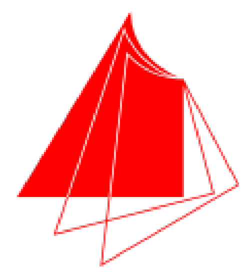
Hochschule Karlsruhe
Technik und Wirtschaft
Fakultät Informatik



Studienarbeit

JavaServer Faces und AJAX
versus
PHP und AJAX

vorgelegt von: Christian Kolodziej (Matrikelnummer 20330)
Rheinauer Ring 68/15
76437 Rastatt

betreuender Professor: Prof. Dr. Holger Vogelsang

Abgabetermin: 20. März 2007

Inhaltsverzeichnis

1.Motivation....3

2. Verwendete Techniken....4
  2.1 Ajax....4
  2.2 JavaServer Faces....5
    2.2.1 Dateien und Struktur einer JSF-Anwendung....5
    2.2.2 Die Umsetzung des MVC-Patterns....5
    2.2.3 Eingabevalidierung....6
    2.2.4 Navigation....7
    2.2.5 Internationalisierung....8
    2.2.6 Daten in Backing-Beans verwalten....9
    2.2.7 Internationalisierung innerhalb einer Backing-Bean....10
    2.2.8 Daten einfach darstellen mit h:dataTable....11
    2.2.9 Erweiterbarkeit....12
  2.3 Das JSF-AJAX-Framework „Ajax4jsf“....12
    2.3.1 Eine erste AJAX-Verwendung....13
    2.3.2 h:commandButton vs. a4j:commandButton....13
    2.3.3 Fortgeschrittene AJAX-Funktionen....14
  2.4 PHP....14
    2.4.1 XML-Datei als Antwort senden....14
    2.4.2 Internationalisierung....15
  2.5 Die Ajax-Bibliothek „Prototype“....15
    2.5.1 Wichtige Prototype-Funktionen....15
    2.5.2 Asynchrone Kommunikation mit Prototype....16
    2.5.3 AJAX-Antwort verarbeiten....17
    2.5.4 Asynchroner Submit....17

3. Realisierung einer Beispielapplikation....18
  3.1 Die Beispielapplikation – MuchToDo....18
    3.1.1. Neuregistrierung....18
    3.1.2 Passwort-vergessen-Funktion....19
    3.1.3 Der Login....19
    3.1.4 Die Verwaltungsoberfläche....19
    3.1.5 Gleiche Voraussetzungen: die CSS-Dateien....19
    3.1.6 Die Datenbankstruktur....20
  3.2 Realisierung mit JSF und Ajax4jsf....20
    3.2.1 Die Entwicklungsumgebung Exadel Studio....20
    3.2.2 Einbinden der Tag-Bibliotheken und Strukturierung....21
    3.2.3 Datenspeicherung innerhalb einer Session....21
    3.2.4 Sprachbestimmung und Sprachwechsel....22
    3.2.5 Der Footer-Bereich....22
    3.2.6 Ein neues Benutzerkonto anlegen....23
    3.2.7 Die Passwort-vergessen-Funktion für Vergessliche....26
    3.2.8 Einloggen mit den korrekten Zugangsdaten....27
    3.2.9 Aufgaben hinzufügen und verwalten....27
  3.3 Realisierung mit PHP und Prototype....32
    3.3.1 Die Entwicklungsumgebung....32
    3.3.2 Aufteilung der Anwendung....32
    3.3.3 PHP-Sessions....33

3.3.4 Sprachwahl und -wechsel......33
3.3.5 „Footer wechsel dich”......34
3.3.6 Die Registrierung......35
3.3.7 Neues Passwort mailen per PHP......37
3.3.8 Einloggen in die ToDo-Liste......37
3.3.9 Aufgaben verwalten......38

4. Fazit......43
4.1 Vor- und Nachteile der Realisierung mit JavaServer Faces......43
4.2 Vor- und Nachteile der PHP/Prototype-Realisierung......44
4.3 Zusammenfassung......45

5. Quellenangabe......46
Literatur......46
Quellen im Internet......46

1.Motivation

„I think there is a world market for maybe five computers" - jeder von uns kennt dieses Zitat, das seinerzeit der IBM-Gründer Thomas J. Watson geprägt haben soll, und auch jeder von uns weiß, dass diese Einschätzung falsch war. Heute findet sich vielerorts in jedem Haushalt mind. ein PC und über das mittlerweile zum Alltag gehörende Internet sind weltweit hunderte Millionen Computer miteinander vernetzt.

Was ursprünglich als reines Projekt zur Vernetzung von Forschungsrechnern begann, hat sich über E-Mail und World wide Web bis zum heutigen mobilen Internet weiterentwickelt. Internet und Mobiltelefone lassen uns und alles jederzeit überall auf der Welt erreichbar sein. Was für Manche eher einem Fluch gleichkommt, ist beispielsweise für Unternehmen ein Segen. Der Trend geht dazu hin, immer mehr Unternehmensanwendungen über das Internet zugänglich zu machen. Auf der einen Seite kann man dies über eine Web-Service-Schnittstelle realisieren, auf der anderen Seite ist es ebenso möglich, die komplette Anwendung in den Browser auszulagern.

Die Auslagerung stellt in jedem Fall eine weitreichende Entscheidung fest, die hohe Entwicklungskosten mit sich bringt. Deshalb ist es essentiell sich im Vorherein über die möglichen Techniken zur Umsetzung gründlich zu informieren, die jeweiligen Stärken und Schwächen auszuloten und anhand der Anforderungen die beste Lösung zu suchen anstatt im Nachhinein die zu programmierende Anwendung abspecken zu müssen, da einzelne Features in der gewählten Programmier- und Skriptsprache nicht umsetzbar sind.

Im Bereich der Anwendungsentwicklung für das Internet gibt es u. a. zwei konkurrierende Techniken, die sich für die Umsetzung anbieten: JavaServer Faces und PHP. Was sich genau dahinter verbirgt und welche Möglichkeiten dem Entwickler jeweils eröffnet werden warum vor allem auch das Thema AJAX eine immer wichtigere Rolle spielt soll diese Studienarbeit theoretisch und praktisch beleuchten um abschließend die jeweiligen Vor- und Nachteile sowie Potentiale herauszuarbeiten.

2. Verwendete Techniken

2.1 Ajax

Mit Aufkommen des Themas „Web 2.0” geriet auch der Bergriff AJAX in den Fokus. AJAX steht für „**A**synchronous **J**avaScript **A**nd **X**ML“ und ist keineswegs eine neue technische Neuerfindung oder Innovation, sondern basiert tatsächlich auf der Kombination von Altbekanntem.

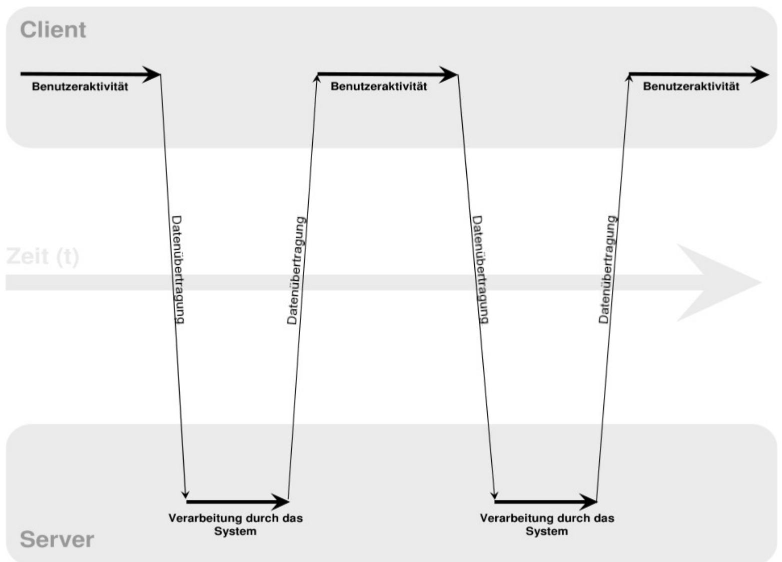


Abbildung 1: Bei synchroner Kommunikation sind immer nur Client oder Server aktiv (Quelle: Wikipedia)

Während das Internet an sich zustandslos ist und für jede inhaltliche Änderung auf einer Seite diese immer komplett neu geladen und übertragen werden muss, ermöglicht es AJAX lediglich einzelne Teile der bereits angezeigten Seite zu aktualisieren. Dazu wird, während die Seite dargestellt und der Benutzer ungestört weiter agieren kann, im Hintergrund per JavaScript eine Anfrage an den Webserver gesendet. Die Antwort des Servers wird dann wieder von der JavaScript-Funktion aufgegriffen und ausgewertet. Als Ergebnis ist dann in den meisten Fällen eine Änderung auf der Seite zu beobachten.

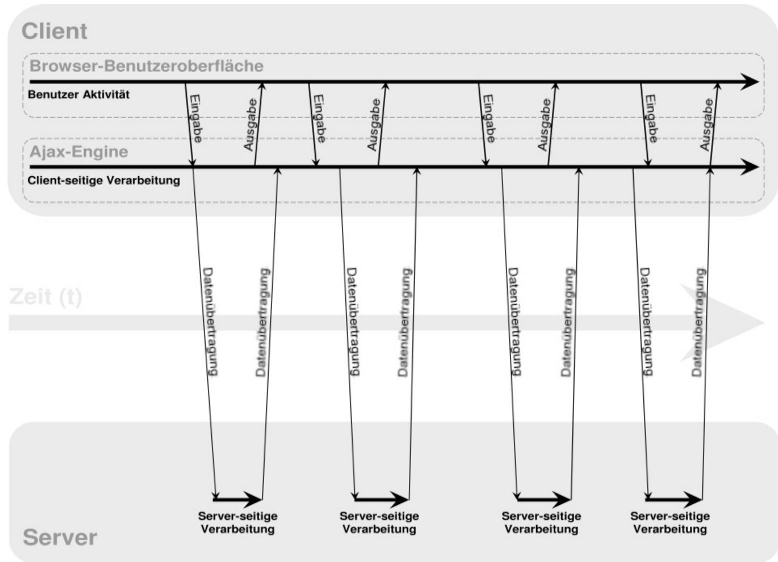


Abbildung 2: AJAX entkoppelt durch asynchrone Kommunikation Client und Server (Quelle: Wikipedia)

Die Vorteile dieser Technik liegen auf der Hand: die Benutzung wird schneller, da immer nur einzelne Bereiche ausgetauscht und somit weniger Daten übertragen werden. Für den Benutzer ist das zum Einen optisch ansprechender und ruhiger, zum Anderen auch komfortabler, da die Wartezeit abnimmt und die Ergebnisse sowie Auswirkungen seiner Aktionen schneller ersichtlich werden. Viel statischer Overhead vermieden wird dabei ebenfalls vermieden.

2.2 JavaServer Faces

Bei den JavaServer Faces (kurz: JSF) handelt es sich um ein noch junges Standard-Framework für Webanwendungen, die aktuelle Version 1.2 wurde am 11. Mai 2006 veröffentlicht. Über Tag-Bibliotheken werden JavaServer Pages um Oberflächenkomponenten erweitert, so lässt sich auf einer höheren Abstraktionsebene die Benutzerschnittstelle aus Komponenten zusammensetzen. JSF unterstützt das MVC-Konzept, also die strikte Trennung von **M**odell, Ausgabe (engl. **V**iew) und Steuerung (engl. **C**ontroller).

2.2.1 Dateien und Struktur einer JSF-Anwendung

Selbst für den einfachen Einstieg mit der obligatorischen „Hello World"-Anwendung benötigt man bei JSF schon eine umfangreiche Anzahl an Dateien, wobei 6 Arten zu unterscheiden sind.

Backing-Beans (siehe 2.2.6) sind Java-Klassen, zur Verwaltung von Informationen und führen die Logik der Anwendung aus. Auch bei den in Form von JAR-Archiven eingebundenen Bibliotheken handelt es sich um Java-Klassen, die die funktionale Basis der Anwendung darstellen.

Innerhalb von PROPERTIES-Dateien werden ebenfalls Informationen gespeichert, diese sind aber real-only, wie beispielsweise die Übersetzungen bei mehrsprachigen Anwendungen (siehe 2.2.5). Eine Vielzahl von Konfigurationsdateien definiert Beziehungen und Abhängigkeiten der einzelnen Dateien zueinander und macht Backing-Beans, Bibliotheken oder auch Sprachdateien innerhalb der Anwendung bekannt und nutzbar.

Die eigentlichen JSF-Dateien fügen schließlich die JSF-Komponenten zusammen. Das letztendliche Erscheinungsbild wird dann noch von sonstigen Dateien wie CSS-Definitionen oder Bildern bestimmt.

2.2.2 Die Umsetzung des MVC-Patterns

Jede JSF-Anfrage durchläuft insgesamt sechs Bearbeitungsphasen bis das Ergebnis im Browser zu sehen ist. Jede Phase ist dabei abhängig vom Erfolg der vorhergehenden Phase. Tritt ein Fehler auf, werden die nachfolgenden Phasen gar nicht mehr durchlaufen.

Erzeugung/Wiederherstellung des Komponentenbaums

Während eine Seite angezeigt wird, wird eine Vielzahl von Komponenten zwischengespeichert. Deren Zustände werden bei einer Anfrage in dieser Phase wiederhergestellt, wobei die Zuordnung über die in der Session gespeicherte View-ID erfolgt. Wenn die Seite das erste Mal aufgerufen wird, existiert die View-ID noch nicht und der Komponentenbaum neu erstellt.

Das Wiederherstellen des Komponentenbaums umfasst alle Komponenten mit ihren zugehörigen Validierern (engl. validator) und Konvertierern (engl. converter) sowie der Backing-Beans. Außerdem wird dabei auch noch die Lokalisierung (siehe 2.2.5) vorgenommen, damit die Anwendung später in der richtigen Sprache erscheint.

Übernahme der Anfragewerte

Bei jedem Request werden Daten übertragen, die der Benutzer beispielsweise über Eingabefelder eingegeben hat. Diese werden als POST-Parameter eines HTTP-Requests zunächst codiert und müssen von JSF in dieser Phase wieder decodiert werden. Die Werte werden dann vorläufig den zugehörigen Komponenten zugewiesen. Diese Zuweisungen werden aber wieder verworfen, falls in den nachfolgenden Phasen Fehler auftreten, z. B. eine Validierung fehlschlägt.

Validierung/Konvertierung

In diesem Fall werden die übertragenen Werte auf ihre Validität hin überprüft und ggf. kovertiert. Wie im nachfolgenden Kapitel noch genauer erläutert wird, kann jedem Eingabefeld ein Validierer zugewiesen werden, der bestimmte Gültigkeitsbedingungen mit diesem verbindet. Werden Letztere nicht erfüllt, so werden an dieser Stelle die Fehlermeldungen erzeugt und die weitere Abarbeitung abgebrochen.

Aktualisierung der Modellobjekte

Nun können die meist über das *value*-Attribut gebundenen Werte der Backing-Beans den jeweiligen UI-Komponenten zugewiesen werden.

Abbildung 3: Aus JavaServer Faces (Bernd Müller), Seite 49

Aufruf der Anwendungslogik

Nachdem alle Werte im gewünschten Format sind, kann die Anwendungslogik mit ihnen arbeiten. Über ihren Rückgabewert steuern die Methoden auch die Navigation (siehe 2.2.4) zwischen den verschiedenen Seiten der Anwendung.

Rendern der Antwort

Nach Abschluss aller Berechnungen ohne Fehler kann die Antwort gerendert werden. Dazu werden die JSF-Komponenten mit ihren zugewiesenen Werten in HTML-Code aufgelöst und als Abschluss der entstandene Komponentenbaum bis zur nächste Anfrage zwischengespeichert.

2.2.3 Eingabevalidierung

Eines der oft genannten Features von JSF ist die Validierung von Formularfeldern. Was in klassischem HTML noch nicht direkt möglich ist (die Web Forms 2.0[1] versprechen hier zukünftig Abhilfe), kann mit wenigen Zeilen Code dank JSF hinzufügt werden.

Der einfachste Fall der Eingabevalidierung ist, ein Feld zum Pflichtfeld zu machen. Hierfür muss dem Feld lediglich zusätzlich das Attribut *required="true"* mitgegeben werden:

```
<h:inputText value="#{bean.mandoryValue}" required="true" />
```

Außerdem kann man mit Hilfe von Validierern auch dafür sorgen, dass die Daten einen bestimmten Dateityp oder eine bestimmte Länge haben, z. B. zwischen 5 und 50 Zeichen:

1 http://www.whatwg.org/specs/web-forms/current-work/

```
<h:inputText id="minmax" value="#{bean.mandoryValue}">
    <f:validateLength minimum="5" maximum="50" />
</h:inputText>
<h:message for="minmax" />
```

Listing 2.1: Dem Eingabefeld werden ein Längen-Validierer sowie ein Feld zur Fehlerausgabe zugeordnet

Um Fehler während der Validierung anzuzeigen gibt es das Element *h:message*, dessen *for*-Attribut sich auf die ID eines Eingabefeldes bezieht, dem ein Validierer zugeordnet ist. Schlägt in diesem Eingabefeld die Validierung fehl, so wird der Fehler im Element *h:message* angezeigt. Während *h:message* den Fehler genau eines Textfeldes anzeigt, listet *h:messages* alle Fehlermeldungen der Seite, wenn es z. B. mehrere Felder mit Validierern gibt.

