir einen HTTP GET Request erhalten, die letztere wenn wir einen POST Request erhalten. Servlets erlauben es uns also zwischen den beiden zu unterscheiden. Um den Unterschied zu sehen, betrachten wir eine kleine JSP Seite:

```
<html>
    <body>
        <h1>SimpleServlet </h1>

        <p>Access servlet via <a href="simpleservlet">GET</a>
           request.</p>

        <form action="simpleservlet" method="POST">
            <p>Access servlet via <input type="submit" value="POST"/>
               request.</p>
        </form>

    </body>
</html>
```

Über den Link rufen wir die *doGet()* Methode auf, über das Formular die *doPost()* Methode.

Interessant ist auch wie Servlets aufgerufen werden. Die URL lautet einfach

```
http://localhost:8080/Pr6_Service/simpleservlet
```

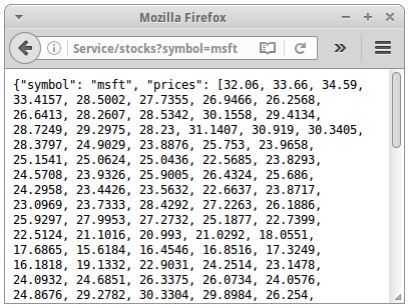
wir sagen einfach "simpleservlet" also ohne eine Endung wie ".jsp" oder ".servlet" oder so. Der Grund dafür ist die *web.xml* Datei. Die Datei befindet sich im /WEB-INF Verzeichnis bei unseren JSP Seiten, und war schon immer da, falls nicht, kann man sie autogenerieren. Für jedes Servlet das wir schreiben, müssen wir einen Eintrag in die *web.xml* Datei machen:

```
<web-app version="3.1" ...>
    <servlet>
        <servlet-name>SimpleServlet </servlet-name>
        <servlet-class>
          de.variationenzumthema.internet.simpleservlet.SimpleServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SimpleServlet </servlet-name>
        <url-pattern>/simpleservlet</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Wir sehen also, dass der URL-Pattern "simpleservlet" auf das Servlet mit Namen *SimpleServlet* gemapped wird. Dessen Deklaration wiederum in der Klasse de.variationenzumthema.internet.simpleservlet. SimpleServlet zu finden ist.

Zwei andere Dinge die wir hier noch sehen: zum Einen wird hier anscheinend die Session-Timeout Zeit festgelegt, in diesem Fall 30 Minuten. Und die Begrüßungsdatei, also das welcome-file, ist "index.jsp". Das bedeutet, immer wenn ein Nutzer nur das Verzeichnis und keine genaue Dateiangabe für die URL macht, wird diese Datei angezeigt.

StocksServlet



Natürlich können Servlets auch auf Datenbanken zugreifen. Als Beispiel nehmen wir das Aktienbeispiel aus dem letzten Kapitel. POJO, Dao und Hibernate Konfiguration sind vollkommen unverändert. Nur anstelle einer JSP Datei, verwenden wir jetzt ein Servlet für die Ausgabe:

```
public class StocksServlet extends HttpServlet {

    @Override
    public void init() throws ServletException {
        super.init();
        String path = getServletContext().getRealPath("/") +
            "SP500_HistoricalStockDataMonthly.csv";
        ServletContext application = get ServletContext();

        // init database with data from file:
        StockDao dao = new StockDao();
        List<String> dates = dao.initStockPriceTable(path);
        application.setAttribute("Stocks.dao", dao);
        application.setAttribute("Stocks.dates", dates);
    }
```

```
    @Override
    protected void doGet( HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        // query database for symbol
        String symbol = request.getParameter("symbol");
        StockDao dao =
            (StockDao) getServletContext().getAttribute("Stocks.dao");
        List<String> dates = (List<String>)
            getServletContext().getAttribute("Stocks.dates");
        StockSymbol sp = dao.getStockPrice(symbol);

        PrintWriter out = response.getWriter();
        if (sp != null) {
            out.println(sp.getJSON());
        } else {
            //out.println("symbol not found");
            //response.setStatus(HttpServletResponse.SC_NOT_FOUND);
            response.sendError(
                HttpServletResponse.SC_NOT_FOUND,
                "stock symbol not found");
        }
    }
}
```

Wir sehen, dass Servlets auch eine *init()* Methode haben, die vollkommen analog zur *jspInit()* Methode funktioniert. Auch auf das *application* Objekt können wir dort zugreifen. In der *doGet()* Methode ist dann die eigentliche Logik: auch wieder mit request.getParameter() greifen wir auf die Parameter zu. Einzig, das *out* Objekt ist jetzt nicht mehr vordefiniert, das können wir aber auch selbst machen, wie wir oben sehen.

JSON

Ein kleiner Unterschied zur JSP Version existiert: die Klasse *StockSymbol* hat zusätzlich eine Methode namens *getJSON()*:

```
public class StockSymbol {
    ...
    public String getJSON() {
        return "{\"symbol\": \"" + symbol + "\", \"prices\": " +
            prices + '}';
    }
}
```

JSON steht für "JavaScript Object Notation" [1] und ist ein Standard der sehr häufig im Zusammenhang mit Webservices verwendet wird. Es ist ein sehr einfaches Format das es erlaubt Daten, und vor allem Objekte, zwischen zwei Parteien auszutauschen. Früher war dafür hauptsächlich XML vorgesehen, heute wird aber fast ausschließlich JSON verwendet.

In JSON werden Arrays einfach mit eckigen Klammern dargestellt, also aus dem Java

```
String[] months = {"Jan","Feb",...,"Dec"};
```

wird einfach das folgende JSON

```
["Jan","Feb",...,"Dec"]
```

Noch interessanter aber ist, dass wir auch Objekte mit JSON darstellen können. Wenn wir aus der Java Klasse *Dog*:

```
public class Dog {
    private String name;
    private int age;
    public Dog(String n, int a) {
        name = n;
        age = a;
    }
}
```

ein Objekt erzeugen, z.B., new Dog("Lassie", 42), dann wird daraus folgendes JSON Objekt:

```
{name: "Lassie", age: 42}
```

Richtig cool ist, wie JavaScript mit JSON Objekten umgehen kann, und das ist eigentlich der Hauptgrund für den Siegeszug von JSON.

Histogram

Wir können natürlich Servlets verwenden um Webservices zu schreiben. Das geht ohne Probleme, denn neben den doGet() und doPost() Methoden gibt es auch noch eine doPut() und eine doDelete() Methode. Allerdings geht es etwas einfacher mit dem *Jersey Framework* [2], das inziwschen Teil von Standard Java ist. Damit Jersey funktioniert muss man einen Eintrag in die *web.xml* Datei hinzufügen:

```
<web-app version="3.1" ...>
    ...
    <servlet>
        <servlet-name>rest</servlet-name>
        <servlet-class>
            org.glassfish.jersey.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>
                jersey.config.server.provider.packages
            </param-name>
            <param-value>
                de.variationenzumthema.internet.service
            </param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>rest</servlet-name>
        <url-pattern>/service/*</url-pattern>
    </servlet-mapping>

</web-app>
```

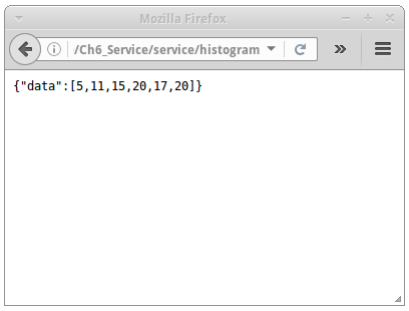
Wie wir sehen basiert Jersey auf Servlets. Und wir sehen, dass alles was mit der URL /service/ beginnt an Jersey weitergeleitet wird.

Betrachten wir unser Histogram Beispiel aus dem ersten Semester. Zunächst geht es wieder darum JSON zu erzeugen. Beginnen wir mit einer Klasse *HistogramData*, die die Daten für ein Histogramm beinhaltet.