Die Fehlermeldungen sind dabei zunächst Standard-Texte, die sich bearbeiten und damit den eigenen Anforderung anpassen lassen.

Im Rahmen dieser Studienarbeit wird diese JSF-Funktionalität aber nicht weiter Verwendung finden, da die Validierung erst nach einem Request erfolgt (siehe 2.2.2). Durch den Einsatz von AJAX soll aber verhindert werden, dass ein Formular mit fehlerhaften Eingaben abgesendet wird, weshalb die Eingabevalidierung kontinuierlich schon während der Eingaben mit Hilfe von asynchronen Requests durchgeführt wird.

2.2.4 Navigation

Nach dem Auslösen eines Action-Events und der serverseitigen Verarbeitung wird standardmäßig die aufrufende JSF-Seite selber wieder angezeigt. Soll stattdessen auf eine andere Seite verlinkt werden, bedienen wir uns sog. Navigationsregel (engl. navigation rule), mit Hilfe derer anhand des Rückgabe-Strings der im *action*-Attribut angegebenen Funktion die anzuzeigende Seite bestimmt wird.

```
<navigation-rule>
 <from-view-id>/pages/login.jsp</from-view-id>
 <navigation-case>
         <from-outcome>success</from-outcome>
         <to-view-id>/pages/overview.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
         <from-outcome>failed</from-outcome>
         <to-view-id>/pages/login.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
         <from-outcome>passwordlost</from-outcome>
         <to-view-id>/pages/passwordlost.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

Listing 2.2: Navigationsregeln für die Datei ***/pages/login.js***

Die Navigationsregeln sind Teil einer Konfigurationsdatei, die über die *web.xml* eingebunden werden muss. Innerhalb einer Navigationsregel können neben einigen weiteren optionalen Elementen zudem auch Wildcards verwendet oder eine Beschreibung angegeben werden:

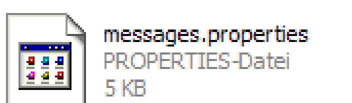
```
<navigation-rule>
    <description>
        Example using a wildcard within a navigation rule
    </description>
    <from-view-id>/pages/*</from-view-id>
    <navigation-case>
        <from-outcome>home</from-outcome>
        <to-view-id>/pages/index.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

Listing 2.3: Navigationsregeln mit optionalen Elementen und Wildcards

2.2.5 Internationalisierung

In Zeiten zunehmender Globalisierung gewinnt das Thema Internationalisierung eines Internetauftritts oder einer Webanwendung zunehmend an Bedeutung und kann auch oft ein grundlegender Pfeiler des angepeilten Erfolgs werden. JavaServer Faces bieten hier von Anfang an ein gutes Konzept, das nach der Trennung von Logik und Layout auch zusätzlich auch noch die Entkopplung von der Sprache bietet. Im Quelltext der JSF-Dateien finden sich keine hart kodierten Wörter und Sätze, sondern nur logische Bezeichner. Diese logischen Namen werden mit Hilfe von PROPERTIES-Dateien übersetzt.

Um die Anwendung mehrsprachig zu programmieren, benötigt man neben einer Standard-Sprach-Datei (in der zwingend alle Platzhalter übersetzt sein sollten) pro unterstützte Sprache eine Datei. Der Dateiname kann frei gewählt werden, da er später in der Konfigurationsdatei angegeben wird. Für die in Deutsch und Englisch zweisprachig angelegte Beispielapplikation werden also insgesamt drei Dateien benötigt. Der Name ist dabei frei wählbar, in diesem Fall also „messages".



Die Dateigröße lässt schon erahnen, dass die Datei *messages_de.properties* nicht viel Inhalt besitzt, in Wirklichkeit ist sie sogar komplett leer. Die deutschen Sprachinformationen sind in der Datei *messages.properties* enthalten, die als Fallback-Lösung fungiert, falls ein Bezeichner nicht aufgelöst werden kann. Deshalb wäre es unnötig und fehlerträchtig die deutschen Textpassagen redundant in zwei Dateien zu pflegen. Kann JSF auch auf diesem Weg den Platzhalter nicht auflösen, bleibt der entsprechende Platz in der Ausgabe leer.

Bei den Sprachdateien handelt es sich um einfache Textdateien, sodass die Sprachinformationen mit jedem Texteditor bearbeitet werden können, wie der kurze Ausschnitt zeigt:

```
label_realname=First name, Last name
label_email=Email address
label_username=Username
label_password1=Password
label_password2=Repeat password
```

Listing 2.4: In einer Sprachdatei werden alle Übersetzungen gespeichert

Kommentare sind dabei in der Sprachdatei nicht möglich, d.h. die Vielzahl an Übersetzungen könnte teilweise schnell unübersichtlich werden, weshalb gerade bei größeren Dateien auf eine Struktur geachtet werden sollte.

Um der Anwendung die Sprachdateien bekannt zu machen, benötigt es in einer beliebigen Konfigurationsdatei (sie muss lediglich in der *web.xml* definiert sein) folgende Zeilen:

```
<application>
    <locale-config>
        <default-locale>de</default-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>en</supported-locale>
    </locale-config>
    <message-bundle>messages</message-bundle>
</application>
```

Listing 2.5: Die Spracheinstellungen innerhalb der Konfiguration

Wie sich unschwer erkennen lässt, unterstützt unsere ToDo-Liste die deutsche („de") und englische („en") Sprache, wobei erstere Sprache der Standard ist. Als Wert des Elements *message-bundle* wird der Name der Sprachdateien angegeben: „messages".

Die Anwendung weiß nun von den Sprachdateien, in den einzelnen Dokumenten müssen diese trotzdem noch mit einer Zeile erwähnt und einem Bezeichner zugewiesen werden.

```
<f:view locale="en">
    <f:loadBundle basename="messages" var="msg" />
    ...
</f:view>
```

Erst jetzt kann auf die Sprachvariablen in gleicher Weise wie auf die von Beans verwalteten Variablen zugegriffen werden. Das Attribut *locale* weist im obigen Code die Anwendung an, die englische Sprachdatei zu verwenden. Ohne dieses Attribut würden die deutschen Übersetzungen eingebunden, da dies als Standard definiert wurde (Listing 2.5). Der Codeausschnitt

```
<h:outputLabel for="inputfield_email" value="#{msg.label_email}" />
```

erzeugt beispielsweise die HTML-Ausgabe

```
<label for="inputfield_email">Email address</label>
```

da wir in der PROPERTIES-Datei unter dem Bezeichner „label_email" wie oben gesehen die Zeichenkette „Email address" hinterlegt haben.

Problematisch werden kann das Thema Übersetzung vor allem dann, wenn innerhalb eines Fließtextes z. B. Links eingefügt werden sollen. Dann muss nämlich der Text in die Elemente der Typen *h:outputText* und *h:commandLink* aufgeteilt werden, was bei den sehr unterschiedlichen Satzaufbauten der einzelnen Sprachen kein leichtes Unterfangen darstellt.

2.2.6 Daten in Backing-Beans verwalten

Um auf Werte in Eingabefeldern zuzugreifen, nutzt JSF die schon von JSP her bekannten sog. Backing-Beans. Dabei handelt es sich um Java-Klassen, die Variablen verwalten und die Programmlogik in Methoden ausführen. Zur Einführung genügt eine minimale Klasse, in der lediglich die Variable *username* verwaltet wird und die ansonsten keine Funktionen enthält.

```
public class User {
    private String username;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
}
```

Listing 2.6: Die minimale Backing-Bean *User*

Zu jeder verwalteten Variable muss zwingend eine öffentliche GET- sowie SET-Methode existieren, die der Anwendung lesenden wie schreiben Zugriff ermöglicht. Selbstverständlich können innerhalb dieser Methoden neben dem Zurückliefern bzw. Setzen noch weitere Aktionen ausgeführt werden. Die Backing-Bean muss nun noch innerhalb der Anwendung durch einen neuen Eintrag in einer eingebundenen Konfigurationsdatei bekannt gemacht werden.

```
<managed-bean>
    <managed-bean-name>bean_user</managed-bean-name>
    <managed-bean-class>User</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

2.7: Durch diesen Eintrag in einer Konfigurationsdatei kann die Anwendung auf das Objekt zugreifen

Innerhalb der Anwendung kann nun mit dem unter *managed-bean-name* definierten Bezeichner auf die Variable *username* zugegriffen werden, z. B. über ein Eingabefeld:

```
<h:inputText value="#{bean_user.username}" />
```

Unter *managed-bean-scope* ist anstatt *request* auch noch der Wert *session* möglich. Das erzeugte Objekt der Klasse *User* wird dann innerhalb einer Session nur ein Mal durch den Aufruf des Konstruktors initialisiert und die in dieser Instanz gespeicherten Werte sind auch auf anderen Seiten als der aktuellen verfügbar. Allerdings sollten aus Performancegründen so wenig Variablen wie möglich in der Session gespeichert werden. Eine Lösung hierfür wäre beispielsweise eine eigene Klasse, die alle nötigen Session-Variablen enthält.

Ebenso kann ein in der Backing-Bean gespeicherter Wert ausgeben werden indem das Element *h:outputText* benutzt wird. Dabei kann auch Gebrauch von logischen und arithmetischen Ausdrücken gemacht werden. Alle folgenden Ausdrücke sind dabei u. a. möglich:

```
<h:outputText value="#{user.username}" />
<h:outputText value="#{user.geburtsjahr + 18}" />
<h:outputText value="#{user.geburtsjahr > 1985}" />
<h:outputText value="#{user.geburtsjahr%4==0 ? 'Schaltjahr' : 'keins'}" />
```

Neben Variablen kann die Backing-Bean zudem beliebige Funktionen enthalten, die auch von außen über einen *h:commandButton* oder *h:commandLink* aufgerufen werden können:

```
<h:commandButton action="#{user.doSomething}" />
```

Mit einem Klick auf die Schaltfläche wird die Methode *doSomething()* der Klasse *User* ausgeführt, die am Ende einen String zurück liefern sollte, der in den Navigationsregeln der aktuellen Seite einen Eintrag besitzt. Diese Regel würde dann ausgeführt, ansonsten wird die aktuelle Seite neu geladen. Die Übergabe von Parametern ist dabei nicht möglich, diese müssten notfalls aus Eingabe- oder versteckten Feldern gelesen werden.

Innerhalb der Methode kann nun alles Mögliche geschehen. Ein mögliches Szenario wie es auch bei der späteren Implementierung der Beispielanwendung vorkommen wird, ist die Speicherung der Daten in einer Datenbank. Abschließend können über die GET- und SET-Methoden können auch die Werte von Eingabefeldern zurückgesetzt werden, damit diese nach dem Neuladen der Seite leer sind.

2.2.7 Internationalisierung innerhalb einer Backing-Bean

In unseren Backing-Beans sollen später die aufgerufenen Methoden mit Meldungen über Erfolg oder Misserfolg berichten. Dass Problem an der Sache ist dann nur noch, dass diese Meldung auch in der richtigen Sprache sein muss, die jeweiligen Übersetzungen finden sich in de n PROPERTY-Dateien. Auch für diesen Fall gibt es in JSF eine Lösung. Der folgende Code-

auschnitt zeigt, wie innerhalb einer Backing-Bean auf die übersetzte Zeichenkette mit dem Bezeichner „message“ zugegriffen werden kann:

```
FacesContext context = FacesContext.getCurrentInstace();
ResourceBundle rb = ResourceBundle.getBundle(
    context.getApplication().getMessageBundle(),
    context.getViewRoot().getLocale() );

String message = rb.getString("message");
```

Listing 2.8: Auch innerhalb einer Backing-Bean kann auf die Übersetzungen zugegriffen werden

2.2.8 Daten einfach darstellen mit h:dataTable

Um Daten tabellarisch darzustellen benötigt es meist mehrere Arbeitsschritte: die benötigten Daten müssen gesucht, in einem Array o. Ä. untergebracht und anschließend durch das Durchlaufen von Schleifen dargestellt werden. Das Auffinden der gewünschten Daten kann auch JSF dem Entwickler nicht ersparen, die Darstellung erfolgt aber ganz einfach mit dem Element *dataTable* aus der JSF-HTML-Bibliothek.

Ein Codeausschnitt aus der späteren JSF-Implementierung zeigt anschaulich die Funktionsweise, wobei zur Erläuterung eine Spalte genügen soll:

```
<h:dataTable value="#{todolist.tasks}" var="task">
     <h:column>
          <f:facet name="header">
               <h:outputText value="Heading" />
          </f:facet>
          <h:outputText value="#{task.title}" />
     </h:column>
</h:dataTable>
```

Listing 2.9: Eine Tabelle mit den Titeln mehrere *Task*-Objekte wird dargestellt

Aus der Bean *todolist* wird ein Container mit Objekten der Klasse *Task* benötigt, von denen ausgewählte Werte (in diesem Fall der Titel) später in der Tabelle dargestellt werden sollen. Diese Objekte könnten z. B. als Liste zur Verfügung gestellt werden, die entsprechende GET-Methode hätte dann folgende Signatur:

```
public List<Task> getTasks() { ... }
```

Über das Attribut *var* bekommt die die Liste einen Bezeichner zugewiesen, der nachher in der Spalte wieder benötigt wird. Nacheinander werden nun die gewüschten Spalten definiert. Diese benötigen neben einer Überschrift, in Beispiel also „Heading“, ein Element *h:outputText* mit dem gewünschten Wert des Objekts. In obigem Fall wird also das Attribut *title* eines *Task*-Objekts angezeigt, der Zugriff erfolgt über den zuvor definierten Parameter *task,* wobei im Hintergrund wie immer die entsprechende GET-Methode aufgerufen wird.

Mehr Arbeit bereitet die Erstellung der Tabelle nicht. Zusätzliche Attribute erlauben z. B. auch den Spalten und Zeilen (alternierende) CSS-Klassen zuzuweisen (*rowClasses* bzw. *columnClasses*) oder die Tabelle an sich mit CSS zu gestalten (*style* bzw. *styleClass*).

Interessant ist die Kombination der Attribute *first* und *rows*. Ersteres Attribut gibt an, beim wievielten Objekt der Liste gestartet werden soll und wie viele (*rows*) Objekte dann in der Tabelle dargestellt werden sollen. In Kombination mit einer Zählervariablen sowie Buttons oder Links zur Navigation kann so eine Navigation innerhalb einer langen Liste realisiert werden. Die konkrete Umsetzung wird später bei der Implementierung beschrieben (siehe 3.2.8).

2.2.9 Erweiterbarkeit

Trotz der doch sehr langen Beschreibung der Features von JavaServer Faces gibt es doch noch einige fehlende Punkte. Es beginnt damit, dass es nicht für alle HTML-Elemente Äquivalente in JSF gibt. Gewöhnliche Absätze sowie Listen lassen sich in JSF beispielsweise nur mithilfe des Elements *f:verbatim* darstellen.

```
<f:verbatim><p></f:verbatim>
    <h:outputText value="#{msg.p_registerintro1}" />
<f:verbatim></p></f:verbatim>
```

Listing 2.10: Ein gewöhnlicher HTML-Absatz ist nur kompliziert darstellbar

Der Code wird dabei um ein Vielfaches länger und damit auch unübersichtlicher. Professioneller wäre es in einem solchen Fall jedoch einen eigenen Renderer zu schreiben. Weiterhin bietet JSF auch die Möglichkeit der Erstellung kompletter Komponenten – darunter ist im Wesentlichen die Zusammenfassung mehrere Elemente zu einem einzelnen Modul zu verstehen – sowie eigener Validierer, die wie bereits gesehen auf Eingabefelder angewendet werden können. Für die Beispielanwendung kann auf diese fortgeschrittenen Techniken aber noch verzichtet und lediglich mit den Standard-Elementen gearbeitet werden.

Außerdem bietet JSF auch nicht eine eigene Template-Engine um die Anwendung aufzuteilen und redundante Bereiche nur an einer Stelle pflegen zu müssen. Dieser Aufwand rentiert sich aber erst bei größeren Projekten und wird deshalb ebenso keine weitere Verwendung finden.

2.3 Das JSF-AJAX-Framework „Ajax4jsf"

Ajax4jsf erweitert JSF-Anwendungen um AJAX-Fähigkeiten, ohne dass der Entwickler auch nur eine Zeile JavaScript-Code schreiben muss. Stattdessen wird lediglich der Code der JSF-Anwendung um zusätzliche Ajax4jsf-Elemente erweitert, aus denen beim Compilieren dann automatisch die benötigten JavaScript-Funktionen erzeugt werden. Zuvor müssen noch einige Einstellungen vorgenommen werden, damit die neuen Elemente innerhalb der Anwendung bekannt sind. Zum Zeitpunkt der Studienarbeit war Ajax4jsf in der Version 1.0.6 vom 24. Januar 2007 aktuell.

Der erste Schritt besteht im Download[2] zweier JAR-Archive, die in der Dateistruktur der Anwendung im Ordner */WebContent/WEB-INF/lib* abgelegt werden müssen. Dann können sie von der Anwendung eingebunden werden, nachdem in der *web.xml* wie folgt ein neuer Filter hinzugefügt wurde. Zu beachten dabei: der Ajax4jsf-Filter muss VOR eventuell weiteren Filtern in dieser Datei stehen.