```
@XmlRootElement
public class HistogramData {
    private Integer[] data = {5, 11, 15, 20, 17, 20};

    public Integer[] getData() {
        return data;
    }

    public void setData(Integer[] data) {
        this.data = data;
    }
}
```

Es handelt sich um eine ganz einfache Klasse, aber es könnte natürlich auch ein POJO sein, dessen Daten aus einer Datenbank kommen. Die Annotation *@XmlRootElement* ist nur nötig wenn wir XML erzeugen wollen. Solange wir nur JSON erzeugen wollen können wir sie auch weglassen.

Um daraus jetzt einen Webservice zu machen, verwenden wir wie bei Hibernate Annotationen:




```
@Path("/histogram")
public class HistogramResource {

    public HistogramResource() {
        super();
    }

    @GET
    @Produces ( MediaType.APPLICATION_JSON )
    public HistogramData getHistogram() {
        return new HistogramData();
    }
}
```

Die erste Annotation @Path("/histogram") besagt, dass der Service unter der URL

```
http://localhost:8080/Ch6_Service/service/histogram
```

aufgerufen werden kann. Die @GET Annotation besagt, dass diese Methode aufgerufen werden soll wenn ein HTTP GET Request anfragt. Und schließlich die @Produces Annotation ist der Hauptgrund warum wir Jersey verwenden: denn die sorgt dafür, dass aus dem Objekt "new HistogramData()" automatisch JSON generiert wird. Falls wir möchten, dass XML generiert wird, müssen wir lediglich MediaType.APPLICATION_JSON durch MediaType.APPLICATION_XML ersetzen (und noch die Annotation @XmlRootElement vor die Klasse HistogramData schreiben).

Die *histogram.jsp* Datei zeigt, wie man dann mittels JavaScript und diesen JSON Daten einen sehr hübschen Barchart erzeugen kann. Der JavaScript Code ist praktisch unverändert von Scott Murray's "Making a bar chart" Tutorial übernommen [3].

IPBlocking

Kommen wir zum nächsten wichtigen Konzept in diesem Kapitel, den Filtern. Nehmen wir an wir möchten den Zugriff auf unseren Server einschränken. Z.B. haben wir festgestellt, dass von den IP Adressen die mit "127.0." beginnen ein Denial-of-Service Attacke auf unseren Server gefahren wurde, und wir daher diese Adressen nicht auf unseren Server lassen wollen. Oder umgekehrt, wir möchten nur Zugriff aus dem lokalen Netzwerk erlauben. Beides kann man sehr einfach mit Filtern erledigen.

Dazu muss man zunächst in der *web.xml* Datei den Filter anmelden:

```
<web-app version="3.1" ...>
    ...
    <filter>
        <filter-name>IPBlockingFilter</filter-name>
        <filter-class>
de.variationenzumthema.internet.ipblockingfilter.IPBlockingFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>IPBlockingFilter</filter-name>
        <url-pattern>/ipblocking/*</url-pattern>
    </filter-mapping>

</web-app>
```

Jetzt wird also der gesammte Verkehr der an die URL /ipblocking/ gerichtet ist, erst einmal an den *IPBlockingFilter* weitergeleitet. Dieser entscheidet dann was mit den Anfragen passieren soll:

```
public class IPBlockingFilter implements Filter {

    private FilterConfig config;

    @Override
    public void init(FilterConfig filterConfig)
        throws ServletException {
        this.config = filterConfig;
    }

    @Override
    public void do Filter (ServletRequest request,
            ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

        String ip = request.getRemoteAddr();
        if (!ip.startsWith("127.0.")) {
            chain.doFilter(request, response);

        } else {
            if (response instanceof HttpServletResponse ) {
                ((HttpServletResponse) response).sendError(
                        HttpServletResponse.SC_FORBIDDEN,
                        "Your IP has been blocked!");
            }
        }
    }

    @Override
    public void destroy() {
        // does nothing
    }
}
```

Die ganze Magie passiert in der *doFilter()* Methode, hier entscheiden wir ob wir den weiteren Zugriff zulassen wollen, das geht dann mit *chain.doFilter()*, oder ob wir dem Browser einfach ein freche Antwort schicken, dass der Zugriff verboten ist. Natürlich sehen wir sofort, dass Filter super-praktisch für unser Sicherheitsmanagement sind. Dazu später mehr.

Review

In diesem Kapitel haben drei wichtige Konzepte kennengelernt:

- Servlets,
- Filter und
- RESTful Webservices und JSON.

Wobei wir die RESTful Webservices nur angedeutet haben. Gleich kommt aber mehr dazu.

Projekte

Die Projekte in diesem Kapitel vertiefen Webservices an zwei Beispielen, und zeigen wie Filter für die Sicherung unserer Webseiten verwendet werden können.

WorldMap

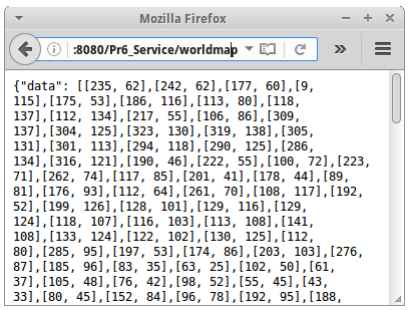
Natürlich glaubt mir immer keiner, dass JSON wirklich cool ist, deswegen machen wir gleich ein Beispiel. Auch wieder in unserem ersten Semester haben wir ja eine WorldMap gemalt. Damals mit der ACM Graphics Library. Für Webgeschichten stellt sich heraus, dass die D3JS JavaScript Library fast genauso cool ist. Wir haben sie ja schon zweimal weiter oben in Aktion gesehen.

Zunächst schreiben wir ein Servlet, dass unsere Geo-Daten für Städte in einem JSON Format ausgibt. Der Code ist fast identische zu dem aus dem ersten Semester.

```
public class WorldMapServlet extends HttpServlet {

    private final int WIDTH = 350;
    private final int HEIGHT = 200;
    private String json = "";

    @Override
    public void init() throws Servlet Exception {
        super.init();
        String path =
            getServletContext().getRealPath("/") + "Cities.txt";
        json = loadAndDisplayData(path);
    }

    private String loadAndDisplayData(String fileName) {
        String json = "{\"data\": [";
        //json += "[ [5, 20], [480, 90], [250, 50], [100, 33]]";
        try {
            BufferedReader br =
                new BufferedReader(new FileReader(fileName));
            while (true) {
                String line = br.readLine();
                if (line == null) {
                    break;
                }
```

```
                    if (!line.startsWith("#")) {
                        // Germany, Berlin, 52", 32', N, 13", 25', E
                        String[] data = line.split(",");
                        String country = data[0].trim();
                        String name = data[1].trim();
                        String lat1 = data[2].trim();
                        String lat2 = data[3].trim();
                        String lat3 = data[4].trim();

                        int lat = Integer.parseInt(lat1);
                        if (lat3.endsWith("S")) {
                            lat = -lat;
                        }
                        String lon1 = data[5].trim();
                        String lon2 = data[6].trim();
                        String lon3 = data[7].trim();
                        int lon = Integer.parseInt(lon1);
                        if (lon3.endsWith("E")) {
                            lon = -lon;
                        }