```
<filter>
    <display-name>Ajax4jsf Filter</display-name>
    <filter-name>ajax4jsf</filter-name>
    <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ajax4jsf</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

Listing 2.11: Das AJAX-Framework wird als neuer Filter hinzugefügt

2 https://ajax4jsf.dev.java.net/nonav/ajax/ajax-jsf/download.html

In der JSF-Datei muss das Framework nun noch am Dateibeginn registriert und einem Präfix zugewiesen werden. Dieses ist frei wählbar, es hat sich aber schon die Konvention gebildet hierfür „a4j“ zu verwenden. Die zum Einbinden notwendige Zeile sieht dann in JSP-Schreibweise folgendermaßen aus:

```
<%@ taglib uri="https://ajax4jsf.dev.java.net/ajax" prefix="a4j" %>
```

Nun weiß die Anwendung von der neuen Bibliothek und dem Entwickler stehen die neuen AJAX-Möglichkeiten zur Verfügung.

2.3.1 Eine erste AJAX-Verwendung

Beim ersten Beispiel soll das Referenzbeispiel genügen. Einerseits ist dieses sehr bildhaft, andererseits findet genau diese Funktion in der späteren Implementierung mehr als nur ein Mal Anwendung.

```
<f:view>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputText value="Type the Text:" />
            <h:inputText value="#{bean.text}">
                <a4j:support event="onkeyup" reRender="repeater" />
            </h:inputText>
            <h:outputText value="Text in the AJAX Response:" />
            <h:outputText id="repeater" value="#{bean.text}" />
        </h:panelGrid>
    </h:form>
</f:view>
```

Sendet AJAX-Request bei diesem Event

aj4:support erweitert das aktuelle Element um AJAX-Funktionalität

Zeigt auf die zu aktualisierenden Elemente

Listing 2.12: Erstes Beispiel der Arbeitsweise von Ajax4jsf

Der Quellcodeauschnitt zeigt ein Eingabefeld sowie drei Ausgabetexte. Davon besitzen zwei Elemente einen festen Wert, die dritte Textausgabe ist an die Variable *text* gebunden, die in der Backing-Bean *bean* verwaltet wird und zudem auch die Eingabe im Textfeld ist. Die Zeile

```
<a4j:support event="onkeyup" reRender="repeater" />
```

sorgt nun mit einem AJAX-Request im Hintergrund dafür, dass der Wert der Variablen *text* in *bean* bei jedem Tastendruck – *onkeyup* – aktualisiert wird. Zudem geben wird im Attributs *reRender* die ID des Ausgabetextes an, der diese Variable anzeigen soll und ebenfalls bei jedem Tastenanschlag aktualisiert wird.

Mehr Code ist schon gar nicht nötig, bei der Übersetzung des Programms werden automatisch die nötigen JavaScript-Funktionen generiert und bei der Eingabe von Text in das Textfeld wird dieser von nun an mit jedem Tastendruck auch außerhalb des Eingabefeldes zusätzlich angezeigt. Dem Attribut *reRender* können auch mit Kommata getrennt die IDs mehrerer Elemente übergeben werden.

2.3.2 h:commandButton vs. a4j:commandButton

Mit Hinzufügen von Ajax4jsf bekommt der CommandButton einen Zwilling, der sich aber ein wenig anders verhält. Von Code her sind die Unterschiede nur marginal:

```
<h:commandButton action="#{bean.method}" value="Submit h" />
<a4j:commandButton action="#{bean.method}" value="Submit a4j"
                   reRedender="element" />
```

Listing 2.13: Unterschiedliches Verhalten bei nahezu identischem Code

Beide rufen exakt dieselbe Methode auf – *method()* der Klasse *bean*. Während beim ersten Submit-Button allerdings die komplette Seite neu geladen wird, aktualisiert sich im zweiten Fall lediglich das JSF-Element mit der ID „element“. Sinnvollerweise sollte deshalb die Methode den Inhalt eben dieses Elements geändert haben.

Analoges gilt für die Elemente *h:commandLink* und sein AJAX-Pendant *a4j:commandLink*, auch hier kann die Aktualisierung auf bestimmte Bereiche beschränkt werden.

2.3.3 Fortgeschrittene AJAX-Funktionen

Über die angesprochenen Funktionen hinaus bietet Ajax4jsf auch noch einige andere Features, die aber im Rahmen dieser Studienarbeit keine weitere Anwendung finden und deshalb an dieser Stelle auch nicht weiter vorgestellt werden sollen. Einen guten Überblick über alle Funktionen bieten die Beispiele auf der Download-Seite unter:

https://ajax4jsf.dev.java.net/nonav/ajax/ajax-jsf/download.html.

2.4 PHP

Ursprünglich unter dem Namen „Personal Home Page Tools“ entwickelt, ist PHP heute ein rekursives Backronym für „PHP: Hypertext Preprocessor” und eine der meist verwendeten serverseitigen Skriptsprachen im Internetbereich.

PHP ist einfach zu erlernen, ein Grund dafür ist sicherlich die fehlende Typisierung. Trotzdem ist PHP im Laufe der Zeit immer mehr professionellen Anforderungen gerecht geworden und führte mit Version 5.0 umfangreiche objektorientierte Funktionen ein. Dabei ist das Einsatzgebiet von PHP nicht auf kleine und mittlere Projekte beschränkt, zahlreiche erfolgreiche Start-ups der Web2.0-Bewegung wie sevenload.de, foldk.com oder YouTube beweisen, dass die Skriptsprache auch für große und teils sicherheitskritische Anwendungen geeignet ist.

2.4.1 XML-Datei als Antwort senden

Im Rahmen dieser Arbeit soll der Einsatz von PHP auf das Thema AJAX beschränkt bleiben. Serverseitig übernimmt PHP dabei den Part der Anwendungslogik nachdem eine Datei von außen aufgerufen wurde und liefert nach Abschluss der Berechnungen das Ergebnis zurück. Auch wenn AJAX das „XML“ bereits im Namen trägt, können von JavaScript theoretisch auch unformatierte Textnachrichten verarbeitet werden. Die Zeile

```
echo 'Ich wurde per AJAX übertragen.';
```

liefert ein Dokument zurück, das eben nur diesen einen Satz enthält und das eine JavaScript-Funktion in dieser Weise auch ebenso verarbeiten könnte wie sein XML-Pendant, das ein paar Zeilen mehr Code benötigt, ohne dabei mehr Informationen zu übertragen.

```
header('Content-type: text/xml');
echo '<?xml version="1.0" ?>
    <xml>
        <msg>Ich wurde per AJAX übertragen.</msg>
        <html_msg><![CDATA['.$htmlcode.']]></html_msg>
    </xml>';
```

Listing 2.14: PHP liefert eine XML-Datei zurück

Zu beachten hierbei ist, dass die notwendige Angabe des Headers VOR der ersten Ausgabe von Text erfolgen muss, ansonsten kann die Header-Angabe nicht mehr geändert werden und PHP

liefert deshalb stattdessen vor dem restlichen Inhalt eine Fehlermeldung zurück. Auch mehrere Header-Angaben sind mit PHP möglich, diese werden jeweils als neue Zeile eingefügt.

Um HTML-Code, der üblicherweise einige öffnenden und schließenden spitzen Klammern enthält, zu übertragen, muss dieser von „<![CDATA[“ und „]]>“ eingeschlossen werden um nicht beim späteren Parsen der XML-Datei interpretiert zu werden.

2.4.2 Internationalisierung

Auch die PHP-Implementierung soll später zweisprachig ausgelegt werden und wie auch schon in JSF werden für die Übersetzung der Textpassagen einfache Textdateien genutzt. Pro unterstützter Sprache gibt es wieder eine Datei, die in diesem Fall ein Array mit allen übersetzten Texten enthält. Die Anwendung lädt später je nach gewünschter Sprache die passende Sprachdatei und ermittelt über den Index die jeweilige Übersetzung.

In PHP sehen dann die Einträge in der Sprachdatei wie folgt aus:

```
$msg['label_realname']  = 'First name, Last name';
$msg['label_email']     = 'Email address';
$msg['label_username']  = 'Username';
```

Listing 2.15: Übersetzungen von Textpassagen in PHP

In der Ausgabe wird dann beispielsweise einem HTML-Label der Text zugewiesen:

```
<label for="email"><?= $msg['label_email'] ?></label>
```

2.5 Die Ajax-Bibliothek „Prototype“

„Prototype is a JavaScript Framework that aims to ease development of dynamic web applications.“ – dieses Versprechen geben die Entwickler des unter *http://www.prototypejs.org* kostenlos verfügbaren clientseitigen Java-Script-AJAX-Frameworks, mit dem sich auch leicht visuelle Effekte erzeugen lassen. Und tatsächlich hat sich Prototype aufgrund des hohen Funktionsumfangs und der leichten Benutzung zu einem der populärsten AJAX-Frameworks gemausert und ist mittlerweile auch die Basis einer Vielzahl anderer Frameworks.

Im Januar 2007 ist die Version 1.5 erschienen, die erstmals auch eine detaillierte API-Dokumentation mit sich bringt. Gerade die fehlende Dokumentation war einer der größten Kritikpunkte der Vorgängerversionen. Da es sich bei Prototype um ein clientseitiges Framework handelt, kommt man hier aber nicht ohne zumindest grundlegende JavaScript-Kenntnisse aus.

Die gesamte Funktionalität von Prototype ist in einer knapp 70 KB großen Datei enthalten, die sich mit einer einzelnen Zeile einbinden lässt:

```
<script type="text/javascript" src="prototype.js"></script>
```

Von nun an kann in allen JavaScript-Funktionen auf die Prototype-Objekte und -Methoden zugegriffen werden.

2.5.1 Wichtige Prototype-Funktionen

Um Prototype für asynchrone Kommunikation benutzen zu können, sollten zumindest die zwei wichtigsten Prototype-Funktionen bekannt sein, die den Zugriff auf HTML-Elemente sowie deren Werte und Eigenschaften über deren ID ungemein erleichtern.

$F() liefert direkt den Wert des per ID angesprochen Elements zurück, insbesondere interessant um auf die Werte von Eingabefelder zuzugreifen.

$() bietet direkt Zugriff auf das DOM-Element mit der übergebenen ID. Mit einer solchen Referenz auf ein Element, können der Inhalt ausgetauscht oder die CSS-Eigenschaften und -Klassen manipuliert werden um beispielsweise Blöcke ein- und auszublenden. Neben dem Zugriff auf einzelne CSS-Eigenschaften gibt es oft auch Funktionen, die den gleichen Effekt erzielen. So blenden zum Beispiel die beiden folgenden Zeilen den Block *sampleDIV* aus:

```
$('sampleDIV').style.display = 'none';
$('sampleDIV').hide();
```

2.5.2 Asynchrone Kommunikation mit Prototype

Wichtig für diese Studienarbeit sind die AJAX-Funktionalitäten. Hierfür implementiert Prototype die Klasse *Ajax.Request*, die normalerweise asynchron (synchrone Requests sind optional auch möglich) einen Request auf einen angegebenen URL ausführt.

Im Hintergrund wartet die aufrufende JavaScript-Funktion dann auf die Antwort, die weiterverarbeitet werden kann. Der Benutzer kann in der Zwischenzeit uneingeschränkt weiter agieren ohne von den Vorgängen im Hintergrund überhaupt etwas mitzubekommen.

Die Nutzung eines asynchronen Requests erfolgt nach folgendem Muster:

```
var my1stAjax = new Ajax.Request(
    url,
    {
        method: "post",
        parameters: params,
        onComplete: complete_action
    } );
```

Listing 2.16: Ein erster asynchroner Request mit einem Objekt der Klasse *Ajax.Request*

Es wird also *url* mit den Parametern *params* als POST-Request aufgerufen, bei Antwort wird die Funktion *complete_action* ausgeführt. Wahlweise können an dieser Stelle anstatt eines Funktionsaufrufs auch direkt JavaScript-Befehle angegeben werden, die die Antwort des Servers behandeln. Bei der Antwort muss es sich nicht notwendigerweise um ein XML-Dokument handeln, es können auch reine Textdateien zurückgegeben und verarbeitet werden.

Um die Antwort direkt zu bearbeiten muss der Quellcode hinter *onComplete* ein wenig modifiziert werden:

```
...
onComplete:function(r) { // Antwort bearbeiten }
...
```

Listing 2.17: Direkte Verarbeitung der Antwort statt Funktionsaufruf

Obiges Beispiel ist noch etwas unflexibel wenn es darum geht, zwischen erfolgreichen und weniger erfolgreichen Requests (z. B. kann der URL nicht existieren) zu unterscheiden. Um bei dabei unterschiedlich zu reagieren, kann anstatt *onComplete* auch die Kombination aus *onSuccess* und *onFailure* benutzt werden.

```
var my2ndAjax = new Ajax.Request(
    url,
    {
        method: "post",
        parameters: params,
        onSuccess: success_action,
        onFailure: failure_action
    } );
```

Listing 2.18: Fallunterunterscheidung bei der Verarbeitung der Antwort

Damit kann man auch auf Seiten des PHP-Skripts zwischen der behandelnden JavaScript-Version wählen, indem man explizit über die Funktion *header()* den Letzteren setzt. Ein

```
header('HTTP/1.0 400 Bad Request');
```

in der PHP-Datei würde JavaScript einen Fehler signalisieren und somit explizit die *onFailure*-Routine ansprechen.

Weitere optionale Parameter bieten zusätzliche Steuerungsmöglichkeiten für den Request. So können auch abhängig vom aktuellen Status – ein AJAX-Request durchläuft die Status *uninitized*, *loading*, *loaded*, *interactive* und *complete* – Aktionen ausgeführt werden.

Oft soll beim Thema AJAX ein Teil der dargestellten Seite ausgetauscht werden ohne die Seite komplett neu zu laden. Um dann nicht immer explizit den Inhalt im DOM-Baum ersetzen zu müssen, kann man dies auch per *Ajax.Updater* bzw. periodisch per *Ajax.PeriodicalUpdater* erledigen. Beide Klassen bekommen neben den von *Ajax.Request* her bekannten Optionen zusätzlich noch die ID des zu aktualisierenden Elements übergeben.

2.5.3 AJAX-Antwort verarbeiten

Egal ob die Funktion außerhalb oder innerhalb des AJAX-Requests definiert wurde, muss das zurückgelieferte (in unserem Fall XML-)Dokument in irgendeiner Weise verarbeitet werden.

```
function handleResponse(r) {
    var msg = r.responseXML.getElementsByTagName('msg')
[0].childNodes[0].nodeValue;
    $('message').innerHTML = msg;
}
```

Listing 2.19: Die Antwort-XML wird ausgelesen und als Inhalt des Elements *message* im DOM-Baum gesetzt

Mit den JavaScript-Funktionen für XML-Dokumente lassen sich auch komplexe XML-Dateien durchlaufen. Im Listing 2.19 ermittelt JavaScript den Inhalt der Variablen *msg* aus dem XML-Dokument, das durch die Variable *r* repräsentiert wird. Per *innerHTML* wird der Inhalt des Elements *message* durch *msg* ersetzt. Während *innerHTML* bei fast alle HTML-Elementen anwendbar ist, muss bei Eingabefeldern stattdessen die Eigenschaft *value* benutzt werden.

2.5.4 Asynchroner Submit

Ein asynchroner Request, wie er in 2.5.2 vorgestellt wurde, kann insbesondere dazu genutzt werden, Eingaben schon vor dem Abschicken des Formulars auf ihre Gültigkeit hin zu prüfen und bei Fehlern das Abschicken zu verhindern, indem die Absenden-Schaltfläche entsprechend inaktiv geschaltet wird.

Um einen Schritt weiter zu gehen kann auch beim Abschicken eines korrekt ausgefüllten Formulars das komplette Neuladen der Seite noch unterbunden werden, indem dieser Fall abfangen und ebenso asynchron im Hintergrund eine (PHP-)Datei zur Verarbeitung aufrufen wird.

```
$('form_submitemail').onsubmit = function() {
    var submit_form = new Ajax.Request(
            ...
        );
    return false;
}
```

Listing 2.19: Das Absenden des Formulars *form_submitemail* wird im Hintergrund behandelt

Wichtig im Listing 2.19 ist die letzte Zeile „return false“. Es damit wird das eigentliche Abschicken verhindert.