                        int x = (int)((0.5 - lon / 360.0) * WIDTH);
                        int y = (int)((0.5 - lat / 180.0) * HEIGHT);
                        json += "["+x+", "+y+"],";
                    }
                }
                br.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
            json = json.substring(0,json.length()-1);
            json += "]}";
            return json;
        }

        /**
         * Simply return the JSON object:
         * {"data": [[235, 62],[242, 62],...]}
         */
        @Override
        protected void doGet( HttpServletRequest request,
                HttpServletResponse response)
                throws ServletException, IOException {
            PrintWriter out = response.getWriter();
            out.println(json);
        }
    }
```

Wir laden die Datei "Cities.txt" und speichern die Werte als JSON in einer Instanzvariable. Das machen wir in der *init()* Methode. Die *doGet()* Methode gibt dann einfach die Instanzvariable zurück. Was wir hier auch sehen, dass Servlet Programmierung eigentlich viel ähnlicher zu unserer normalen Art und Weise zu programmieren ist, als es etwas JSP Programmierung ist.

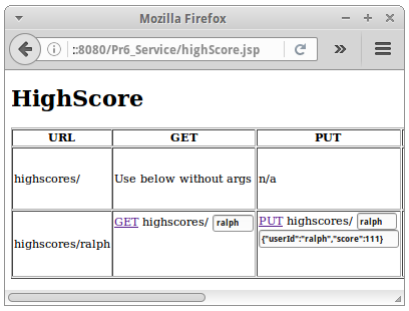


Wenn wir jetzt im Browser folgende URL aufrufen

```
http://localhost:8080/Pr6_Service/worldmap
```

erhalten wir die Geo-Daten im JSON Format.

Die Datei *worldMap.jsp* enthält das notwendige D3JS JavaScript, dass die Daten als Scatterplot anzeigt. Der JavaScript Code ist praktisch unverändert von Scott Murray's "Making a scatterplot" Tutorial [4].



HighScore Webservice

Wir wollen aus unserer HighScore Anwendung einen RESTful Webservice machen. Dazu fassen wir kurz die Anforderungen zusammen, die wir von unserem Webservice erwarten:

- die HighScores aller Spieler sortiert nach höchsten Score auflisten
- einen neuen HighScore für einen Spieler anlegen
- den HighScore eines Spielers ausgeben
- den HighScore eines Spielers verändern
- den HighScore eines Spielers löschen.

Wenn wir einen RESTful Webservice kreieren, durchlaufen wir immer die gleichen Schritte. Wir identifizieren nacheinander

1. die *Resourcen*, also URLs,
2. die HTTP *Methoden*, also GET, PUT, POST und/oder DELETE,
3. die *Repräsentationen*, z.B. JSON oder XML,
4. und die HTTP *Status Codes* die zurückgegeben werden.

Gehen wir die Punkte im Einzelnen durch.

1. Resourcen

Resourcen sind immer URLs. In der Regel werden aus den POJOs Resourcen. Da es in unserem Beispiel nur das HighScore POJO gibt, gibt es auch nur eine Resource:

```
/service/highscores/
```

In der Regel wählt man die Pluralform des POJOs.

2. Methoden

Was die Methoden angeht, da ist die Auswahl beschränkt: es gibt GET, PUT, POST und DELETE. Die entsprechen den CRUD Operationen die es in einer Datenbank gibt, also Create, Read, Update und Delete, und genügen in der Regel.

GET wird zum Lesen einer Resource verwendet und ist unschädlich, da sie nichts verändert. PUT wird verwendet um eine exitierende Resource zu verändern. Manchmal wird PUT auch verwendet um eine neue Resource anzulegen, aber normalerweise wird dafür POST verwendet. Und DELETE macht genau das, es löscht eine exitierende Resource. PUT, POST und DELETE sind gefährlich, da sie Änderungen vornehmen.

Meist nimmt man seine Anforderungen und fasst diese in eine Tabelle mit den Resourcen und den Methoden zusammen. Für unser HighScore Beispiel sieht das dann so aus, und gibt uns eine schöne Übersicht über den Webservice:

URL	GET	PUT	POST	DELETE
highscores/	HighScores aller Spieler listen	(HighScores aller Spieler ersetzen)	HighScore für einen Spieler neu anlegen	(HighScores aller Spieler löschen)
highscores/ralph	HighScore eines Spielers ausgeben	HighScore für einen Spieler verändern		HighScore eines Spielers löschen

3. Repräsentationen

Unter Repräsentation versteht man das Datenformat das ein Webservice versteht. In der Regel sollte es immer ein existierender Webstandard sein, muss es aber nicht. Häufig verwendete Formate sind z.B. Text, JSON, HTML, XML, aber auch GIF und andere MIME Formate sind möglich.

Nehmen wir an, wir möchten den Highscore des Nutzers "ralph". Wir würden dann einen GET Request an die Resouce "/service/highscores/ralph" schicken, und als Antwort das folgende JSON erhalten:

```
{"score":42,"userId":"ralph"}
```

Umgekehrt wenn wir einen neuen Highscore für den neuen Nutzer "fish" anlegen wollen, würden wir einen POST Request an die Resouce "/service/highscores/" schicken, und im Body des POST Requests wäre folgendes JSON erhalten:

```
{"userId":"fish","score":222}
```

Auch hier kann man alle verwendeten Repräsentationen in einer schönen Tabelle zusammenfassen:

URL	GET	PUT	POST	DELETE
highscores/			JSON	
highscores/ralph	JSON	JSON		

4. Status Codes

Bei den Status Codes geht es darum dem Client mitzuteilen ob alles funktioniert hat, oder ob etwas schief gelaufen ist. Hierzu verwendet REST die genz normalen HTTP Status Codes [5]. Wenn alles in Ordnung ist, dann werden die 200er verwendet, z.B.

- 200 OK
- 201 Created
- 204 No Content (also wie 200, aber im HTTP Body ist nix)

Wenn etwas schief gelaufen ist, gibt es die 400er:

- 400 Bad Request
- 404 Not Found
- 405 Method Not Allowed
- 409 Conflict (resource already exists)

Wenn etwas total schief gelaufen ist, also am Server, z.B. Datenbank kaputt oder Meteoriteneinschlag, dann gibt es die 500er:

- 500 Internal Server Error

Am besten schauen wir uns mal an wie wir die Status Codes in unserem Beispiel verwenden würden:

URL	GET	PUT	POST	DELETE
highscores/	200 OK 404 Not Found	405 Method Not Allowed	201 Created 409 Conflict (resource exists)	405 Method Not Allowed
highscores/ralph	200 OK 404 Not Found	204 No Content 404 Not Found	405 Method Not Allowed	204 No Content 404 Not Found

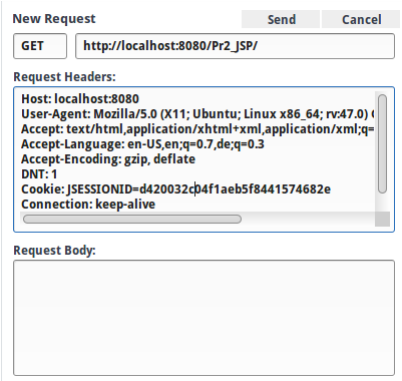
Jersey

Nachdem wir ja die Datenbank schon im letzten Kapitel aufgesetzt haben, ist die Implementierung des Webservices eigentlich ganz einfach, wenn man das Jersey Framework verwendet.

```
@Path("/highscores")
public class HighScoresResource {

    public HighScoresResource() {
        super();
    }

    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public List<HighScore> getHighScores() {
        return HighScoreDao.getInstance().findAll();
    }

    @GET
    @Path("/{param}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getHighScore( @PathParam("param") String userId) {
        HighScore hs = HighScoreDao.getInstance().findById(userId);
        if (hs != null) {
            return Response.status(200).entity(hs).build();
        } else {
            return Response.status(404).entity(
                "Unknown userId.").build();
        }
    }

    @PUT
    @Path("/{param}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response putHighScore(
                    @PathParam("param") String userId, HighScore hs) {
        if ((userId != null) && (hs != null)) {
          if (userId.equals(hs.getUserId())) {
            if ( HighScoreDao.getInstance().findById(userId)!=null) {
                HighScoreDao.getInstance().merge(hs);
                String result = "HighScore updated: "+hs.getUserId();
                return Response.status(204).entity(result).build();
            }
          }
        }
        return Response.status(404).entity("Not Found").build();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response postHighScore(HighScore hs) {
      if ( HighScoreDao.getInstance().findById(hs.getUserId())==null){
          HighScoreDao.getInstance().save(hs);
          String result = "HighScore created: " + hs.getUserId();
          return Response.status(201).entity(result).build();
      }
      return Response.status(409).entity(
                        "Conflict (resource already exists)").build();
    }
```

```
    @DELETE
    @Path("/{param}")
    public Response deleteHighScore(
                                @PathParam("param") String userId) {
        if ( HighScoreDao.getInstance().findById(userId) != null) {
            HighScore hs =
                HighScoreDao.getInstance().findById(userId);
            HighScoreDao.getInstance().delete(hs);
            String result = "HighScore deleted: " + hs.getUserId();
            return Response.status(204).entity(result).build();
        }
        return Response.status(404).entity("Not Found").build();

    }
}
```

Testen



Das Testen des Webservice gestaltet sich etwas schwieriger, aber kann entweder mit der Firefox Webconsole erfolgen, wie z.B. rechts zu sehen, oder mittels JavaScript.

Das JavaScript ist in der Datei *webservice.js*, die im Header der *highScore.jsp* Datei geladen wird. Wenn man etwas JavaScript versteht, ist sie gar nicht so schwer. Wir zeigen nur die beiden Teile für einen GET Request. In der *highScore.jsp* Datei steht dort:

```
...
<a href="#" onclick="sendGetRequest(
    'service/highscores/');return false;">GET</a> highscores/
<input type="text" size="5" id="urnGET" value="ralph"><br/><br/>
<span id="responseGET"> </span><br/>
<span id="statusCodeGET"> </span>
...
```

d.h. bei einem Klick auf den Link, wird die JavaScript Methode *sendGetRequest()* aufgerufen. Die ist wiederum in der Datei *webservice.js* definiert:

```
function sendGetRequest(url) {
    var urn = document.getElementById("urnGET").value;
    url = url + urn;
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == XMLHttpRequest.DONE) {
            document.getElementById("responseGET").innerHTML =
                xmlhttp.responseText;
            document.getElementById("statusCodeGET").innerHTML =
                xmlhttp.status;
        }
    }
    xmlhttp.open("GET", url, true);
    xmlhttp.send();
}
```

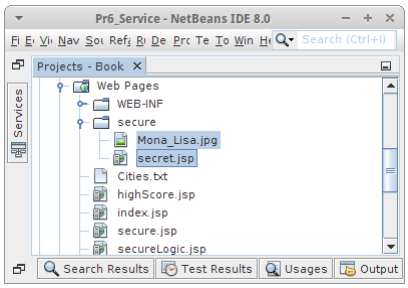
Das Ganze ist eigentlich nur zum Testen gedacht, ansonsten ist es eher nutzlos.

Secure

Bei der Secure Anwendung verwenden wir einen Filter, um den Zugriff auf das /secure/ Verzeichnis unserer Webanwendung zu schützen. Nur Anwender die sich eingeloggt haben, dürfen auf Mona_Lisa.jpg oder secret.jsp zugreifen. Um das einloggen etwas einfacher zu machen, genügt es auf den Link zu klicken, dann wird *secureLogic.jsp* aufgerufen,

```
<%
    session.setAttribute("User", "blahblah");
    response.sendRedirect("secure.jsp");
    return;
%>
```

die nichts anderes macht als uns "einzuloggen", also das Attribut "User" im *session* Objekt zu setzen.



Der Filter ist wieder überraschend einfach:

```
public class SecurityFilter implements Filter {
    private FilterConfig config;

    @Override
    public void init(FilterConfig filterConfig)
            throws ServletException {
        this.config = filterConfig;
    }
    @Override
    public void doFilter(ServletRequest request,
            ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest httpReq = (HttpServletRequest) request;
        HttpServletResponse httpResp = (HttpServletResponse) response;
        HttpSession session = httpReq.getSession();

        String user = (String)session.getAttribute("User");
        if ( user != null  ) {
            chain.doFilter(request, response);
        } else {
            httpResp.sendRedirect("../index.jsp");
            return;
        }
    }
    @Override
    public void destroy() {
        // does nothing
    }
}
```

Wenn wir den SecurityFilter mit unserer Login Anwendung aus dem letzten Kapitel verbinden, dann sehen wir, dass es jetzt auf einmal viel einfacher geworden ist eine Anwendung sicher zu machen, denn es ist nicht mehr nötig jede Seite individuell mit den Zeilen

```
<%
    User usr = (User)session.getAttribute("Login.User");
    if ( usr == null ) {
        response.sendRedirect("login.jsp");
        return;
    }
%>
...
```

zu beginnen, sondern es genügt alle zu schützenden Seiten und Dateien in das Verzeichnis /secure/ zu stecken.

Fragen

1. Vergleichen Sie die Technologien Java Servlets und Java Server Pages (JSP). Geben Sie je ein Beispiel wofür sich die eine besser eignet als die andere.

2. Bei der Gestaltung eines RESTful Web-Services folgt man in der Regel mehreren Schritten: Zunächst ermittelt man, welche Ressourcen benötigt werden (URI), dann muss man feststellen, welche der HTTP Methoden umgesetzt werden sollen, und schließlich muss man angeben, welche Representation für jede Ressource und Methods verwendet wird (z.B. JSON). Anhand einer einfachen Version von Yahoo Finance definieren Sie bitte die Ressourcen, Methoden und Repräsentation die nötig sind, um die folgenden Anforderungen abzubilden. Ein authenifizierter Benutzer sollte
   a) alle Aktien die man kaufen oder verkaufen kann auflisten
   b) das Depot des Benutzers auflisten, also eine Liste aller Aktien die der Benutzer besitzt
   c) den momentanen Preis einer Aktie abfragen können
   d) eine bestimmte Anzahl einer Aktie kaufen können
   e) eine bestimmte Anzahl einer Aktie verkaufen können

3. Bei der Gestaltung eines RESTful Web-Service, folgt man in der Regel mehreren Schritten: Zunächst ermittelt man, welche Ressourcen benötigt werden (URI), dann muss man feststellen, welche der HTTP Methoden umgesetzt werden sollen, und schließlich muss man angeben, welche Representation für jede Ressource und Methods verwendet wird (z.B.JSON). Anhand einer einfachern Version von Facebook definieren Sie bitte die Ressourcen, Methoden und Repräsentation die nötig sind, um die folgenden Anforderungen abzubilden:
   a) es können neue User angelegt und gelöscht werden
   b) jeder User hat einen Status, der gelesen und geändert werden kann
   c) jeder User hat eine List von Freunden, zu der neue Freunde hinzugefügt werden können und Freunde entfernt werden können.

4. Wofür ist die Firefox Web Console gut?

5. Das HTTP Protokoll verwendet Status-Codes. Bitte erläutern Sie die fünf verschiedene Arten von Status-Codes (wie z.B. einen 404) und geben Sie für jeden je ein Beispiel, d.h., in welchem Zusammenhang ein solcher Code auftreten könnte.

Referenzen

Anbei sind die Referenzen zu diesem Kapitel. Eine sehr schöne Zusammenfassung von RESTful Webservice s ist Referenz [6].

[1] Introducing JSON, www.json.org

[2] Jersey, RESTful Web Services in Java, https://jersey.java.net/

[3] Scott Murray, Making a bar chart, http://alignedleft.com/tutorials/d3/making-a-bar-chart

[4] Scott Murray, D3 Tutorials, alignedleft.com/tutorials/d3/

[5] List of HTTP status codes, https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

[6] Roger L. Costello, Building Web Services the REST Way, www.xfront.com/REST-Web-Services.html

Appendix

Projects

Falls man noch nach anderen Ideen für mögliche Projekte sucht wollen wir im folgenden einige auflisten, die man ohne große Problem mit dem Gelernten realisieren kann.

Temperature

Ein Programm das Fahrenheit in Celsius umrechnet. Der Nutzer gibt die Temperatur in Fahrenheit ein und mittels der Formel