3. Realisierung einer Beispielapplikation

3.1 Die Beispielapplikation – MuchToDo

Anhand einer Beispielanwendung soll ein Vergleich zwischen JavaServer Faces mit AJAX und PHP mit Ajax gezogen werden. Bei „MuchToDo“ handelt es sich – wie der Name schon vermuten lässt – um eine ToDo-Liste, in der Aufgaben verwaltet werden können. Aufgrund der zeitlichen Beschränkung der Studienarbeit ist die Anwendung bewusst etwas kompakter ausgefallen und besteht lediglich aus vier Seiten mit den folgende Funktionen:

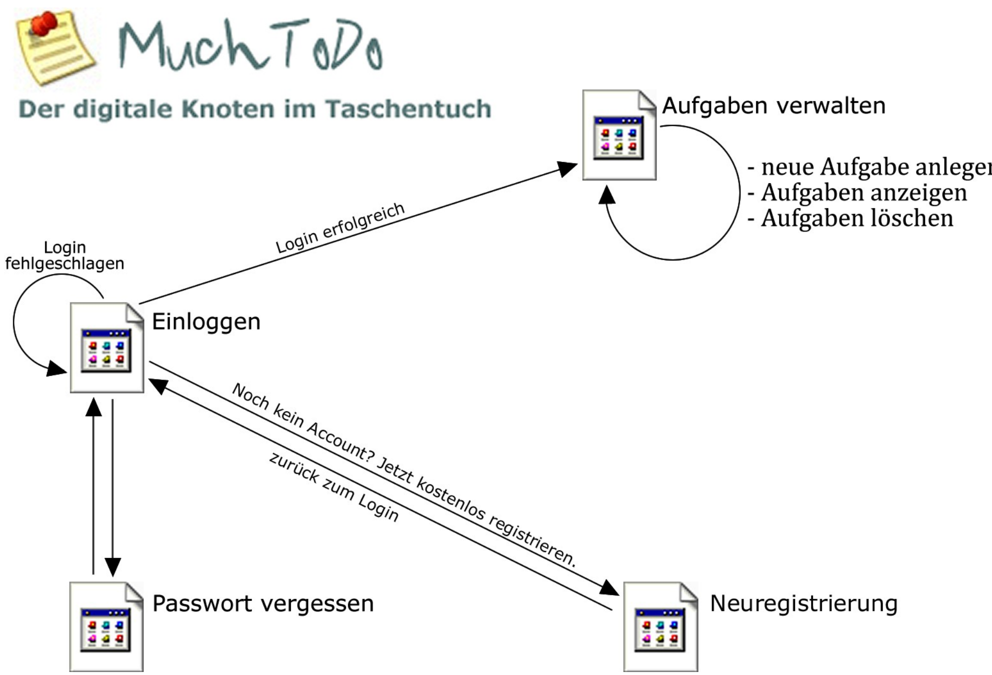


Abbildung 4: Struktur der Beispielanwendung "MuchToDo"

3.1.1. Neuregistrierung

Um die Anwendung nutzen zu können muss sich der Benutzer mit Benutzername und Passwort authentifizieren, wozu einmalig eine Registrierung erforderlich ist. Das einfache Formular fragt dazu neben dem Namen und der E-Mail-Adresse des Benutzers den gewünschten Benutzernamen sowie das Passwort ab. Die Gültigkeit der Eingaben des Benutzers soll dabei sofort per Ajax überprüft werden, erst wenn alle Eingaben korrekt und zulässig sind, kann die Registrierung abgeschlossen werden.

Für die einzelnen Felder gelten dabei die folgenden Bedingungen:

Vorname, Name des Benutzer. Der Name des Benutzers muss mindestens ein Länge von fünf Zeichen haben, Leerzeichen am Anfang oder Ende werden nicht mitgezählt.

E-Mail-Adresse. Die E-Mail-Adresse muss syntaktisch korrekt und noch nicht registriert sein, da pro E-Mail-Adresse nur ein Account erstellt werden darf.

Benutzername. Der Benutzer kann den gewünschten Benutzernamen frei wählen, solange er nicht schon verwendet wird. Auch der Benutzername muss mindestens fünf Zeichen lang sein.

Passwort und Passwortwiederholung. Um eine gewisse Passwortsicherheit zu gewährleisten, sind auch beim Passwort fünf Zeichen das Minimum. Zur Erfüllung dieses Kriteriums müssen zudem Passwort und -wiederholung übereinstimmen.

Solange nicht alle Bedingungen erfüllt sind, ist die Schaltfläche „Registrieren" inaktiv, der Benutzer erhält zusätzlich hinter den fehlerhaften Eingabefeldern entsprechende Hinweise.

3.1.2 Passwort-vergessen-Funktion

Jeder von uns hat schon einmal etwas vergessen. Damit der Benutzer in diesem Fall noch die Möglichkeit hat auf seinen Account zuzugreifen, bietet die To-Do-Liste eine Passwort-vergessen-Funktion.

Nach Eingabe einer registrierten E-Mail-Adresse wird ein zufälliges Passwort generiert, zum Benutzerkonto der eingegeben E-Mail-Adresse gespeichert und dem Benutzer per E-Mail zugeschickt.

3.1.3 Der Login

Ein Benutzer kann sich mit seinem Benutzernamen und Passwort authentifizieren und so Zugriff auf seine offenen Aufgaben erhalten. Auch hier soll der Anmeldeversuch nur möglich sein, wenn ein registrierter Benutzername sowie ein Passwort angegeben wurde, andernfalls ist die Schaltfläche inaktiv.

Ist die Anmeldung erfolgreich, erfolgt die Weiterleitung zur eigentlichen Verwaltungsoberfläche, ansonsten wird auf der Login-Seite eine Fehlermeldung ausgegeben.

3.1.4 Die Verwaltungsoberfläche

Der Kern der Beispielapplikation: Es sollen Aufgaben verwaltet werden. Zur Verwaltung gehört in Beispiel das Hinzufügen neuer und das Löschen vorhandener Aufgaben, die in der Zwischenzeit erledigt wurden. Entsprechend teilt sich auch die Benutzeroberfläche auf, im oberen Teil können neue Aufgaben mit Titel, Beschreibung und Stichwötern (sog. Tags) hinzugefügt werden, im unteren Teil ist eine filter- und blätterbare Liste aller vorhandenen Aufgabe zu sehen.

Bei einer neuen Aufgabe sind Titel und mindestens ein Tag Pflicht, früher kann das Formular nicht abgeschickt werden. Nachdem das Formular abgeschickt und die neue Aufgabe gespeichert wurde, wird dies dem Benutzer durch eine entsprechende Erfolgsmeldung mitgeteilt, gleichzeitig wird auch die Liste anstehender Aufgaben auf den aktuellen Stand gebracht.

Die Liste selber lässt sich auch nach Suchbegriffen filtern, wobei Titel, Beschreibung und Tags durchsucht werden. Der gewünschte Suchbegriff wird dazu in einem Suchfeld eingegeben und die Liste nach jedem Tastenanschlag aktualisiert.

Schließlich soll es noch möglich sein, eine Aufgabe als erledigt zu kennzeichnen und damit zu löschen. Auch nach jedem Löschen wird die Liste aktualisiert.

3.1.5 Gleiche Voraussetzungen: die CSS-Dateien

Leider ließ sich der Plan komplett identische CSS-Dateien für beide Implementierungen zu nutzen nicht vollends umsetzen. Die Style-Definitionen in den verwendeten Dateien stimmen bei beiden Implementierungen zwar größtenteils überein, aber eben nur größtenteils. Vorwegnehmen will ich deshalb die Problematik in JSF mit den IDs der gerenderten HTML-Ele-

mente. Die bei JSF-Elementen eingegebene ID wird beim Rendern mit einem Präfix der Art „_idX“ versehen, wobei *X* eine Zahl darstellt.

Da das Präfix nicht vorhersagbar ist und sich auch beim Einfügen weiterer Elemente immer wieder ändern kann, ist die Nutzung der ID für CSS-Angaben in der JSF-Implementierung fast unmöglich. Man muss sich stattdessen auch bei eindeutigen Elementen mit CSS-Klassen behelfen, was nicht dem Grundgedanken von CSS entspricht, einmaligen Elementen durch die ID eine höhere Spezifität zuzuweisen als durch das *class*-Attribut, das für mehrfach vorkommende Elemente gedacht ist.

Somit ergeben sich bei einzelnen Elementen Unterschiede innerhalb der verwendeten CSS-Dateien, die aber nahezu identische Auswirkungen auf die Darstellung haben.

3.1.6 Die Datenbankstruktur

Zur Umsetzung der in den Kapiteln 3.1.1 bis 3.1.4 definierten Anforderungen bedarf es nur einer einfachen Datenbankstruktur. Beide Implementierungen in JSF und PHP greifen dabei auf dieselbe Datenbasis zu.

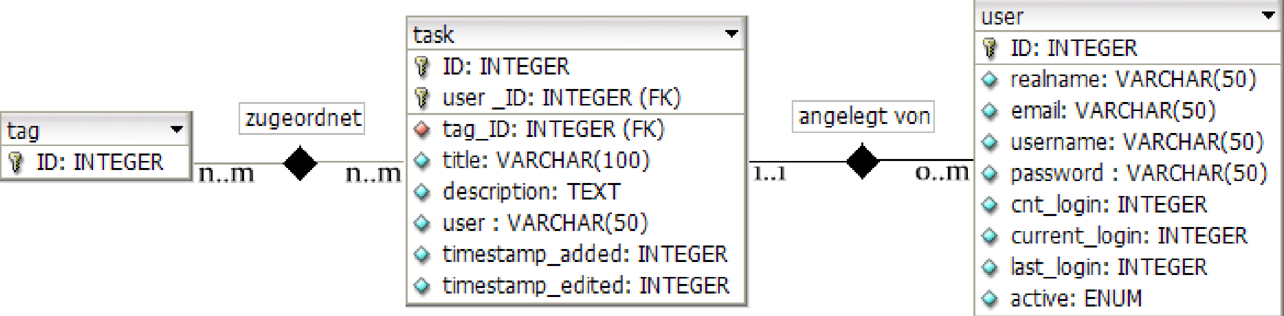


Abbildung 5: Die Datenbankstrutur der Beispielanwendung

3.2 Realisierung mit JSF und Ajax4jsf

3.2.1 Die Entwicklungsumgebung Exadel Studio

Aufgrund der vielen für eine JSF-Anwendung nötigen Dateien und der gegenseitigen Abhängigkeiten bietet es sich an, auf eine Entwicklungsumgebung zurückzugreifen. Da die Beispielapplikation noch weit von einer komplexen Web-Anwendung entfernt ist, kann die Wahl auf ein kostenloses Produkt aus dem OpenSource-Lager fallen: Exadel Studio – ein Plugin für eine spezielle Ansicht in Eclipse, das auch gleich den Servlet-Container Tomcat mitbringt.

Exadel Studio zeigt wahlweise den Quelltext, eine grafische Ansicht oder auch beides zusammen. Etwas ärgerlich fällt schon auf den ersten Blick auf, dass die Vorschau CSS nicht berücksichtigt und deshalb das Projekt zunächst immer mit dem Tomcat-Server synchronisiert werden muss um die Änderungen im Browser zu kontrollieren.

Ansonsten bietet die Entwicklungsumgebung eine große Hilfestellung. Automatische Tag-Vervollständigung ist ebenso vorhanden wie Syntax-Hightlighting. Außerdem hilft die Tastenkombination <Strg>-<Leertaste> auch weiter um die möglichen Attribute zu einem JSF- oder Ajax4jsf-Element nachzuschlagen. Von dieser Möglichkeit sollte man Gebrauch machen, denn unzulässige Attribut für bestimmte Elemente werden nicht von Exadel Studio, sondern erst vom Tomcat erkannt und mit einer Fehlermeldung statt der gewünschten Seite quittiert. Da die nach Änderungen jeweils nötige Synchronisation (optional auch automatisch nach jeder

Änderung) mit dem Servlet-Container immer auch eine gewisse Zeit in Anspruch nimmt, verlängern solche vermeidbaren Fehler merklich die Arbeitszeit.

Leider kommt bezüglich der Synchronisation noch hinzu, dass die Verzahnung zwischen Exadel Studio und Tomcat nicht immer einwandfrei funktioniert. Unzählige Male im Laufe der Studienarbeit verweigerte sich die Synchronisation mit der Begründung fehlender oder vermeintlich schreibgeschützter Ordner – ohne erkennbaren Grund. In diesem Fall hilft nur mehrfaches Neustarten des Tomcat, Entfernen und Neu-Hinzufügen der Projekte, oder schlimmstenfalls auch nur das manuelle Kopieren von Dateien vom Eclipse Projektordner in das virtuelle Tomcat-Verzeichnis.

3.2.2 Einbinden der Tag-Bibliotheken und Strukturierung

Wie schon angesprochen erweitern Tag-Bibliotheken gewöhnliche JSP-Seiten um JSF-Komponenten. Diese Bibliotheken müssen in jeder Datei zu Beginn eingebunden werden, was in JSP-Schreibweise wie folgendermaßen aussieht:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="https://ajax4jsf.dev.java.net/ajax" prefix="a4j" %>
```

Listing 3.1: Die Tagbibliotheken werden in die aktuelle Seite eingebunden

Neben der HTML- und der CORE-Bibliothek binden wir auch gleich die Ajax4jsf-Bibliothek ein. Jede Bibliothek bekommt dabei ein frei wählbares Präfix zugewiesen, über das die einzelnen Elemente angesprochen werden, die gewählten Bezeichnung haben sich als Quasi-Standard etabliert.

Durch die Nutzung von des *jsp:include*-Elements sollte es eigentlich möglich sein die immer identischen Bereiche Header und Footer in separate Dateien auslagern – eigentlich. Leider steht in diesem Fall aus unerfindlichen Gründen Ajax4jsf im Weg. Auch in jeder ausgelagerten Datei müssen die Tag-Bibliotheken, wie in Listing 3.1 gezeigt, eingebunden werden, was bei den ersten beiden JSF-Bibliotheken auch problemlos funktioniert. Sobald allerdings versucht wird die Ajax4jsf-Bibliothek einzubinden, verweigert diese ihren Dienst auf der gesamten Seite. Eine Fehlermeldung gibt es von Java nicht, nur der Browser moniert, auf ein bestimmtes JavaScript-Objekt nicht zugreifen zu können. Kein Wunder, denn es wurde überhaupt keine JavaScript-Datei erzeugt.

Eine Erklärung dieses Phänomens gibt es nicht, jedoch bleibt nichts Anderes übrig als den Code für den Footer auf jeder Seite redundant einzufügen. Wenigstens lässt sich der Header auslagern, da dieser keine Ajax4jsf-Elemente enthält.

```
<!-- include header -->
<jsp:include page="/blocks/header.jsp" />
```

3.2.3 Datenspeicherung innerhalb einer Session

Bei der Besprechung von Backing-Beans wurde schon erwähnt, dass diesen unter *managed-bean-scope* die Werte *reqeuest* oder *session* zugewiesen werden können. Der Unterschied liegt dann darin, dass bei *request* ein Objekt der jeweiligen Klasse bei jedem neuen Seitenaufruf neu erstellt, d. h. der Konstruktor aufgerufen wird. Bei *session* ist dies nur einmalig beim ersten Aufruf der Anwendung der Fall.

Damit können wir innerhalb dieses Objekts Daten speichern, auf die über mehrere Seiten hinweg zugegriffen werden soll. Die Daten sind also quasi in der Session gespeichert. Durch die

Verwendung statischer Attribute und Methoden lassen sich die Daten von überall setzen und auslesen. In der Beispielanwendung gibt hierfür es die Klasse *MuchToDoSession*.

```
<managed-bean-class>muchtodo.MuchToDoSession</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
```

Listing 3.2: Die „Session-Klasse" wird in einer Konfigurationsdatei definiert

3.2.4 Sprachbestimmung und Sprachwechsel

Wie in 2.2.5 gesehen kann über das Attribut locale in f:view der Anwendung die zu verwendende Sprache mitgeteilt werden. Was noch nicht erwähnt wurde ist, dass JSF die Sprachen für jede Seite einzeln verwaltet, weshalb die Sprache innerhalb der Session gespeichert werden muss. Zum Setzen der Sprache gibt es die zwei Methoden *toGerman()* sowie *toEnglish()*, in denen der Sprachwechsel persistiert wird.

```
public String toGerman() {
    FacesContext context = FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale(Locale.GERMAN);
    rb = ResourceBundle.getBundle(
             context.getApplication().getMessageBundle(), Locale.GERMAN );
    MuchToDoSession.setLg("de");
    return null;
}
```

Listing 3.3: Sprachwechsel zur deutschen Sprache (analoge Funktion für Englisch)

Über den *FacesContext* wird die neue Sprache für die aktuelle Seite gespeichert und zudem noch mittels der statischen Methode *setLg()* der Klasse *MuchToDoSession* sozusagen in der Session gespeichert. Der Aufruf erfolgt synchron über zwei kleine Flaggen im Browserfenster.

```
<h:form>
    <h:commandLink action="#{login.toEnglish}" immediate="true">
        <h:graphicImage value="/images/flag-usa.gif" alt="englisch"
                        title="englisch" />
    </h:commandLink>
    <h:commandLink action="#{login.toGerman}" immediate="true">
        <h:graphicImage value="/images/flag-germany.gif" alt="deutsch"
                        title="deutsch" />
    </h:commandLink>
</h:form>
```

Listing 3.4: Über zwei Grafiken als Links wird die Sprache der Anwendung geändert

Innerhalb der Backing-Beans wird nun auch noch die korrekte Übersetzung der Meldungen über die in der Session-Klasse gespeicherte Sprache ermittelt.