```
int c = (int) ( (5.0 / 9.0) * (f - 32) );
```

können wir die Temperatur in Celsius umrechnen.

LeapYear

Mit der Formel aus dem ersten Semester können wir feststellen ob ein Jahr ein Schaltjahr ist. Der Nutzer gibt ein Jahr ein, vielleicht sein Geburtsjahr, und wir geben dann aus, ab es sich bei dem Jahr um ein Schaltjahr handelt.

YearlyRate

Es geht darum zu berechnen wie hoch die jährliche Rate ist, wenn man einen Kredit nach einer gewissen Laufzeit zu einem gewissen Zinssatz abbezahlt haben will. Karel will sich ein Auto kaufen, deswegen hat er angefangen zu sparen. Er will sich einen Mini kaufen (Mercedes ist zu teuer), und er hat einen gebrauchten für 5000 Euro gesehen. Er hat einen günstigen Kredit bei einer Bank von 5% pro Jahr gesehen. Wie hoch ist seine jährliche Rate, wenn er den Kredit in 5 Jahren abbezahlt haben will? Karel braucht also ein Programm bei dem er die Kreditsumme (k) eingeben kann, den Zinssatz (z) und die Laufzeit in Jahren (n). Das Programm soll ihm dann sagen wie hoch seine jährliche Rate (y) ist:

```
double q = 1.0 + z;
double qn = Math.pow(q, n);    // q^n
double y = k * qn * (q-1) / (qn-1);
```

Die Formel dafür kann man in der Wikipedia unter Sparkassenformel nachsehen.

Roulette

In einer vereinfachten Version von Roulette spielt ein Spieler gegen den Computer. Der Spieler kann auf *odd* oder *even* (oder *high* oder *low*) setzen. Der Computer wählt dann eine Zufallszahl zwischen 0 und 36. In einem zweiten Schritt, könnte man noch mit Credits spielen. Der Spieler erhält am Anfang 100 Credits, und falls er gewinnt erhält er 10 zusätzliche Credits, falls er verliert werden 10 Credits abgezogen. Schließlich könnte man auch noch ein MultiPlayer Spiel daraus machen.

BlackJack

Auch Black Jack können wir in einer etwas vereinfachten Version implementieren. Anstelle von Karten verwenden wir einfach Zahlen, und zwar Zahlen zwischen 1 und 11. Der Computer spielt den Croupier und beginnt indem er eine Zufallszahl zwischen 17 und 25 erzeugt. Dann ist der Spieler an der Reihe. Dieser fängt mit einer Karte an, also eine Zufallszahl zwischen 1 und 11. Er kann dann entscheiden ob er noch eine Karte möchte. Falls ja, wird wieder eine Zufallszahl zwischen 1 und 11 erzeugt und zur momentanen "Hand" hinzuaddiert. Wenn der Spieler keine neue Karte mehr haben möchte, wird die "Hand" des Spielers mit der des Computers verglichen. Gewonnen hat derjenige der 21 Punkte oder weniger hat und mehr als der andere. Ansonsten ist es ein Unentschieden.

BattleShip

Im ersten Semester haben wir den Spieleklassiker Schiffeversenken umgesetzt. Ähnlich wie TicTacToe oder Mines können wir das auch als Webanwendung realisieren.

TicTacToe - Two Players

In Kapitel vier haben wir die Single-Player Variante implementiert. Wie wäre es mit eine Zwei-Spieler Variante? Wahrscheinlich macht es vorher Sinn sich den TwoPlayer Chat noch einmal anzusehen. Ähnlich könnte man auch andere Spiele wie Dame, Schach und Go implementieren.

ESP Game

Das ESP Spiel wurde von Luis van Ahn, Professor an der Carnegie Mellon University, erfunden. Es handelt sich um eine TwoPlayer Game um Bilder zu klassifizieren. Um zu sehen worum es geht, sollte man seine Präsentation im Rahmen der Google TechTalks ansehen: https://www.youtube.com/watch?v=tx082gDwGcM

Translate

Google Translate hat auch einfach angefangen. In einer einfachen Version könnte man basierend auf dem Dictionary Beispiel, einen Internet-Übersetzungsservice anbieten. Einmal mit einem Webinterface, dass eingegebenen Text einfach Wort für Wort übersetzt. Daraus könnte man aber auch einen Web Service machen. Später könnte man auch eine Version schreiben, in der Nutzer neue Wörter hinzufügen könnten.

SpellChecker

Ein Service der die Rechtschreibung überprüft. Man würde das auf dem Dictionary Beispiel aufbauen, eine einfache Version würde einfach auf einem HashSet basieren. Wahrscheinlich wäre eine englische Version einfacher. Evtl. sind die folgenden beiden Resourcen hilfreich: Aspell, http://freedict.org/de/, http://sourceforge.net/projects/freedict/files/ und stackoverflow.com/questions/2294915/what-algorithm-gives-suggestions-in-a-spell-checker.

Eliza

Im ersten Semester haben wir den Psychiater Eliza kennengelernt. Natürlich könnte man daraus auch eine Webanwendung machen.

ToDo List

Eine einfach ToDo Liste in der jeder Nutzer seine eigenen ToDos listen kann. Evtl. wäre auch eine Priorisierung wünschenswert.

Calendar

Die Anwendung soll es einem Nutzer erlauben seinen Kalendar zu managen. Dazu gehört neue Einträge anlegen, löschen und auch die Ausgabe des Kalanders, evtl. mit Druckfunktion. Eine spätere Version könnte auch Email Reminders beinhalten. Auf jeden Fall sollte auch ein Web Service existieren, um eine spätere Anbindung an ein mobiles Endgerät zu ermöglichen.

PhoneBook

Es geht darum seine Adressen und Telefonnummern zu managen. Auch hier sollte wieder ein Web Service existieren, um eine spätere Anbindung an ein mobiles Endgerät zu ermöglichen. Es sollte auch möglich sein beliebige Key-Value Pairs zu speichern. Und eine Suche wäre auch praktisch.

TimeSheet

Wenn man als Freelancer arbeitet oder auch in manchen Firmen, muss man aufschreiben wie viel Zeit man für welches Projekt verwendet hat. Dies soll eine kleine Webanwendung werden, in der Nutzer ganz einfach ihre Zeiten managen können. Ein Web Service Interface wäre bestimmt auch nicht schlecht.

StoreSecrets

Heutzutage hat jeder zig Passwörter und PINs die man sich merken muss. Und kann man den Anwendungen die man da so im Internet oder auf dem Handy hat wirklich trauen? Deswegen nach dem Motto "Max Selber" wollen wir eine Webanwendung zum speichern von Passwörtern und anderen Geheimnissen schreiben.

Votes

Die Idee hinter Votes ist eine Art Umfrage, die es erlaubt mehrere Nutzer bzgl. eines bestimmte Themas abzustimmen. Sollte per Einladung über einen Weblink oder QR Code funktionieren.

Who Has My Stuff

Wenn man wie ich vielen Leuten viele Sachen ausleiht verliert man leicht den Überblick wem man was geliehen hat. Hier könnte eine einfache Webapp Abhilfe schaffen.

Library

Wir haben im ersten Semester die Anforderungen für eine einfache Bibliothek erarbeitet. Jetzt könnten wir selbige als Webanwendung umsetzen.

University

Auch aus dem ersten Semester stammt die Idee eines universitären Studenten-Management Systems. Es geht einfach darum den Fortschritt von Studierenden während des Studiums zu managen.

Doodle

Bei der Webanwendung Doodle (doodle.com) geht es darum gemeinsame Termine für Meetings und Ähnliches zu finden. Es dürfte kein Problem sein eine ähnliche Anwendung mit dem hier gelernten umzusetzen.

Flickr

Flickr (flickr.com) ist eine Webanwendung zum Hochladen und Verwalten von Bildern. Jeder Nutzer sollte seinen eigenen Bereich haben, es sollte aber auch einen gemeinsamen Bereich geben. Bilder können privat, aber auch öffentlich sein. Und es könnte ein Ranking/Voting System geben mit dem man Bilder bewerten kann. Natürlich sollte es auch als Webservice zur Verfügung stehen um mobile Geräte direkt einbinden zu können.

Facebook

Inzwischen kennt Facebook (facebook.com) fast niemand mehr, aber die Idee war eigentlich ganz cool: seinem zukünftigen Arbeitgeber seine schlimmsten Seiten zu zeigen. Im Prinzip ist es ein ähnliches Konzept wie unsere Chirpr Anwendung. Es gibt Nutzer, Freunde und Gruppen, und jeder kann News posten. Das Ganze kann man dann zeitlich sortieren und nennt es Timeline.

Stack Overflow

Stack Overflow (stackoverflow.com) ist ein Beispiel für eine spezialisierte Form der klassischen NewsGroup. Ein Nutzer stellt eine Frage, idealerweise mit einigen Keywords assoziiert, und andere Nutzer können die Frage beantworten. Man kann dann noch ein Votingsystem einführen, etc., etc.

Shop

Ein einfacher Shop, z.B. für Bücher, CDs oder Lebensmittel. In einer einfachen Version ist das Inventar fest vorgegeben. Der Nutzer kann aber Sachen in einen Warenkorb legen und kaufen. In einer späteren Version könnte es auch ein Interface für den Händler geben, damit er neues Inventar anlegen kann, etc.

Bugs

Ein Tool zum Melden von Bugs, also Fehlern in einem Programm. Die Bugs bekommen eine Nummer, werden dann einem Mitarbeiter zugeordnet, und nachdem sie bearbeitet wurden, bekommt der Melder eine Nachricht. Man sollte u.a. alle offenen Bugs auflisten können. Das Ganze sollte natürlich für mehrere Projekte möglich sein.

Bank

Die Idee hier ist eine Spiel-Bank Anwendung zu schreiben. Kunden haben Konten und können einander Geld überweisen. Das ist eine sehr interessante Anwendung, denn beim Überweisen von Geld kann verdammt viel schief gehen. Da muss man viel nachdenken, damit auch wirklich nichts schief geht. Der nächste Schritt wäre dann sich ein bisschen in Bitcoins einzulesen (z.B. *Bitcoin and Cryptocurrency Technologies*).

Hospital

In einem Krankenhaus gibt es Patienten und Doktoren. Jeder Patient hat eine Krankenakte, und die Doktoren dürfen da drin rumschreiben.

Recipies

Es geht um das Sammeln von Rezepten. Dabei könnte jeder Nutzer seine eigenen Rezepte haben, es könnte aber auch die Möglichkeit geben, das Rezepte global sind, also für jeden einsehbar. Natürlich sollten Rezepte auch Bilder haben, sonst schmecken sie nicht so gut. Man könnte sich auch Erweiterungen ausdenken, wir ein Voting System.

Classifieds

Bei Kleinanzeigen geht es darum irgendetwas zu verkaufen, anzubieten oder zu finden. Hier gibt es Nutzer und Anzeigen. Anzeigen sollten einen Titel, eine Kategorie, einen Text und ein Verfallsdatum haben. Das Ganze sollte natürlich durchsuchbar sein. Was Kleinanzeigen auch ausmacht ist, dass sie meist lokal sind, deswegen könnte man in einer Erweiterung auch noch den Ort/Lokation mit aufnehmen.

Document / Image Management System

Ein einfaches Dokumenten-Managementsystem erlaubt es einem auf seine Dokumente über ein Webinterface zuzugreifen. Dokumente sollten nach Namen, Schlüsselwörtern und evtl. Inhalt klassifiziert werden, damit man sie auch durchsuchen kann. Ein WebService zur Anbindung mobiler Endgeräte könnte sinnvoll sein.

Crossword Puzzle

Im zweiten Semester haben wir gesehen wie wir Kreuzworträtsel mit Hilfe der *Trie* Datenstruktur erzeugen können. Was uns aber fehlt sind die Inhalte. In diesem Projekt, sollen Nutzer Fragen und Antworten für Kreuzworträtsel eingeben können, und daraus dann Kreuzworträtsel generieren lassen können. Die Fragen sollten öffentlich sein, damit jeder sie nutzen kann. Evtl. macht es bei jeder Frage einen Schwierigkeitsgrad mit anzugeben. Auch ein Voting System vielleicht inspiriert von Stack Overflow könnte hilfreich sein.

Newspaper

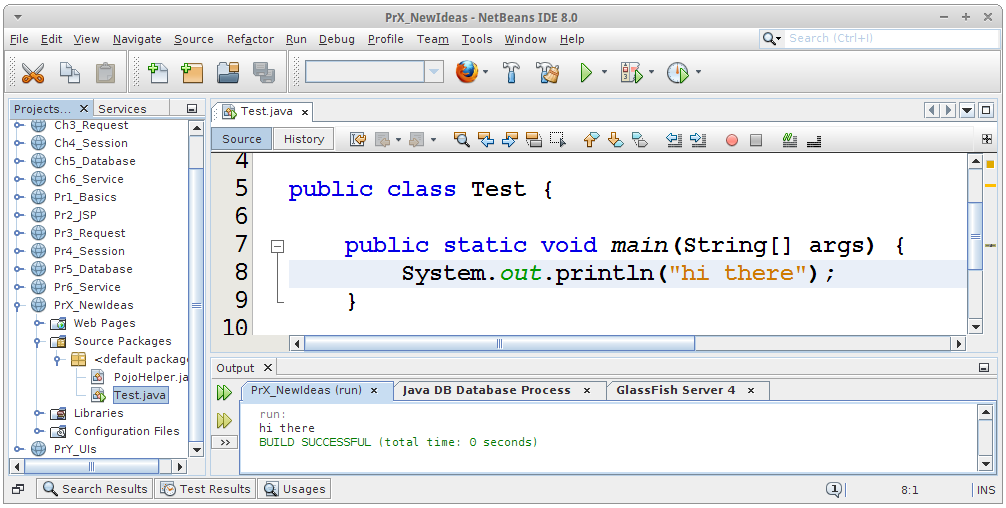
Bei Newspaper geht es um eine Zeitung, z.B. eine Mitarbeiter- oder Vereinszeitung. Es gibt Autoren und Artikel, sowie Ausgaben. Evtl. macht es auch Sinn einen Administrator zu haben. Auf jeden Fall sollte es die Möglichkeit geben eine gedruckte Version der Zeitung zu erstellen. Hierfür gibt es in CSS die @media Regel "@media print".

NetBeans

Sobald man sich auf eine Entwicklungsumgebung (IDE) festlegt, und womöglich sogar noch eine Versionsnummer nennt, ist ein Buch automatisch datiert, und zwei Wochen später eigentlich nur noch zum Heizen nützlich. Aber wir wollen es trotzdem wagen.

Download and Installation

Der erste Schritt ist NetBeans vom Internet zu laden [1], und auf dem eigenen Rechner zu installieren. Es gibt da mehrere verschiedene Versionen, einmal für verschiedene Betriebssysteme, da sollten wir die Version die für unseren Rechner geeignet ist nehmen. Aber es gibt auch verschiedene NetBeans IDE Download Bundles und wir wollen das "Java EE" Bündel. NetBeans benötigt die Java JDK Version 7 oder später, falls wir die noch nicht auf unserem Rechner haben, müssen wir die vorher noch installieren. Ansonsten ist NetBeans ganz ähnlich wie Eclipse:



Noch zwei kleine Anmerkungen: immer die englischen Versionen von Software installieren. Die funktionieren wenigstens. Und folgendes zu Updates: da ich nur für mich selbst programmiere und nicht in einer Firma, vermeide ich Updates. Denn solange alles funktioniert gibt es keinen Grund das zu ändern, und ein Update könnte das. Nun in einer Firma ist das etwas anderes, man sollte aber damit rechnen, dass nach einem Update erstmal ein bis zwei Tage gar nichts mehr geht. In der Firma ist das o.k., da wird man ja bezahlt.

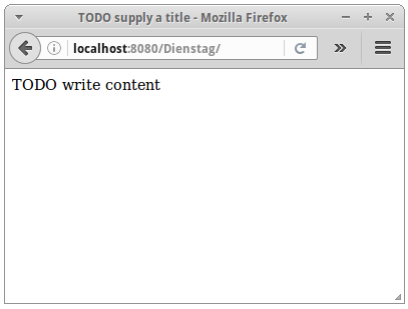
Get to Know NetBeans

Bevor man mit den Websachen anfängt sollte man mal ein kleines Konsolenprogramm schreiben oder eine kleine Swing Anwendung. Neue Projekte legt man mit "File -> New Project" an, und Java Projekte dann über "Java -> Java Application". Wir können dann mal eine einfache Java Klasse wie oben anlegen. Ausführen kann man die über das große grüne Dreieck oder per "Run -> Run Project".

Etwas gewöhnungsbedürftig sind die NetBeans Shortcuts. Die gehen auch über "Ctrl-Space", aber "System.out.println()" ist jetzt "sout" und "public static void main()" ist jetzt "psvm". Aber die Shortcuts gibt's als pdf zum Download [2].

NetBeans hat auch ganz viele Tutorials. Die meisten taugen nichts, aber besser als Eclipse alle mal, denn da gibt's gar keine. Das Swing Tutorial [3] ist ganz o.k. wenn man in der U-Bahn sitzt.

Web Project



Um unser erstes Webprojekt anzulegen, gehen wir wieder über "File -> New Project" dieses mal wählen wir aber unter "Java Web" die "Web Application" aus und klicken auf Next. Dann geben wir dem Projekt einen Namen, z.B., "Dienstag" und klicken wieder auf Next. Bei der Serverauswahl würde ich "GlassFish Server 4" verwendet. Ganz selten macht der aber Ärger, dann kann man auch den Tomcat nehmen, geht auch. Dann dürfen wir auf "Finish" klicken, auch wenn wir kein Finisch können. Es wird ein Projekt angelegt und eine HTML Seite namens "index.html". Gestartet wird das Projekt wie oben auch über "Run -> Run Project" oder das grüne Dreieck.
Es dauert dann ein bisschen, aber früher oder später müsste sich ein Webbrowser, meist Firefox, öffnen und die Seite "index.html" anzeigen.

Warum "dauert" es ein bisschen? Das hat damit zu tun, dass mehrere Schritte durchlaufen werden bis man das Resultat im Browser bewundern kann:

1. der Server, GlashFish oder Tomcat, wird gestartet, es sei denn er läuft bereits
2. das Projekt wird verpackt, auf den Server hochgeladen (in der Regel einfach kopiert) und deployed [6]
3. der Browser wird gestartet, falls er noch nicht läuft
4. der Browser muss die Seite vom Server laden.

Debugger

NetBeans hat auch einen sehr mächtigen Debugger, der auch für Webseiten funktioniert. Dazu muss man zunächst einen Breakpoint in seinem JSP oder Servlet Code setzen (geht nicht mit HTML). Und danach startet man das Projekt im Debug-Modus entweder über "Debug -> Debug Project" oder dem Knopf rechts neben dem grünen Dreieck.

Um den Debugger einmal zu testen, schreiben wir ein paar einfache Zeilen JSP Code. Zum Anlegen einer neuen JSP Datei, klicken wir mit der rechten Maustaste auf das Verzeichnis "Web Pages" in unserem "Dienstag" Projekt, wählen dann "New -> JSP" aus, und nennen die neue Seite "hello". Das ".jsp" wird automatisch angehängt. Dann löschen wir alles was in der Datei evtl. schon steht und fügen die folgenden fünf Zeilen ein:

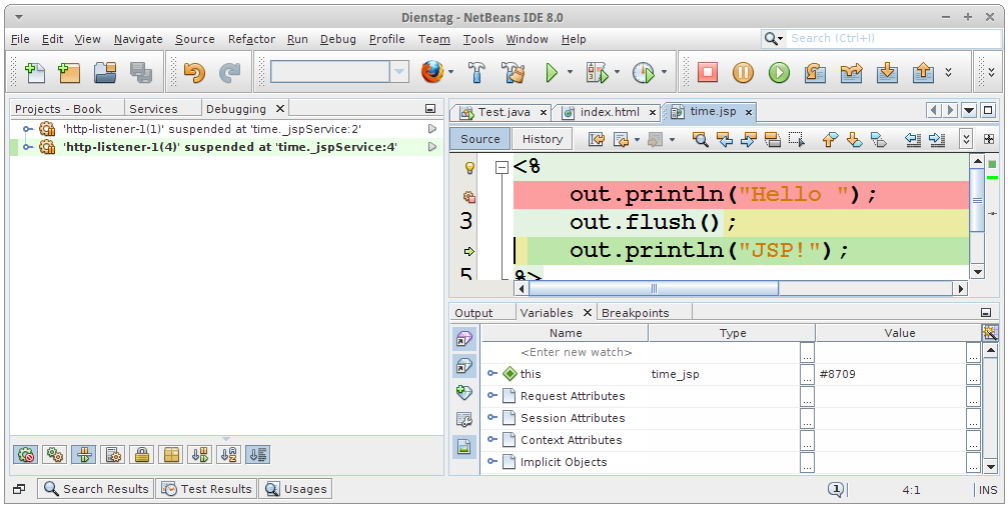
```
<%
    out.println("Hello ");
    out.flush();
    out.println("JSP!");
%>
```

Dann setzen wir einen Breakpoint in der Zeile zwei (out.println()) indem wir einfach einmal mit der linken Maustaste auf die "2" der Zeilennummer klicken. Wir starten dann das Projekt im Debug-Modus und es sollte ein Browser sich öffnen. Der Browser lädt zunächst die "index.html" Seite, wir müssen ihm erst sagen dass er zur