```
FacesContext context = FacesContext.getCurrentInstance();
if( MuchToDoSession.getLg().equals("de") ) {
    rb = ResourceBundle.getBundle(
            context.getApplication().getMessageBundle(), Locale.GERMAN );
} else {
    rb = ResourceBundle.getBundle(
            context.getApplication().getMessageBundle(), Locale.ENGLISH );
}
```

Listing 3.5: Die Klasse übernimmt die Sprache für die internen Übersetzungen

Leider funktioniert dies direkt nach dem Sprachwechsel noch nicht korrekt, die Meldungen erscheinen bei der ersten Anzeige noch in der falschen Sprache, erst nachdem sie beispielsweise über *reRender* eines *a4j:support*-Elements aktualisiert wurden, sind diese korrekt übersetzt.

3.2.5 Der Footer-Bereich

In Abschnitt 3.2.2 ist schon angeklungen, warum sich der Footer nicht in eine separate Datei auslagern lässt. Deshalb müssen auf jeder Seite dieselben knapp 40 Zeilen Code hinein kopiert

werden. Zwei *a4j:commandLink*-Elemente sorgen asynchron dafür, dass zwei *a4j:outputPanel* durch in der Backing-Bean verwaltete CSS-Klassen ein- und ausgeblendet werden.

```
...
<a4j:commandLink action="#{footer.showCopyright}" value="Copyright"
                 reRender="inc_contact, inc_copyright" />
<a4j:commandLink action="#{footer.showContact}" value="Kontakt"
                 reRender="inc_contact, inc_copyright" />
...
<a4j:outputPanel layout="block" id="inc_copyrighth">...</a4j:outputPanel>
<a4j:outputPanel layout="block" id="inc_contact">...</a4j:outputPanel>
...
```

Listing 3.6: Die entscheidenden Elemente des Footers auf jeder Seite

Das Attribut *layout* mit dem Wert „block“ sorgt dafür, dass die zwei Elemente vom Typ *a4j:outputPanel* als DIV gerendert werden, wobei das DIV mit der ID „inc_copyright“ Copyright-Informationen zur Studienarbeit und das zweite ein kleines Kontaktformular enthält.

Das Formular kann nur abgeschickt werden wenn alle Felder korrekt ausgefüllt wurden. Per *a4j:support* wird auf gewohnte Weise die Validierung durchgeführt und etwaige Fehler über die CSS-Klasse des Labels dargestellt, indem der Text fett darstellt wird. Die E-Mail-Adresse wird dabei zusätzlich auf Syntax geprüft, bei allen anderen Feldern ist lediglich die Zeichenanzahl ausschlaggebend.

Als Button wird ein *a4j:commandButton* verwendet, der leider nicht aktiv und inaktiv geschaltet werden kann, dafür aber das Formular asynchron abgeschickt werden kann. Fehler werden hinter den entsprechenden Feldern angezeigt. Im korrekten Fall erscheint eine positivere Meldung, die Felder werden zurückgesetzt und mit Hilfe der bereits bei der Passwort-vergessen-Funktion verwendeten Klasse *JavaMailer* eine E-Mail verschickt. Anschließend ist das Formular direkt bereit für die nächste Anfrage.

Leider gilt es erneut eine zusätzliche Einschränkung zu beachten. Das Kontaktformular darf weder beim Laden der Seite noch bei angezeigter Seite nicht ausgeblendet sein, sonst verweigert die AJAX-Bibliothek auch an dieser Stelle ihren Dienst. Deswegen wird dem *a4j:outputPanel* mit der ID „inc_copyright“ die CSS-Klasse „hidden“ zugewiesen um diesen Teil des Footers auszublenden und standardmäßig das Formular anzuzeigen. Auch an anderen Stellen wird dieser Bug noch auftauchen.

3.2.6 Ein neues Benutzerkonto anlegen

Bevor das Registrierungsformular abgeschickt wird, müssen die Eingaben in fünf Feldern korrekt sein. Durch das Hinzufügen eines Elements des Typs *a4j:support* wird bei jedem Tastenanschlag die Eingabe überprüft und eventuelle Fehlermeldungen angezeigt. Dies sieht am Beispiel der E-Mail-Feldes folgendermaßen aus:

```
<h:outputLabel for="field_email" value="#{msg.label_email}" />
<h:inputText id="field_email" value="#{register.email}">
    <a4j:support event="onkeyup" reRender="msg_email" />
</h:inputText>
<h:outputText id="msg_email" value="#{register.msg_email}" />
```

Listing 3.7: Das Eingabefeld löst nun bei jedem Tastenanschlag einen asynchronen Request aus

Da dabei das Feld „msg_email“ aktualisiert wird, muss nun innerhalb der Methode *getMsg_email()* die Prüfung der Benutzereingabe vorgenommen werden.

```
public String getMsg_email() {
    // email pattern string
    Pattern p = Pattern.compile(".+@.+\\.[a-z]+");
    Matcher m = p.matcher( email );
    boolean matchFound = m.matches();

    if (!matchFound) {
        setMsg_email( this.rb.getString("msg_insertvalidemail") );
    } else if( emailUsed() ) {
        setMsg_email( this.rb.getString("msg_emailregisteredalready") );
    } else {
        setMsg_email("");
    }
    return msg_email;
}
```

Listing 3.8: Prüfung der E-Mail-Adresse innerhalb der Methode *getMsg_email()*

Nachdem die E-Mail-Adresse auf syntaktische Korrektheit hin geprüft wurde, erfolgt noch der Abgleich mit der Datenbank, ob die eingegebene Adresse nicht schon registriert ist. Aufgrund der Anforderung, dass nur eine Registrierung pro E-Mail-Adresse möglich sein soll, bedeutet ein neuerlicher Registrierungsversuch ebenfalls einen Fehler.

```
public boolean emailUsed() {
    boolean ret = false;

    try {
        PreparedStatement stmt_checkusername =
this.conn.prepareStatement("SELECT COUNT(`ID`) FROM `user` WHERE
`email`=?");
        stmt_checkusername.setString( 1, this.email );
        ResultSet rs_checkusername = stmt_checkusername.executeQuery();

        while( rs_checkusername.next() ) {
            int cnt = rs_checkusername.getInt(1);
            if( cnt>0 ) {
                ret = true;
            }
        }
    } catch( Exception e ) {}

    return ret;
}
```

Listing 3.9: Ist die Adresse schon registriert, wird *true* zurückgegeben

Solange noch eines der fünf Felder Fehler enthält, soll das Formular nicht abgeschickt werden können. Deshalb muss auch der Status der Absenden-Schaltfläche nach jeder Änderung in einem Textfeld aktualisiert werden, weshalb beim Eingabefeld zusätzlich auch noch die ID der Schaltfläche im *reRender*-Attribut mit angegeben wird.

```
<h:inputText id="field_email" value="#{register.email}">
    <a4j:support event="onkeyup" reRender="msg_email, btn_register" />
</h:inputText>
```

Listing 3.10: Das a4j:support-Element wird um die ID der Absenden-Schaltfläche erweitert

Die Schaltfläche erkennt an den gesetzten bzw. nicht gesetzten Fehlermeldungen ob alle Eingaben korrekt sind oder noch Fehler vorliegen.

```java
public boolean isDisableSubmit() {
    boolean allOK = msg_realname.equals("") &&
                    msg_email.equals("") &&
                    msg_username.equals("") &&
                    msg_password1.equals("") &&
                    msg_password2.equals("");
    setDisableSubmit( !allOK );

    return disableSubmit;
}
```

Listing 3.11: Sobald es keine Fehlermeldungen mehr gibt, wird die Absenden-Schaltfläche aktiv

Da in dieser Arbeit das Thema AJAX im Vordergrund steht, wäre es nun noch schön, wenn auf das Speichern asynchron und ohne Neuladen der Seite geschehen könnte. Schließlich haben wir ja den *a4j:commandButton*, der diese Aufgabe übernehmen kann. Doch in unserem Fall funktioniert dies aufgrund eines wahrscheinlichen Fehlers im Framework leider nicht. Sobald wir die Schaltfläche mit dem Attribut *disabled* versehen und sie deshalb mindestens einmalig inaktiv war, führt sie die im Attribut *action* angegebene Funktion aus unerklärlicher Weise nicht aus, erst ohne dieses Attribut funktioniert der asynchrone Aufruf wieder. Ein ähnliches Phänomen haben wird schon im Footer-Bereich (siehe 3.2.5) gesehen.

Es bleibt also nur die Wahl zwischen einem asynchronen Request und der asynchronen Eingabevalidierung, die die Schaltfläche erst dann aktiv schaltet, wenn alle Eingaben fehlerfrei sind. In diesem Fall ist der synchrone Request weniger problematisch, da das komplette Neuladen der Seite nicht wirklich störend wirkt. Deshalb kann der *h:commandButton* Verwendung finden:

```xml
<h:commandButton id="btn_register" value="#{msg.btn_register}"
    action="#{register.register}" disabled="#{register.disableSubmit}" />
```

Listing 3.12: Ein *h:commandButton* löst synchron die Registrierung aus

Innerhalb der Methode *register()* wird dann schließlich der neue Benutzer gespeichert, ohne dass ein Fehler auftritt und zum Schluss eine entsprechende Erfolgsmeldung ausgegeben.

```java
public String register() {
    try {
        PreparedStatement stmt_registeruser =
this.conn.prepareStatement("INSERT INTO `user` (`realname`, `email`,
`username`, `password`) VALUES(?, ?, ?, ?)");
        stmt_registeruser.setString( 1, realname );
        stmt_registeruser.setString( 2, email );
        stmt_registeruser.setString( 3, username );
        stmt_registeruser.setString( 4, password1 );
        stmt_registeruser.execute();

        // reset input fields

        setMessage_type("success");
        setMessage( this.rb.getString("msg_registrationsuccessfull") );
    } catch( SQLException e ) {
        setMessage_type("error");
        setMessage( this.rb.getString("msg_registrationfailed") );
    }
    return getMessage_type();
}
```

Listing 3.13: Der neue Benutzer wird in der Datenbank gespeichert

3.2.7 Die Passwort-vergessen-Funktion für Vergessliche

Auf dieser Seite wird nur ein ein einfaches Formular mit einem Textfeld inkl. Fehlermeldung sowie eine Schaltfläche benötigt.

```
<h:inputText id="mail" value="#{passwordlost.email}">
    <a4j:support event="onkeyup" reRender="msg_mail, btn_passwordlost" />
</h:inputText>
<h:outputText id="msg_mail" value="#{passwordlost.msg_email}" />
<h:commandButton id="btn_passwordlost" value="Neues Passwort anfordern"
                 styleClass="btn" action="#{passwordlost.sendNewPassword}"
                 disabled="#{passwordlost.disableSubmit}" />
```

Listing 3.14: Der Quellcode des Passwort-vergessen-Formulars

Das Eingabefeld erzeugt durch die Erweiterungen per *a4j:support* einen asynchronen Request bei jedem Tastenanschlag, nach dem Fehlermeldung und Schaltfläche aktualisiert werden. Die Schaltfläche wird dabei anhand des Rückgabewerts der Methode *isDisableSubmit()* aktiv oder inaktiv geschaltet. Mittels einem temporären Objekt der Klasse *RegisterHandler* wird geprüft, ob eine E-Mail-Adresse bereits registriert ist oder nicht.

```
public boolean isDisableSubmit() {
    // create an object of RegisterHandler to check if mail is registered
    RegisterHandler tmp_registerObj = new RegisterHandler();
    tmp_registerObj.setEmail( this.email );
    setDisableSubmit( !tmp_registerObj.emailUsed() );

    return disableSubmit;
}
```

Listing 3.15: Die Schaltfläche wird erst bei korrekter E-Mail-Adresse aktiviert

Die Fehlermeldung hängt dann in diesem Fall vom Status der Schaltfläche ab, bei inaktiver Schaltfläche muss also die Methode *getMsg_email()* die Fehlermeldung zurück liefern.

Auch hier verhält sich leider die Anwendung nicht so wie erwartet, die Methode *sendNewPassword()* wird aus fraglichem Grund nicht aufgerufen. Allerdings kann an dieser Stelle der Funktionsaufruf über einen kleinen Umweg erfolgen: in der GET-Methode der anzuzeigenden Erfolgs- bzw. Fehlermeldung:

```
public String getMessage() {
    if( getMsg_email()=="" ) {
        sendNewPassword();
    }
    return message;
}
```

Listing 3.16: Ist die Fehlermeldung nicht gesetzt, wird *getMsg_email()* aufgerufen

In sendNewPassword() wird dann ein zufälliges Passwort generiert, das nach dem Speichern dem Benutzer auch noch per E-Mail zugeschickt wird. Hierzu wird die Klasse *JavaMailer* mit ihrer statischen Methode

```
postMail( String recipient, String subject, String message, String from )
```

benutzt.

3.2.8 Einloggen mit den korrekten Zugangsdaten

Zum Login ist die korrekte Kombination aus Benutzername und Passwort nötig. Den Benutzernamen wird auch hierbei wieder per AJAX überprüft, wenn anschließend auch noch ein Passwort eingegeben wurde, erfolgt der Ausrufe der Methode *login()*, die die eingegeben Daten gegen die Datenbank testet. Im Erfolgsfall werden Benutzername und Passwort in der Session gespeichert und auf die Überblickseite weitergeleitet. Tritt ein Fehler auf, bleibt der Benutzer auf der Login-Seite und bekommt zusätzlich eine Fehlermeldung zu Gesicht.

```
public String login() {
    boolean ret = false;

    if( this.password.length()>4 ) {
        try {
            PreparedStatement stmt_trylogin =
this.conn.prepareStatement("SELECT COUNT(`ID`) FROM `user` WHERE
`username`=? AND `password`=?");
            stmt_trylogin.setString( 1, getUsername() );
            stmt_trylogin.setString( 2, getPassword() );
            ResultSet rs_trylogin = stmt_trylogin.executeQuery();

            while( rs_trylogin.next() ) {
                int cnt = rs_trylogin.getInt(1);
                if( cnt==1 ) {
                    ret = true;
            } }
        } catch( Exception e ) {
            e.printStackTrace();
        }
    }

    if( !ret ) {
        setMessage_type("error");
        setMessage( this.rb.getString("msg_loginfailed") );
        return "error";
    } else {
        User loginUser = new User( getUsername() );
        loginUser.updateLogin();
        Session.setUsername( loginUser.getUsername() );
        Session.setPassword( loginUser.getPassword() );
        return "success";
    }
}
```

Listing 3.17: Die Login-Daten des Benutzers werden überprüft

Die Weiterleitung erfolgt über Navigationsregeln zu dieser Seite, die für die beiden Rückgabewerte innerhalb einer Konfigurationsdatei definiert wurden.

3.2.9 Aufgaben hinzufügen und verwalten

Die Verwaltung der Aufgaben gliedert sich in Hinzufügen, Ansehen und Löschen. Diese Vorgänge werden dank Ajax4jsf komplett ohne Reload der Seite ausgeführt.

Doch bevor die Seite angezeigt wird, müssen zunächst überprüft werden, ob der Benutzer die nötigen Zugriffsrechte besitzt, wobei der in der Session gespeicherte Benutzername sowie das Passwort verwendet wird. Die statische Methode *checkAccess()* aus *MuchToDoSession* liefert

im Erfolgsfall *true* zurück. per einfache JSP-Anweisung kann dann entweder die Umleitung zur Login-Seite oder die Anzeige der Übersichtsseite erfolgen:

```
<%
if( !ToDoListHandler.checkAccess( MuchToDoSession.getUsername(),
                                  MuchToDoSession.getPassword() ) ) {
    MuchToDoSession.setLoginerror( true );
    %>
    <jsp:forward page="/pages/login.jsf" />
    <%
} else {
    // Übersichtsseite anzeigen
}
%>
```

Listing 3.18: Prüfung des Zugriffsberechtigung am Seitenbeginn

Neue Aufgaben hinzufügen

Zu einer Aufgabe müssen Titel und Tags angegeben werden. Die Problematik mit der Kombination eines *a4j:commandButton*, der zuvor deaktiviert war, und einem anschließenden asynchronen Request (siehe 3.2.6, Seite 25) zwingt dazu, in diesem Fall keine dynamischen Fehlermeldungen hinter den fehlerhaften Feldern anzuzeigen um damit die Funktionaliätet des AJAX-Buttons sicherzustellen. Etwaige Eingabefehler werden stattdessen vom asynchronen Request erkannt und mit der entsprechenden Fehlermeldung quittiert.

Innerhalb der Speichern-Methode *addTask()* muss zum Speichern zunächst eine neue Instanz der Klasse *Task* instantiiert werden, der die über das Formular eingegebenen Werte zugewiesen werden. Die Methode *save()* dieses Objekts informiert darüber, ob die Werte korrekt waren und die Daten erfolgreich gespeichert werden konnten oder ob dies nicht der Fall war. Im Fehlerfall wird eine entsprechende Fehlermeldung angezeigt, ansonsten werden die Felder des Formulars zurückgesetzt und der Erfolg der Aktion dem Benutzer mit einer Erfolgsmeldung mitgeteilt.

```
public void addTask() {
    // create a new instance
    Task nTask = new Task();
    nTask.setData( getTitle(), getDescription(), getTags(),
                   Session.getUsername() );

    // try to save and handle return value
    if( nTask.save() ) {
        setMessage_type("success");
        setMessage( this.rb.getString("msg_taskadded").replace("?",
                    getTitle()) );
        setTitle("");
        setDescription("");
        setTags("");
    } else {
        setMessage_type("error");
        setMessage( this.rb.getString("msg_erroraddingtask") );
    }
}
```

Listing 3.19: Abhängig vom Erfolg des Hinzufügens wird eine entsprechende Meldung ausgegeben

Damit die Ergebnisse des asynchronen Requests für den Benutzer direkt sichtbar werden, müssen beim Klick auf die Schaltfläche einige Elemente aktualisiert werden. Das Element vom Typ *a4j:commandButton* sieht aus diesem Grund wie folgt aus:

```
<a4j:commandButton id="btn_addtask" reRender="message, newtask_title,
     newtask_desc, newtask_tags, alltasks, pagination_previous,
     pagination_next" value="#{msg.btn_addtask}" styleClass="btn" />
```

Listing 3.20: Nach dem Hinzufügen werden zahlreiche Elemente der Seite aktualisiert

Bei „message" handelt es sich um die Meldung, „newtask_title", „newtask_description" und „newtask_tags" repräsentieren die Eingabefelder. Die Bedeutung der übrigen Elemente wird im nächsten Abschnitt beschrieben.

Vorhandene Aufgaben

Für die Darstellung der vorhandenen Aufgaben eines Benutzers ein Element vom Typ *h:dataTable* verwendet, welches in diesem Fall wie folgt aussieht:

```
<h:dataTable id="alltasks" value="#{todolist.tasks}" var="task"
    columnClasses="right, null, null, null" rowClasses="zebra_odd,
    zebra_even" style="width:100% !important;" styleClass="t10" rows="5"
    first="#{todolist.offset}">
    <h:column>
        <f:facet name="header"><h:outputText value="ID" /></f:facet>
        <h:outputText value="#{task.id}" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{msg.text_task}" />
        </f:facet>
        <h:outputText value="#{task.title}" title="#{task.desc}" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{msg.label_tags}" />
        </f:facet>
        <h:outputText value="#{task.tags_string}" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{msg.text_added}" />
        </f:facet>
        <h:outputText value="#{task.date_string}" />
    </h:column>
</h:dataTable>
```

Listing 3.21: Der Quellcode der vierspaltigen Ergebnisliste

Da es nicht möglich ist Zellen in einer *h:dataTable* zu schachteln, muss die Beschreibung der Aufgabe im *title*-Tag angegeben werden. Die Standardeinstellungen der Browser sorgen dann dafür, dass beim Überfahren mit dem Mauszeiger die Beschreibung als Tooltip angezeigt wird.

Dem Attribut *first* wurde bereits der Wert *offset* zugewiesen, der für die Navigation innerhalb einer größeren Liste von Aufgaben benötigt wird. Über zwei Schaltflächen des Typ *a4j:commandButton* kann den einzelnen Ergebnisseiten navigiert werden, indem jeweils die Variable *offset* erhöht oder erniedrigt und im Anschluss die Liste aktualisiert wird.

```
<a4j:commandButton id="pagination_previous"
    action="#{todolist.previousPage}" value="#{msg.btn_previous}"
    reRender="alltasks, pagination_previous, pagination_next" />
```

Listing 3.22: Mit dieser Schaltfläche kann innerhalb der Ergebnisliste geblättert werden

Da sich wie schon gesehen eine Schaltfläche des Typs a4j:commandButton nicht mehr korrekt bedienen lässt sobald sie einmal inaktiv war, können wir die beiden Schaltflächen zur Navigation nicht aktiv und inaktiv schalten, das Blättern wäre sonst nur eingeschränkt möglich. Aus diesem Grund muss innerhalb der Funktionen *previousPage()* und *nextPage()* verhindert werden, dass die Variable *offset* 0 unter- oder den größtmöglichen Wert überschreitet.

```
public void previousPage() {
    if( getOffset()>(this.ITEMSPERPAGE-1) ){
        setOffset( getOffset()-this.ITEMSPERPAGE );
} }
```

Listing 3.23: Die Variable *offset* wird innerhalb ihrer Grenzen erhöht und erniedrigt

Nachdem nun schon in der Liste geblättert werden kann, soll im nächsten Schritt noch die Ergebnismenge über einen Suchbegriff einschränkt werden können. Hierzu gibt es ein Suchfeld, das bei jedem Tastenanschlag das Neuladen der Liste auslöst.

```
<h:inputText id="search" value="#{todolist.searchstring}">
    <a4j:support event="onkeyup" reRender="alltasks, message_alltasks,
        pagination_previous, pagination_next" />
</h:input>
```

Listing 3.24: Jede Änderung im Suchfeld aktualisiert die Ergebnisliste

Die Methode *getTasks()* der Klasse *User* bekommt dann auch den Suchbegriff übergeben um die gewünschten Aufgaben zurückliefern zu können. Diese Methode wird von der gleichnamigen Methode der Klasse *ToDoListHandler* aufgerufen, die zusätzlich noch die Nummer der aktuellen Ergebnisseite übergeben bekommt um die richtigen Aufgaben zurückzuliefern.

```
public List getTasks( String searchfor ) { ... }
```

Listing 3.25: Die Methode *getTasks()* der Klasse *User* liefert alle Aufgaben zu einem Suchbegriff

```
public List getTasks() {
    this.tasks = new ArrayList<Task>();

    if( !searchstring.equals("") ) {
        setTasks( this.user.getTasks( searchstring ) );
    } else {
        setTasks( this.user.getTasks( "" ) );
    }
    if( this.tasks.size()==0 ) {
        setMessage_type("error");
        if( !searchstring.equals("") ) {
            setMessage( rb.getString("msg_notaskcriteria") );
        } else {
            setMessage( rb.getString("msg_notask") );
        }
     } else {
        setMessage("");
     }
     return this.tasks;
}
```

Listing 3.26: Innerhalb der Klasse *ToDoListHandler* wird dann *getTasks()* eines *User*-Objekts aufgerufen

Die Methode *unsetFilter()* stellt wieder die ungefilterte Ergebnisliste dar nachdem Sie über einen grafischen Link aufgerufen wurde.

```
<a4j:commandLink action="#{todolist.unsetFilter}" reRender="search,
                        alltasks, message_alltasks, pagination_previous,
                        pagination_next" value="#{msg.btn_resetfilter}">
<h:graphicImage value="/images/error.gif" alt="#{msg.btn_resetfilter}" />
</a4j:commandLink>
```

Listing 3.27: Ein Reset-Button setzt durch Aufruf von *unsetFilter()* Suchfeld und Tabelle zurück

Da es in JSF nicht möglich ist innerhalb einer *h:dataTable* dynamisch eine Schaltfläche darzustellen geschweige denn der damit verknüpften Funktion einen Parameter wie z. B. die ID einer Aufgabe zu übergeben, wird für das das Löschen von Aufgaben ein weiteres Formular mit einem Eingabefeld sowie einem *a4j:commandButton* für den asynchronen Request benötigt.

```
<h:outputLabel for="finish_id" value="#{msg.label_finishid}" />
<h:inputText id="finish_id" value="#{todolist.id_tofinish}" />
<a4j:commandButton value="#{msg.btn_finish}"
    action="#{todolist.finishTask}" reRender="alltasks, finish_id,
    pagination_previous, pagination_next, message_alltasks" />
```

Listing 3.27: Eine Aufgabe kann durch Eingabe der ID abgeschlossen werden

Innerhalb der aufgerufenen Methode wird geprüft, ob es sich bei der Eingabe um eine gültige ID einer dem Benutzer gehörenden Aufgabe handelt. In diesem Fall würde diese zusammen mit allen Beziehungen in der Datenbank gelöscht und eine Erfolgsmeldung angezeigt sowie das Eingabefeld zurückgesetzt. Tritt ein Fehler auf, so wird dies mit einer Fehlermeldung quittiert und die Eingabe in diesem Fall nicht zurückgesetzt.

```
public void finishTask() {
    int id;

    try {
        id = Integer.parseInt( getId_tofinish() );
        Task task_tofinish = new Task();
        task_tofinish.loadByID( id );

        if( task_tofinish.getTitle()!=null
            && task_tofinish.getUser().toLowerCase().equals(
                                  Session.getUsername().toLowerCase() ) ) {
            String task_title = task_tofinish.getTitle();
            task_tofinish.delete();

            setId_tofinish("");
            setMessage_type("success");
            setMessage( rb.getString("msg_taskfinished").replace( "?",
                                                          task_title ) );
        } else {
            setMessage_type("error");
            setMessage( rb.getString("msg_invalidtask") );
        }
    } catch( Exception e ) {
        setMessage_type("error");
        setMessage( rb.getString("msg_errorfinishingtask") );
    }
}
```

Listing 3.28: Die Aufgabe mit der eingegebene ID wird abgeschlossen wenn sie dem Benutzer gehört

Die Aufgaben des Benutzers lassen sich nun in gewünschter Weise verwalten, der in der Session gespeicherte Benutzername wird dabei immer wieder herangezogen um auch wirklich die dem Benutzer gehörenden Aufgaben anzuzeigen bzw. ihm neue zuordnen. Erst bei einem Logout werden diese Daten dann schließlich wieder zurückgesetzt und nach dem Klick auf den entsprechenden Link wieder die Login-Seite angezeigt, eine entsprechende Navigationsregel (siehe 2.2.4) muss dazu zuvor definiert worden sein.

```
public static String logout() {
    setUsername("");
    setPassword("");

    return "login";
}
```

Listing 3.29: Die *logout()*-Methode der Klasse *Session*

3.3 Realisierung mit PHP und Prototype

Um einen Vergleich zwischen der Realisierung einer Webapplikation in JSF mit Ajax und PHP mit Ajax einigermaßen fair ziehen zu können, wird die Beispielapplikation ein zweites Mal in PHP und der AJAX-Bibliothek Prototype umgesetzt. Der Vergleich hinkt zugegebener Weise ein wenig, denn während es sich bei Ajax4jsf um ein Java-Framework handelt, beim dem der Entwickler keine JavaScript-Kenntnisse benötigt, ist Letzteres bei Prototype der Fall, hier werden zumindest rudimentäre JavaScript-Kenntnisse benötigt. Je besser die Kenntnisse sind, desto feiner und besser lassen sich die Funktionen steuern.

3.3.1 Die Entwicklungsumgebung

Bei der Vorstellung von PHP (siehe 2.4) ist bereits angeklungen, dass für die Programmierung bereits ein einfacher Texteditor ausreicht. Nachdem aber die JSF-Implementierung unter Zuhilfenahme eines Eclipse-Plugins umgesetzt wurde, soll dies im zweiten Fall ebenso sein.

Für PHP gibt es mit PHPEclipse[3] und dem PDT Project[4] zwei gute Plugins. Funktional nehmen sich beide Plugins nicht viel, letzten Endes dürfte in vielen Fällen der eigene Programmierstil den Ausschlag geben, da beide Erweiterungen die Tag-Vervollständigung sowie Einrückungen etwas unterschiedlich handhaben. So muss Jeder für sich entscheiden, was den eigenen Gewohnheiten entgegen kommt oder auch nicht. Für die PHP-Implementierung wurde im Rahmen dieser Studienarbeit das PDT Project eingesetzt, das ursprünglich aus dem Hause Zend kommt – in PHP-Kreisen hat sich diese Firma bereits einen sehr guten Ruf erarbeitet.

3.3.2 Aufteilung der Anwendung

Die Beispielanwendung teilt sich im Wesentlichen in drei Bereiche auf. Während der eigentliche Inhalt von Seite zu Seite variiert, bleiben Header und Footer immer gleich.

Bei JSF gab es noch Probleme mit der Auslagerung des Footers sobald darin Ajax4jsf-Elemente vorkamen und das Framework hätte zusätzlich zur eigentlichen Seite ein weiteres Mal referenziert werden müssen (siehe 3.2.5). Mit solchen Problemen sieht man sich in PHP nicht konfrontiert, die redundanten Seitenbereiche lassen sich problemlos in separaten Dateien speichern und anschließend an der gewünschten Stelle einbinden. Der Code

```
include 'blocks/footer.php';
```

3 http://www.phpeclipse.net/
4 http://www.eclipse.org/php/

sorgt dafür, dass der komplette Quellcode der ausgelagerten Datei an der Stelle des Aufrufs eingefügt wird. In der Datei *footer.php* können deshalb problemlos Prototype-Funktionen benutzt werden, die AJAX-Bibliothek muss lediglich in der einbindenden Datei bekannt sein.

Damit reicht es aus eine *index.php* zu programmieren, die das Grundgerüst der Seite erstellt, statisch Header und Footer sowie dynamisch die jeweils passende Seite einbindet. In ihr werden bereits innerhalb des HEAD-Bereichs die nötigen CSS- und JavaScript-Dateien eingebunden. Zusätzlich gibt es noch zu jeder einzelnen Seite eine spezifische JavaScript-Datei mit gleichem Namen, die zusätzlich im BODY referenziert wird.

3.3.3 PHP-Sessions

In PHP existiert das super-globale Array *$_SESSION*, in dem beliebige Werte abgelegt werden können, die über mehrere Seiten hinweg benötigt werden. Da das Array von überall beliebig schreib- und lesbar ist, birgt dies natürlich auch ein gewisses Risiko. Wie bei allen Daten, die vom Benutzer kommen, sollten nach dem Grundsatz einer sicheren Programmierung auch alle Daten aus diesem Array vor der weiteren Verarbeitung überprüft werden.

Die Beispielapplikation nutzt das Session-Array zur Speicherung des angemeldeten Benutzers, der Sprache sowie des zuletzt aktiven Tabs im Footer. Die Überblickseite testet bei jedem Aufruf den Benutzernamen und das Passwort gegen die Datenbank bevor sie angezeigt wird. Bei einem Fehler soll die Login-Seite samt einer Fehlermeldung angezeigt werden.

```
if( $page=='overview' &&
    !LoginHandler::login( $_SESSION['user'], $_SESSION['pw'] ) {
    $page         = 'login';
    $meesage_type = 'error';
    $message      = $lang['accessdenied'];
}
include "pages/$page.php";
```

Listing 3.30: Vor Aufruf der Übersichtseite werden die Zugangsdaten überprüft

Die Prüfung der Session-Variablen *user* und *pw* erfolgt an dieser Stelle – wie auch sonst überall in dieser Applikation – direkt vor dem Einfügen in die SQL-Abfrage über die Klassenmethode *Str2Sql()* der Klasse *Database.*

```
public static function Str2Sql( $str ) {
    return mysql_escape_string( stripcslashes($str) );
}
```

Listing 3.31: Die Methode *Str2Sql()* „entschärft“ Variablen vor der Datenbankabfrage

Somit werden gefährliche Zeichen maskiert und die Gefahr einer SQL-Injection auf ein Minimum reduziert.

Um in einer PHP-Datei auf das Session-Array zugreifen zu können, muss als erste Anweisung und zwingend vor dem ersten *echo*- oder *print*-Befehl „session_start()“ aufgerufen werden. Der Aufruf bewirkt, dass die bestehende Session wieder aufgenommen bzw. beim ersten Aufruf eine neue Session erzeugt wird.

3.3.4 Sprachwahl und -wechsel

Wie soeben erwähnt, speichert die Session auch die aktive Sprache, hierzu wird im Session-Array das Feld *language* benutzt. Ist keine Sprache gewählt, so wird als Standard Deutsch verwendet. Abhängig von der gewählten Sprache erfolgt dann das Einbinden der entsprechenden Sprachdatei.

```
// require translation file
if( $_SESSION['language']=='de' ) {
    $lang_file = 'german.lang.php';
} else {
    $lang_file = 'english.lang.php';
}
require_once( 'lang/'.$lang_file );
```

Listing 3.32: Anhand der Session-Variable *language* wird die richtige Sprachdatei eingebunden

Über zwei kleine Flaggen in der rechten oberen Ecke des Browserfensters kann die Sprache jederzeit umgeschaltet werden. Aufgrund der Vielzahl der Übersetzungen ist dieser Vorgang ohne AJAX realisiert. Der dem URL angehängte Parameter *language* signalisiert den Wunsch der Sprachänderung. Nach der Prüfung gegen eine Whitelist, wird die Sprache in Session-Array geändert bevor die Sprachdatei eingebunden wird (siehe Listing 3.32).

```
if( $language = $_REQUEST['language'] ) {
    $language_allowed = array('de', 'en');
    $_SESSION['language'] = (in_array( $language, $language_allowed ) ?
                             $language : $lang_allowed[0]);
}
```

Listing 3.33: Nach der Prüfung gegen eine Whitelist wird die Sprache geändert

3.3.5 „Footer wechsel dich”

Mit PHP und Prototype ist es ein Leichtes zwischen den zwei disjunkten Bereichen des Footers zu wechseln ohne dabei die restliche Seite oder auch die Funktion innerhalb des Footers in irgendeiner Weise zu beeinträchtigen. Hierzu werden zwei DIV-Elemente mit dem jeweiligen Inhalt und jeweils einer eindeutigen ID angelegt. Beim Klick auf den Link „Copyright“ wird dann per *onClick*-Event eine JavaScript-Routine aufgerufen, die das Kontakt-DIV aus- und das Copyright-DIV einblendet.

```
<a href="" onclick="return showCopyright();">Copyright</a>
```

```
function showCopyright() {
    $('footer_contact').addClassName('hidden');
    $('footer_copyright').removeClassName('hidden');
    ...
    return false;
}
```

Listing 3.44: Über CSS-Klassen wird ein Footer-Bereich aus-, der andere eingeblendet

Eine analoge Funktion *showContact()* sorgt für den entgegengesetzten Effekt.

Ein zusätzlicher AJAX-Request sorgt dafür, dass in der Datei *update_state_footer.ajax.php* der aktuelle Zustand des Footers in der Session gespeichert und somit bei einem Seitenwechsel nicht zurückgesetzt wird. Einen Rückgabewert gibt es in diesem Fall nicht.