```
http://localhost:8080/Dienstag/hello.jsp
```

Seite gehen soll. Sobald wir das tun, sehen wir dass in NetBeans die zweite Zeile grün markiert ist, und wir jetzt auf alle möglichen Knöpfe drücken können. Der Browser allerdings zeigt erst mal gar nichts an, denn der wartet geduldig auf sein Antwort. Wenn wir jetzt mit "Step over (Shift-F8)" zwei Schritte weitergehen, dann erscheint im Browser das "Hello" aber noch nicht das "JSP".

Lustig wird es wenn wir jetzt einen zweiten Browser öffnen, und dort auch zu besagter Seite gehen. Auch die bleibt erst mal im Debugger hängen und wir sehen beide Threads im linken Fenster "Debugging":



Wir können hier zwischen den zwei Threads hin und herspringen, und mal den einen ein paar Schritte laufen lassen und dann den anderen wieder. Man kommt sich ein bisschen wie ein Marionettenspieler vor. Obwohl das wie eine Spielerei anmutet, wird das extrem wichtig werden wenn mal "lustige" Sachen passieren, deren Ursache mit Threading zu tun hat.

Load Generation

NetBeans hat alle möglichen coolen Sachen, wie z.B. einen Profiler den man mit dem Knopf neben dem Debugger Knopf startet. Der hat dann noch mehr Knöpfe die man drücken kann, und da gibt es dann ganz viele Graphiken mit denen man seine Manager beeindrucken kann.

Was aber viel nützlicher ist, das ist der eingebaute Load Generator, mit dem man Last auf seinem Server verursachen kann. Ein Tutorial das zeigt wie das geht findet sich auf dem NetBeans Website [4]. Es sei allerdings angemerkt, dass das Tutorial ohne ein jMeter Script nutzlos ist, deswegen vielleicht erst mal hier [5] nachsehen wie man ein jMeter Script erzeugt.

Referenzen

Zu NetBeans gibt es ganz viele Tutorials, die man ganz einfach über Google findet. Die meisten sind auf dem NetBeans eigenen Website. Anbei noch ein, zwei nützliche Links.

[1] NetBeans IDE, https://netbeans.org/

[2] Highlights of NetBeans IDE 8.0 Keyboard Shortcuts & Code Templates, https://netbeans.org/project_downloads/usersguide/shortcuts-80.pdf

[3] Learning Swing with the NetBeans IDE, (http://java.sun.com/docs/books/tutorial/uiswing/learn/index.html

[4] Using a Load Generator in NetBeans IDE netbeans.org/kb/docs/java/profile-loadgenerator.html

[5] Apache JMeter HTTP(S) Test Script Recorder, jmeter.apache.org/usermanual/jmeter_proxy_step_by_step.pdf

[6] Tomcat Web Application Deployment, https://tomcat.apache.org/tomcat-7.0-doc/deployer-howto.html

JavaScript

JavaScript ist ein wenig wie Englisch: oberflächlich scheint es eine sehr einfache Sprache zu sein, und fast jeder kann kleine Programme mit JavaScript schreiben. Aber wie mit dem Englischen ist das sehr trügerisch. Wir werden hier aber nur kurz die oberflächlichen Aspekte der JavaScript Sprache ansprechen. Eine kurze Warnung: obwohl JavaScript das Wort "Java" im Namen trägt, hat es eigentlich gar nichts mit Java zu tun.

HTML

Das JavaScript das uns interessiert ist jenes das im Browser läuft. Deswegen gehen wir immer davon aus, dass es irgendeine HTML Datei gibt die entweder den JavaScript Code direkt enthält, oder einen Link auf eine JavaScript Datei.

Im Prinzip können wir JavaScript sowohl im Body Teil des HTMLs schreiben,