```
function updateStateFooter( newstate ) {
    var params = 'newstate='+newstate;

    var update_state_footer = new Ajax.Request(
        'ajax/update_state_footer.ajax.php',
        {
            parameters:params
        } );
}
```

Listing 3.45: Asynchron wird durch den Request der neue Status in der Session gespeichert

Der anzuzeigende Bereich wird dann bei jedem Aufruf der Seite aus der Session ermittelt, gegen die Whitelist der erlaubten Werte geprüft und sichtbar gemacht. Auch der zugehörige Link wird optisch per CSS-Klasse „active" gekennzeichnet.

```
$tab_allowed = array('copyright', 'contact');
$tab = $_SESSION['state_footer'];
if( !in_array( $tab, $tab_allowed ) ) {
    $tab = $tab_allowed[0];
}
```

Listing 3.46: Der anzuzeigende Bereich wir aus der Session-Variable gelesen

Innerhalb des Footers ist auch das Kontaktformular enthalten. Der Benutzer kann bei Fragen oder Problemen Kontakt mit dem Programmierer aufnehmen. Auch hier werden die Felder mit der bekannten Technik auf Korrektheit geprüft bevor die E-Mail per AJAX abgeschickt werden kann. Das Versenden übernimmt die Datei *send_contactform.ajax.php*. Mit den beim asynchronen Aufruf als Parameter übergebenen Parametern wie Name und E-Mail-Adresse des Benutzers sowie dem Betreff und der eigentlichen Nachricht wird die PHP-Funktion *mail()* aufgerufen und die E-Mail versendet.

```
mail('koch0025@hs-karlsruhe.de',
     $lang['requestfrommuchtodo'].': '.$subject,
     $text,
     'FROM: '.$name.' <'.$email.'>' );

header("Content-type: text/xml");
echo '<xml>
         <msg><![CDATA['.htmlentities( $lang['thanxformail'] ).']]></msg>
    </xml>';
```

Listing 3.47: Die E-Mail wird gesendet und die Meldung als XML zurückgeliefert

Die Nachricht innerhalb der zurückgelieferten XML-Datei wird zum Abschluss im Footer dargestellt und alle Eingabefelder werden zurückgesetzt.

3.3.6 Die Registrierung

Validierungen per asynchroner Requests helfen zu gewährleisten, dass das Registrierungsformular erst dann abgeschickt werden kann, wenn alle in 3.1.1 genannten Bedingungen erfüllt sind. Jedes per ID eindeutig identifizierbare Eingabefeld besitzt dabei noch ein SPAN-Element zur Anzeige eventueller Fehler.

Am Beispiel des E-Mail-Feldes soll die AJAX-Validierung beispielhaft ausgeführt werden. Hierzu bekommt das Eingabefeld per

```
$('field_email').onkeyup = check_email;
```

die JavaScript-Funktion *check_email()* zugewiesen, die beim Eintritt des *onKeyUp*-Events aufgerufen wird. Diese erzeugt einen asynchronen Request auf die PHP-Datei *check_email.ajax.php* und übergibt die momentan eingegebene E-Mail-Adresse als Parameter. Geprüft wird die Adresse auf syntaktische Korrektheit und ob es sich um eine bereits registrierte Adresse handelt. Bei einem auftretenden Fehler wird die entsprechende Fehlermeldung unter der HTTP-Statuscode 400 zurückgeliefert, anhand dessen in JavaScript eine Fehlerroutine aufgerufen werden kann. In diesem Fall bedeutet dies, dass die Fehlermeldung hinter dem Textfeld angezeigt werden soll.

```
$email   = trim( $_REQUEST['email'] );
try {
    if( preg_match("!^\w[\w|\.|\-]+@\w[\w|\.|\-]+\.[a-zA-Z]{2,4}$!",
        $email)!=1 ) {
        throw new Exception( $lang['msg_validaddress'] );
    }
    if( RegisterHandler::emailUsed($email) ) {
        throw new Exception( $lang['msg_emailalreadyused'] );
    }
} catch( Exception $e ) {
    header("HTTP/1.0 400 Bad Request");
    header("Content-type: text/xml");
    echo '<xml>
            <msg><![CDATA['.htmlentities( $e->getMessage() ).']]></msg>
        </xml>';
}
```

Listing 3.48: Die E-Mail-Adresse wird in *check_email.ajax.php* auf Syntax und Eindeutigkeit geprüft

Die aufrufende JavaScript-Funktion kann im Fehlerfall die Fehlermeldung hinter dem jeweiligen Eingabefeld anzeigen, indem die Antwort per *onFailure* verarbeitet wird.

Ähnliche Validierungen gibt es auch zu allen anderen Eingabefeldern, die Methode *updateSubmit()* kann anhand der (nicht) gesetzten Fehlermeldungen entscheiden, ob die Absenden-Schaltfläche aktiv oder inaktiv sein soll und wird deshalb zu Abschluss jedes einzelnen Validierungsschritts für ein Textfeld aufgerufen.

Auch die eigentliche Registrierung erfolgt schließlich per AJAX. Dazu wird das Absenden des Formulars abgefangen und per JavaScript asynchron die Datei *register.ajax.php* aufgerufen, wobei die eingegebenen Daten als Parameter übergeben werden. Das

```
return false
```

ist dabei wichtig, damit das Formular nicht synchron abgeschickt wird und die Seite neu lädt.

```
$('form_register').onsubmit = function() {
    var params = 'realname='+$F('field_realname')+'&email='+
                 $F('field_email')+'&username='+$F('field_username')+
                 '&password='+$F('field_password1');


    var check_realname = new Ajax.Request(
        'ajax/register.ajax.php',
        {
            parameters:params,
            onComplete:function(r) {
                var msg_type =
r.responseXML.getElementsByTagName('msg_type')[0].childNodes[0].nodeValue;
                var msg      = r.responseXML.getElementsByTagName('msg')
[0].childNodes[0].nodeValue;

                // Fehler- bzw. Erfolgsmeldung anzeigen und bei Erfolg
                // Eingabefelder zurücksetzen
        } );

    return false;
}
```

Listing 3.49: Registrierung durch einen asynchronen Request

In der Datei *register.ajax.php* wird durch den Aufruf

```
RegisterHandler::register($realname, $email, $username, $password)
```

versucht, den neuen Benutzer in der Datenbank zu speichern. Anhand des boolschen Rückgabewerts kann festgestellt werden, ob die Registrierung erfolgreich war oder fehlschlug, was bei der Generierung der Antwort von Bedeutung ist. Daraufhin kann die Erfolgsmeldung oder zusammen mit dem HTTP-Statuscode 400 die Fehlermeldung zurückgegeben werden.

3.3.7 Neues Passwort mailen per PHP

Bereits bei der Eingabe erfolgt per AJAX die Prüfung, ob es sich bei der eingegebenen E-Mail-Adresse um eine syntaktisch korrekte und zudem registrierte Adresse handelt. Schon bei der Registrierung wurde auf ein ähnliches Kriterium per AJAX geprüft (Listing 3.48). Damals ging es darum eine noch nicht registrierte E-Mail-Adresse eingegeben zu haben, diesmal ist es genau anders herum. Durch die Einführung eines zusätzlichen Paramters, der dem PHP-Skript den aktuellen Kontext mitteilt, kann das bisherige Listing für beide Szenarien verwenden.

```
if( RegisterHandler::emailUsed($email) && $context!='passwordlost' ) {
    throw new Exception( $lang['msg_emailalreadyused'] );

} elseif( !RegisterHandler::emailUsed($email)
          && $context=='passwordlost' ) {
        throw new Exception( $lang['msg_emailnotregistered'] );
}
```

Listing 3.50: Durch den zusätzlichen Parameter *$context* ist das Skript nun universeller einsetzbar

Im Falle des Kontextes „passwordlost" wird ein Fehler gemeldet wenn die E-Mail-Adresse nicht registriert ist, andernfalls, wenn zu ihr schon ein Benutzerkonto existiert.

Auch das eigentliche Senden des neuen Passworts geschieht per AJAX. Nachdem die E-Mail-Adresse korrekt ist, kann das Formular abgeschickt werden. Per *onSubmit*-Event wird der Submit abgefangen und im Hintergrund die Datei *send_new_pw.ajax.php* aufgerufen.

In der PHP-Datei wird dann zunächst mittels *RegisterHandler::generatePassword()* ein neues Passwort generiert und anschließend mit der PHP-Funktion

```
mail( $to, $subject, $message, $additional_headers )
```

an die angegebene E-Mail-Adresse gesendet. Im Parameter *$additional_headers* kann u. a. die Absenderadresse angegeben werden, die anderen Parameter sind selbsterklärend. Von nun an ist ein Login nur noch mit dem neuen Passwort möglich.

3.3.8 Einloggen in die ToDo-Liste

Im Login-Formular erhält der eingegebene Benutzername ein AJAX-Prüfung auf Korrektheit und analog zur Prüfung der E-Mail-Adresse in 3.3.7 wird hierzu die bestehende Datei *check_username.ajax.php* um einen Kontext erweitert, um unterscheiden zu können ob der Benutzername neu sein (Registrierung) oder existieren (Login) muss um als korrekt zu gelten.

Anschließend erfolgt auch die Prüfung auf die korrekte Kombination von Benutzername und Passwort asynchron im Hintergrund durch den Aufruf der Datei *login.ajax.php* wie gewohnt über ein Objekt der Klasse *Ajax.Request.* Bei korrekter Antwort erfolgt die Weiterleitung auf die Überblickseite, andernfalls eine Fehlerausgabe:

```
$('form_login').onsubmit = function() {
    var params = 'username='+$F('field_username')+
                 '&password='+$F('field_password');
    var check_username = new Ajax.Request(
        'ajax/login.ajax.php',
        {
            parameters:params,
            onSuccess:function(r) {
                top.location.href = 'overview.html';
            },
            onFailure:function(r) {
                var msg = r.responseXML.getElementsByTagName('msg')
[0].childNodes[0].nodeValue;
                $('message').addClassName('error');
                $('message').innerHTML = msg;
            }
        } );

    return false;
}
```

Listing 3.51: Auch der Login erfolgt per asynchronem Aufruf

Bei korrekten Zugangsdaten werden in *login.ajax.php* die Session-Variablen *user* und *pw* gesetzt um die Zugriffsrechte auf der folgenden Seite überprüfen zu können. Die Rücksetzung dieser beiden Variablen erfolgt erst wieder beim Logout.

3.3.9 Aufgaben verwalten

Auch diese Seite lässt sich dank der Symbiose zwischen PHP und Prototype komplett ohne Seiten-Reload umsetzen. Die zwei FIELDSET-Elemente teilen die Seite auch optisch in die Bereiche „Neue Aufgabe hinzufügen“ und „Vorhandene Aufgaben“ und agieren vollständig unabhängig voneinander.

Neue Aufgabe hinzufügen

Gemäß den Spezifikationen aus 3.1.4 handelt es sich bei den Feldern „Titel“ und „Tags“ um Pflichtfelder, eine korrekte Eingabe wird angenommen, wenn mind. 5 Zeichen eingegeben wurden. Diese Prüfung lässt sich komplett clientseitig in JavaScript durchführen. Leider bietet JavaScript von Haus aus keine Funktion um Leerzeichen am Anfang und Ende einer Zeichenkette zu entfernen. Deshalb soll eine Funktion in der Datei */js/common.js*, die von allen Seiten eingebunden wird, diese Aufgabe erledigen, in Anlehnung an PHP erhält die Funktion den Namen *trim().*

```
function trim( string ) {
    while( string.substring( 0, 1 )==' ' ) {
        string = string.substring( 1, string.length );
    }
    while( string.substring( string.length-1, string.length )==' ' ) {
        string = string.substring( 0, string.length-1 );
    }
    return string;
}
```

Listing 3.52: trim() entfernt überflüssige Leerzeichen

Mit ihrer Hilfe kann nun im Event-Handler *onKeyUp* geprüft werden, ob ein Titel mit der geforderten Länge von 5 Zeichen eingegeben wurde oder nicht und entsprechend die Fehlermeldung ein- bzw. ausgeblendet werden.

```
$('newtask_title').onkeyup = check_title;
function check_title() {
    if( trim( $F('newtask_title')).length<5 ) {
        $('msg_title').show();
    } else {
        $('msg_title').hide();
    }
    updateSubmit();
}
```

Listing 3.53: Der Titel muss mind. 5 Zeichen lang sein

Analog funktioniert es auch für das Feld „Tags“. Die jeweils zum Abschluss der Routine aufgerufene Funktion *updateSubmit()* aktiviert die Absenden-Schaltfläche sobald beide Fehlermeldungen ausgeblendet sind.

```
function updateSubmit() {
    $('submit_newtask').disabled = !( $('msg_title').style.display=='none'
                                   && $('msg_tags').style.display=='none' );
}
```

Listing 3.54: Schaltfläche aktivieren wenn keine Fehler mehr vorliegen

Zum Speichern der Aufgabe wird wiederum ein AJAX-Request abgesetzt, der die eingegebenen Daten an die Datei *save_newtask.ajax.php* schickt. In ihr wird zunächst ein neues Objekt vom Typ *Task* erzeugt und die Werte zugewiesen, der zugehörige Benutzer wird dazu aus der Session ermittelt.

```
...
$nTask = new Task();
$nTask->setTitle( $title );
$nTask->setDesc( $desc );
$nTask->setTags( explode( ' ', $tags ) );
$nTask->setUser( $_SESSION['user'] );
if( !$nTask->save() ) {
    throw new Exception( $lang['errorsavingtask'] );
}
...
```

Listing 3.55: Neue Aufgabe speichern in der Datei *ajax/save_newtask.ajax.php*

Die korrekte Persistierung und Zuordnung der Tags in der Datenbank übernimmt die Klasse *Task*. Bereits beim Setzen der Tags werden – wenn nicht bereits geschehen – die übergebene Zeichenkette aufgeteilt und doppelte Elemente entfernt.

```
public function setTags( $tags ) {
    if( is_string($tags) ) {
        $tags = explode(' ', $tags);
    }
    $this->tags = array_unique( $tags );
}
```

Listing 3.56: Die Funktion *setTags()* der Klasse *Task*

Nach dem Speichern der Aufgabe kann deren neue ID ermittelt werden um durch das Durchlaufen einer Schleife die Beziehungen zwischen Aufgabe und Tags in der Datenbank abzubilden.

```
foreach( $this->tags as $tag ) {
    $tag_id = Tag::checkAdd( trim($tag) );
    $sqlsavetag = "INSERT INTO `task_tag`
                        VALUES (".$this->id.", ".$tag_id.")";
    mysql_query( $sqlsavetag );
}
```

Listing 3.57: Zwischen jedem Tag und der Aufgabe wird eine Beziehung erstellt

Die Klassenmethode *checkAdd()* liefert dabei die ID des übergebenen Tags zurück. Existiert das Tag in der Tabelle „Tag" noch nicht, so wird zunächst noch ein neuer Datensatz angelegt.

Vorhandene Aufgaben

Sehr einfach in JSF war die tabellarische Darstellung von Daten sowie eine Art Blättern-Funktion. Die Umsetzung dieser Anforderung erfordert bei PHP und Prototype etwas mehr Aufwand. Innerhalb der Datei *overview.php* wird zunächst ein DIV-Element mit der eindeutigen ID „alltasks" erzeugt, in das die Tabelle per AJAX geladen werden soll.

Den Bau der Tabelle übernimmt die Datei *show_table_tasks.ajax.php*, die von der JavaScript-Funktion *showTable()* per AJAX aufgerufen wird. Als Parameter werden der aktuelle Suchbegriff sowie die aktuelle Seitennummer übergeben, beide Parameter sind beim ersten Aufruf noch undefiniert.

Innerhalb *show_table_tasks.ajax.php* wird zunächst über den in der Session gespeicherten Benutzernamen ein neues *User*-Objekt erzeugt und auf diesem die Methode *getTasks()* aufgerufen, die Suchbegriff und Seitenzahl übergeben bekommt.

```
$tmp_user = new User();
$tmp_user->loadByUsername( $_SESSION['user'] );

// create table head

$tasks = $tmp_user->getTasks( $searchstring, $page );

foreach( $tasks as $task ) {
    // create a line for current task
}
```

Listing 3.58: Jede Aufgabe erscheint als eine Zeile in der Tabelle

Zunächst werden Tabellengerüst und Tabellenkopf angelegt, im Anschluss wird innerhalb der *foreach*-Schleife für jede Aufgabe eine neue Zeile generiert. Der gesamte Code wird zunächst in der Variablen *$output* zwischengespeichert. Sobald die Tabelle komplett ist, bekommt die noch zwei zwei versteckte Felder angehangen, die JavaScript für das spätere Wechseln zwischen den einzelnen Ergebnisseiten benötigen wird.

```
$cnt       = $tmp_user->getCntTasks( $searchstring );
$max_pages = ($cnt+1)%10+1;

$output .= '<input type="hidden" id="current_page" value="'.$page.'" />
            <input type="hidden" id="max_pages" value="'.$max_pages.' />';
```

Listing 3.59: Für die Blättern-Funktion nötige versteckte Felder werden angelegt

Das Feld mit der ID *current_page* enthält dabei die aktuelle Ergebnisseite, *max_pages* dagegen die zuvor ermittelte Anzahl der Ergebnisseiten. Nun kann das Ergebnis verpackt in eine XML-Datei zurückgeliefert und von der aufrufenden JavaScript-Funktion verarbeitet werden.

```
header("Content-type: text/xml");
echo '<xml>
    <content><![CDATA['.$output.']]></content>
    </xml>';
```

Listing 3.60: Die Ergebnistabelle wird innerhalb einer XML-Datei übertragen

Unterhalb der Tabelle werden nun zwei Schaltflächen zur Navigation zwischen den Ergebnisseiten generiert. Wann diese aktiv sind und wann nicht hängt von der aktuellen Ergebnisseite und der Gesamtanzahl Letzterer ab und wird nach jeder Aktualisierung der Tabelle clientseitig per JavaScript geprüft:

```
function updatePaginationActivity() {
    $('pagination_previous').disabled = ($F('current_page')==1);
    $('pagination_next').disabled = ($F('current_page')>=$F('max_pages'));
}
```

Listing 3.61: Die Schaltflächen zur Navigation zwischen den Ergebnisseiten werden aktualisiert

Die Schaltfläche „Zurück“ ruft bei einem Klick die Funktion *paginationPrevious()* auf, die den Wert der aktuellen Seite im versteckten Feld um 1 erniedrigt und die Aktualisierung der Tabelle anstößt. Analoges gilt auch für die Funktion *paginationNext()*, die den Wert aber erhöht.

```
function paginationPrevious() {
    $('current_page').value = parseInt( $F('current_page') )-1;
    showTable();

    return false;
}
```

Listing 3.62: Die aktuelle Seitenzahl wird erniedrigt und die Tabelle neu geladen

Ebenfalls eine Aktualisierung der Tabelle bewirkt jeder Tastenanschlag im Suchfeld. Der Event-Handler *onClick* sorgt dafür, dass auch in diesem Fall *showTable()* ausgeführt wird. Abgerundet wird die Möglichkeit zu filtern durch eine Reset-Funktion. Der momentan eingegebene Suchbegriff wird dazu gelöscht und die komplette Ergebnistabelle angezeigt.

```
function resetSearch() {
    $('field_search').value = '';
    showTable();

    return false;
}
```

Listing 3.63: Suchfeld und gefilterte Ergebnistabelle werden zurückgesetzt

Aufgaben abschließen

Alle Aufgaben lassen sich nun schon anzeigen und nach Belieben filtern. Damit die Liste aber nicht nur am Wachsen ist, braucht es auch noch die Möglichkeit Aufgaben als erledigt zu kennzeichnen und somit aus der Liste zu entfernen. Auch hierbei sorgt ein asynchroner Request über Prototype für die Lösung. Bereits bei der Generierung der Tabelle wurde am Ende jeder Zeile ein Link auf die JavaScript-Funktion *finishTask()* eingefügt, der die ID der Aufgabe übergeben wird:

```
onclick="return finishTask( 1 );"
```

Die Funktion initiiert nun also den asynchronen Aufruf der Datei *finish_task.ajax.php* und übergibt dabei die ID der Aufgabe.

```
function finishTask( taskId ) {
    var params = 'taskid='+taskId;
    var finish_task = new Ajax.Request(
        'ajax/finish_task.ajax.php',
        {
            parameters:params,
            onSuccess:function(r) {
                $('message_finish').addClassName('success');
                $('message_finish').removeClassName('error');
                var msg = r.responseXML.getElementsByTagName('msg')
[0].childNodes[0].nodeValue;
                $('message_finish').innerHTML = msg;
                showTable();
            },
            onFailure:function(r) {
                $('message_finish').addClassName('error');
                $('message_finish').removeClassName('success');
                var msg = r.responseXML.getElementsByTagName('msg')
[0].childNodes[0].nodeValue;
                $('message_finish').innerHTML = msg;
            }
        } );
    return false;
}
```

Listing 3.64: Die Funktion *finishTask()* löscht per AJAX-Request eine abgeschlossene Aufgabe

Sobald das PHP-Skript den entsprechenden Datensatz und alle Beziehungen in der Datenbank mithilfe der Funktion *delete()* der Klasse *Task* gelöscht hat wird die Tabelle neu geladen, die gelöschte Aufgabe wird nun wie zu erwarten nicht mehr dargestellt. Die in der XML-Antwort zurückgelieferte Erfolgs- oder Fehlermeldung wird dem Benutzer angezeigt.

4. Fazit

Nach den theoretischen Betrachtungen und Vergleichen zwischen JSF mit AJAX und PHP mit AJAX sowie der doppelten Implementierung der Beispielanwendung „MuchToDo“ bin ich nun am Ende angekommen und will die jeweiligen Vor- und Nachteile sowie meine Erfahrungen während der Studienarbeit kurz abschließend zusammenfassen.

4.1 Vor- und Nachteile der Realisierung mit JavaServer Faces

Zusammen mit dem MVC-Pattern bringt JSF auch die Programmiersprache Java in die Webprogrammierung ein. Durch einfache Aufgabenteilung lassen sich so optisch ansprechende Webanwendungen erzeugen, bei denen lediglich die Dateiendung auf die verwendete Programmiersprache schließen lässt. Nichts erinnert mehr an die Applets früherer Tage, die man lange mit Java im Internet in Verbindung brachte. Die Übertragung des in der Desktop-Programmierung schon lange üblichen Programmiermodells erleichtert Java-Programmierern den Wechsel. Sie können in völlig anderer Umgebung ihr bisherigen Kenntnisse und Erfahrungen sehr leicht einbringen.

Damit einher zwingt JSF auch zu einer sauberen Programmierweise und strikter Kapselung von Funktionen in Methoden. Die gesamte Anwendung wird dadurch erheblich wartbarer und auch nötiges Debugging über Ausgaben auf der Konsole hilft bei Fehlersuche und Optimierung ungemein.

Die in JSF erstellten Webanwendungen lassen sich durch das einfache Hinzufügen von AJAX-Features in der Bedienfreundlichkeit und Arbeitszeit erheblich optimieren. Durch den Einsatz von Ajax4jsf kann man sich zudem in Sachen Kompatibilität des JavaScript-Codes bezüglich der aktuellen Browser in Sicherheit wägen, ohne jemals auch nur eine Zeile geschrieben zu haben.

Leider erkauft man sich diese Vorteile mit einem erhöhten Zeitaufwand, bei Einsteigern kommt ein hohes Einarbeitungspensum hinzu und wer sich bisher mit Webdesign beschäftigt hat sieht sich zum Umdenken gezwungen, da JSF nicht semantisch, sondern in Tabellen denkt. Die komplexe Dateistruktur und die vielfältigen Abhängigkeiten erschweren die Arbeit zusätzlich und durch die immer wieder nötige Synchronisation mit dem Server muss die eigentliche Entwicklung immer wieder pausieren.

Mit Hilfe von Ajax4jsf lassen sich zwar eine Menge AJAX-Funktion ohne großen Aufwand integrieren, darüber hinausgehende Features können allerdings nicht selber hinzugefügt werden. Der erzeugte Quellcode funktioniert zwar in allen gängigen Browsern, ist aber eher chaotisch, nicht editierbar da automatisch generiert und dadurch auch nicht wartbar. Gleiches gilt für den HTML-Code. Dieser validiert zwar gegen die HTML-Spezifikationen, semantische Korrektheit, die beispielsweise auch die Nutzung der Anwendung mittels Screenreadern ermöglichen würde, ist aber etwas vollkommen Anderes. Zudem vermisst man als früherer Webdesigner JSF-Äquivalente für einige HTML-Elemente (siehe 2.2.9) und sieht sich zudem auch mit der Schwierigkeit konfrontiert, die Anwendung per CSS zu stylen, ohne die Elemente per ID ansprechen zu können (siehe 3.15).

Neben dem ein oder anderen fehlenden Feature nerven auch diverse „Kinderkrankheiten“. Einige Elemente lassen sich partout nicht miteinander kombinieren oder verhalten sich dann komplett anders als erwartet und auch die Entwicklungsumgebung lässt den Entwickler immer wieder im Stich (siehe 3.2.1).

4.2 Vor- und Nachteile der PHP/Prototype-Realisierung

Schon die Beschreibung der Implementierung in PHP/Prototype und die Erläuterungen zum Code lassen erahnen, dass diese zweite Realisierung in Punkto Lines of Codes einen größeren Umfang besitzt. Das zeigt sich schon an den zahlreichen AJAX-Requests, bei denen immer wieder der gleiche Code redundant getippt werden muss. Bei diesem Code handelt es sich zudem um JavaScript-Code, ohne entsprechende Kenntnisse kommt man hier also nicht weiter.

Mit jeder Zeile JavaScript-Code muss man sich zudem um die Kompatibilität mit den gängigen Browsern kümmern. Die heterogene Browserlandschaft macht es dem Entwickler dabei nicht einfach, neben diversen Alleingängen der jeweiligen Hersteller werden je nach Browser auch sehr unterschiedliche JavaScript-Versionen unterstützt. Was für den JavaScript-Code gilt, ist auch beim sonstigen HTML-Quellcode nicht anders. Zumindest mal, wenn man eine valide Seite anstrebt. Diesem Anspruch kann auch diese Studienarbeit nicht ganz gerecht werden, da teilweise durch die Nutzung von AJAX auch der Übersichtlichkeit wegen überflüssige Attribute weggelassen wurden, die aber im Standard vorgeschrieben sind. Das Thema Übersichtlichkeit stellt sich ebenso, da je nach Programmierart – PHP- in HTML-Code bzw. HTML- in PHP-Code – der Quelltext schnell unübersichtlich und zumindest für Dritte schwer verständlich und wartbar werden kann. Hier heißt es Disziplin zu wahren und nicht im Sinne des Extreme Programming jedes überflüssige Zeichen einzusparen, auch wenn PHP hierzu die Möglichkeiten bietet.

Zu guter Letzt sind auch die Sicherheitsprobleme von PHP bei schlampiger Programmierung zu nennen, die aber der Skriptsprache PHP nicht direkt anzulasten sind. Gerade Einsteigern wird es sehr leicht gemacht innerhalb kürzester Zeit Ergebnisse zu erzielen ohne dabei Sicherheitsaspekte zu beachten. Vom Benutzer kommende Daten müssen grundsätzlich geprüft werden, Variablen müssen vor der Nutzung in SQL-Abfragen „entschärft“ werden um keine unliebsamen Bekannschaften mit SQL-Injections oder Server-Hijacking zu machen. Wären dann auch noch Datenbank oder Server unzureichend gesichert, könnten die Auswirkungen weitaus schlimmer sein.

Wenn man sich dieser Nachteile aber bewusst ist und nötige Sicherheitsvorkehrungen sowie auch Tests zur Kompatibilitätssicherung trifft, steht mit PHP und Prototype ein flexibles Duo zur Verfügung um AJAX-Funktionen bis ins kleinste Detail hin zu steuern und auszureizen. Dabei hat der Entwickler eine große Community im Rücken, die schnell Hilfe zu sehr vielen Problemen und Fragestellungen verspricht.

Zudem wird PHP immer mehr professionellen Anforderungen gerecht und bietet ab der Version 5 auch umfangreiche Features der Objektorientierung, die man bisher nur von großen Programmiersprachen wie beispielsweise auch Java her kennt. Neben OO-Eigenschaften ist es aber auch in PHP möglich fortgeschrittene Entwicklungsmuster zu nutzen und es existiert auch eine Reihe Best-Practise-Lösungen. Für eine Nutzung des bei JSF unterstützten MVC-Patterns gibt es erst jetzt die ersten Ansätze, z. B. mit dem Framework Symphony[5], aber noch keine weitreichenden Erfahrungswerte. Durch die Nutzung von Template-Engines wie Smarty[6] lässt sich aber bereits die Trennung von Logik und Layout vollziehen.

Ein großer Vorteil sind unumstritten die vielen zur Verfügung stehenden Klassen und Bibliotheken, die beispielsweise von Projekten wie dem PHP Extension and Application Repository[7] (kurz: PEAR) oder der PHP Extension Community Library[8] (kurz: PECL) angeboten werden. Die XML-Antworten in der Implementierung wurden noch immer selber geschrieben, was durch den Einsatz entsprechender Klassen ebenso vereinfacht werden kann wie die Daten-

5 www.symfony-project.com/
6 http://smarty.php.net/
7 http://pear.php.net/
8 http://pecl.php.net/

bankkommunikation. Nur aus Gründen der Vergleichbarkeit wurde in der vorliegenden Arbeit weitgehend mit grundlegenden PHP-Funktionen gearbeitet.

4.3 Zusammenfassung

Wie so oft kann als abschließende Zusammenfassung keine allgemein gültige Empfehlung für oder gegen eine der vorgestellten Programmier- bzw. Skriptsprache gegeben werden. Die Ergebnisse der beiden Implementierungen weisen auf den ersten Blick wenige Unterschiede auf.

Mit JSF existiert ein guter Ansatz die Programmiersprache Java auch ins Internet zu bringen und damit zukünftig so manche Unternehmensanwendung weltweit zugänglich zu machen. Die Integration in die bisherige Software könnte dabei ebenso einen entscheidenden Vorteil darstellen wie die immer geringen Unterschiede im Verhalten einer Web-Anwendung verglichen mit einem klassischen Desktop-Programm. Allerdings sind die Anwendungsmöglichkeiten auf Programm-ähnliche Applikationen beschränkt. Grafische Spielereien wie die vielfach innerhalb sog. Web-2.0-Anwendungen Verwendung findet sind zumindest momentan und im Zusammenspiel mit Ajax4jsf nicht möglich.

Diese Studienarbeit zeigt aber auch, dass das als Skriptsprache vermeintlich nicht so professionelle PHP sich nicht zu verstecken braucht und vor allem auch durch viele Neuerungen in PHP5 im Business-Bereich immer mehr Verwendung findet. Daneben sind aber anders als noch in JSF bereits mit Prototype visuelle Effekte möglich. Auch existieren bereits weitere Frameworks, die speziell diese Aufgaben erfüllen und sich problemlos in eine PHP-Anwendung integrieren lassen.

Es bleibt abzuwarten in welche Richtung sich sowohl JSF und Ajax4jsf als auch PHP in der Version 6 entwickeln werden. Neue, lange vermisste Funktionen könnten dann den Ausschlag in eine Richtung geben. Andererseits könnte es auch darauf hinauslaufen, dass jede Technik für bestimmte Szenarien besser geeignet ist. Wem die momentanen Features von JSF und Ajax4jsf ausreichen, der kann damit schnell ansprechende Ergebnisse erzeugen, wer mehr Flexibilität wünscht oder braucht, müsste zumindest aktuell zu PHP und Prototype greifen.

Ein Vergleich bezüglich der Entwicklungszeit kann leider nicht abschließend getroffen werden, da diese aufgrund der breiten Vorkenntnisse des Autors in der PHP-Entwicklung bei der JSF-Implementierung um ein Vielfaches höher ausgefallen ist. Während in JSF aufgrund der vielen nötigen Änderungen der Konfigurationsdateien der zeitliche Aufwand der eigentlichen Programmierung höher ausfällt, hat man in PHP/Prototype die Problematik sich selbst zum Ende um die Browser-Kompatibilität kümmern zu müssen.

Egal für welche Variante man sich entscheidet und egal wie gut damit die Umsetzung einer webbasierten Anwendung gelingt, wenn der Anwender später in seinem Browser JavaScript ausschaltet, stehen beide Implementierungen dem hilflos gegenüber und werden ihrem Dienst nicht nachkommen können.

5. Quellenangabe

Bei der Erstellung dieser Studienarbeit habe ich mit folgenden folgenden Quellen gearbeitet.

Literatur

JavaServer Faces – Ein Arbeitsbuch für die Praxis, Bernd Müller, ISBN 3-446-40677-8

AJAX mit Java-Servlets und JSP, Ralph Steyer, ISBN 3-8273-2418-1

AJAX – Web 2.0 in der Praxis, Johannes Gamperl, ISBN 3-89842-764-1

Datenbank & Java, Gunter Saake/Kai-Uwe Sattler, ISBN 3-89864-228-3

iX SPECIAL 1/07: Web 2.0 – das Kompendium

Quellen im Internet

Sauber getrennt ist halb gewonnen – http://www.heise.de/ix/artikel/2006/06/156

JavaServer Faces Tutorial, Dipl.-Informatiker Markus Specker – live.interactivesystems.info/lenya/Lehre/WiSe0506/ias0405/LehreFolienInteraktiveS/TutorialJSF0405.pdf

Ajax4jsf team blog – http://www.jroller.com/page/a4j

Ajax4jsf Examples – https://ajax4jsf.dev.java.net/nonav/ajax/ajax-jsf/download.html

Ajax4jsf Developer Guide – https://ajax4jsf.dev.java.net/nonav/documentation/ajax-documentation/

JavaServer Faces bei Wikipedia – http://de.wikipedia.org/wiki/JavaServer_Faces, http://en.wikipedia.org/wiki/JavaServer_Faces