```
<html>
    <head>
    </head>
    <body>
<script type="text/javascript">
    document.write("This message is written by JavaScript");
</script>
    </body>
</html>
```

als auch im Header:

```
<html>
    <head>
        <script type="text/javascript">
function message() {
    alert("This alert box was called with the onload event");
}
        </script>
    </head>
    <body onload="message()">
    </body>
</html>
```

Bevorzugt ist letzteres, aber im Prinzip kommt es wie immer auf die genauen Umstände an. Das Problem mit dem direkten Einfügen von Code ist aber allgemein, dass man ihn nicht wiederverwenden kann. Deswegen ist es eigentlich bevorzugt das JavaScript in eine externe Datei auszulagern:

```
<head>
    <script type="text/javascript" src="myJavaScript.js"></script>
</head>
<body>
</body>
</html>
```

Die kann dann nämlich auch in anderen Seiten wiederverwendet werden.

Basics

JavaScript ist dem Augenschein nach sehr ähnlich zu Java oder anderen Programmiersprachen. So gibt es Statements, wie z.B.

```
document.write("Hello Dolly");
```

und auch Variablen,

```
var x;
var carName = "Mini";
```

Aber vieles ist in JavaScript optional: so sind die Strichpunkte am Ende eine Statements optional, und auch die Deklaration ist optional. Also auch

```
y = x - 5
carName2 = "Mercedes"
```

ist vollkommen korrekt. Auch sehen wir keine Datentypen: JavaScript ist "weakly typed", d.h. die Datentypen werden zur Laufzeit bestimmt. Das ist zwar ganz praktisch, aber nicht ganz ungefährlich.

In JavaScript gibt es auch Funktionen und Objekte, allerdings keine wirklichen Klassen. Was wirklich toll ist, dass wir Funktionen auch basierend auf Events aufrufen können, z.B., verwenden wir im Services Kapitel den *onclick* Event eines Links,

```
<body>
    <a href="#" onclick="sendGet Request (' service / highscore s/');
            return false;">
        GET</a>
    <span id=" response GET"> </span><br/>
</body>
```

um JavaScript auszuführen. Und JavaScript kann auf HTML Elemente zugreifen, diese verändern, sogar neu kreieren und existierende löschen. Z.B., würde das JavaScript

```
document.getElementById(" response GET").innerHTML = "hi there";
```

in dem Tag oben den Text "hi there" einfügen.

Mit dieser kleinen Einführung und unseren existierenden Java Kenntnissen, müssten wir jetzt eigentlich den JavaScript Code den wir an verschiedenen Stellen im Buch verwenden verstehen können.

Referenzen

Wenn es um das klassische JavaScript geht, also keine Libraries, dann ist der Klassiker von David Flanagan wohl immer noch eine gute Quelle der Inspiration. Natürlich die papierlose Generation möchte auch bedient werden, hier sollte wie immer W3Schools nützliche Dienste erweisen.

[1] JavaScript: The Definitive Guide von David Flanagan

[2] JavaScript Tutorial - W3Schools, www.w3schools.com/js/

Epilogue

Das war's.

Index

Q

R

S

T

U

V

W

X