Semesterarbeit

Hochschule für Technik Zürich

Erstellung eines Prototypen zur Ferienplanung auf der Basis von Lift und Scala

Abgabe:	15. November 2010
Präsentation:	1. Dezember 2010

Student:	Raffael Schmid, rschmid@hsz-t.ch
Dozent:	Beat Seeliger, seb@panter.ch
Studiengang:	Informatik

Zusammenfassung

Ziel dieser Semesterarbeit ist es, die Möglichkeiten und das Potential der Programmiersprache Scala respektive des Webframeworks Lift zu erforschen und das notwendige Knowhow zu erarbeiten. In erster Linie wurden Funktionalitäten wie die Persistenz, Internationalisierung und Support für RESTful Webservices untersucht. Daneben ging es aber auch um die Analyse von nichtfunktionalen Eigenschaften wie Architektur, Erweiterbarkeit, Deployment und Testbarkeit.

Zum Erreichen dieses Ziels wird auf der Basis von Lift und Scala eine Webapplikation zur Ferienplanung erstellt. Diese war ursprünglich als Basis für eine Software zur Ressourcenplanung gedacht - dient aber in erster Linie vorerst als "Spielwiese" um verschiedene Anforderungen der Zielsoftware zu diskutieren.

Die resultierende Webapplikation wurde mittels dem Webframework Lift als Backend implementiert, das Frontend besteht teils aus normalen HTML Seiten, teils aus in Flex [1] implementierten Komponenten welche via REST Schnittstelle auf die Services im Hintergrund zugreiffen. Zur persistierung wurde die Java Persistence API verwendet, als Implementation war Hibernate am naheliegensten. Lift selbst und auch darauf basierende Applikationen sind in Scala umgesetzt. Die Applikation läuft Produktiv in der STAX[2] Cloud.

[1] Flex ist ein Framework von Adobe mittels welchem man mit relativ geringem Zeitaufwand Webclients erstellen kann. http://www.adobe.com/de/products/flex

[2] http://www.stax.net

Inhaltsverzeichnis

Einleitung

Während vielen Stunden habe ich mich mit Scala, dem Lift Webframework, dem Design eines Prototypen zur Ferienplanung, der entsprechenden Implementation und der Dokumentation beschäftigt. Für mich waren die Technologien relativ neu, ich kannte sie nur aus der Theorie und wie sehr oft in der Informatik, aller Anfang war schwer. Die ersten Versuche waren zwar wegbereitend, aber mühsam. Die Einarbeitung in die verschiedenen Persistenz Frameworks war sehr aufwändig und ich war unmittelbar davor, auf ein mir bekanntes Webframework zu wechseln. Ich hätte dann das Ziel verfehlt, mir am Ende ein Bild von Scala und Lift machen zu können. Die Prozesse der Applikation waren mir zu diesem Zeitpunkt schon relativ klar, trotzdem ging es jetzt an die Definition dieser, ans erstellen eines passenden Rollenkonzepts, Navigation und basierend auf diesen Vorgaben zu Design und anschliessend Implementation des Domänenmodells mit JPA. Nun war es relativ schnell möglich, Objekte in der Datenbank zu persistieren und dem lang ersehnten Flow stand nichts mehr im Wege. Der Planung lief ich zu diesem Zeitpunkt leider schon etwas hinterher, denn im Allgemeinen habe ich mir unter "fast to build"[14] einen einfacheren Einstand vorgestellt. Trotzdem, gegen Ende schwanden die Wolken am Horizont, es entstand eine gewisse Hingabe und ich überlege mir, wo ich Scala oder Lift als nächstes einsetzen könnte. Hoffentlich hilft die eine oder andere Feststellung auch weiteren Leuten oder ich kann jemanden für Scala oder Lift begeistern.

Das Dokument der Semesterarbeit ist in drei Teile gegliedert. Teil eins mit der Aufgabenstellung, Teil zwei mit deren Analyse, den zur Implementation benötigten Grundlagen, Design und Konzeption, Implementation sowie der Beschreibung von Entwicklungs- und Testumgebung. Anschliessend folgt im dritten Teil der Rückblick, hier wird die Applikation anhand der Vorgaben bewertet und ein Fazit über die verwendeten Technologien gezogen.

Danke fürs Interesse
Raffael Schmid

Teil I

Projektdetails

Kapitel 1

Aufgabenstellung

1.1 Ausgangslage

In vielen Firmen, welche ohne ERP Software auskommen, wird die Planung von Ressourcen manuell mit der Hilfe verschiedenster Standardsoftware (Excel, Access) oder Papier gemacht. Was in vielen Projekten, Unternehmen fehlt, ist die Software zur Planung von Ressourcen. Als Grundlage dazu soll ein webbasierter Ferienplaner implementiert werden, der später sukzessive zu einer Gesamtlösung erweiter werden kann. Zur Implementierung dieses Prototypen wird das Lift Webframework verwendet. Lift [1] ist ein Framework auf der Basis von Scala (eine Programmiersprache die mitunter an der Ecole Polytechnique Fdrale de Lausanne entwickelt wurde) und verinnerlicht auch deshalb in vielen Bereichen völlig neue Konzepte. Der Prototyp soll auch die Möglichkeiten der doch eher neueren Bibliothek transparent darstellen. Ich möchte darauf hinweisen, dass ich mir zusätzlich zum Prototypen das Knowhow im Bereich Scala und Lift erarbeiten muss. Die Arbeit soll es mir ermöglichen, eine Aussage über das Potential von Lift machen zu können. Scala hat, wenn man sich auf Magazine oder verschiedenster Internetseiten wie zum Beispiel den Tiobe-Index[2] bezieht, den Durchbruch ja schon fast geschafft.

1.2 Ziel der Arbeit

Als Vorarbeit zur Konzeption und Entwicklung dieses Prototypen muss das Knowhow im Bereich Lift respektive Scala erarbeitet werden. Bei diesem Prozess stehe ich zwar nicht mehr am Anfang, ich werde jedoch trotzdem zusätzlich Zeit zum Aufbau meines Wissens benötigen. Darauf aufbauend werden die Requirements der Applikation definiert und entsprechende Konzepte erstellt (User, Rollen, Prozesse). Aufgrund dieser Requirements werden Recherchen durchgeführt, um herauszufinden, ob Konzepte oder Teile bestehender Lösungen übernommen

[1] http://liftweb.net
[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

werden können. Anschliessend werden die Requirements des Prototypen umgesetzt.

1.2.1 Optionale Ziele

Ich erwarte, dass diese Zielsetzung den Rahmen einer Semesterarbeit bereits deckt, für den Fall dass ich noch Zeit finde, fasse ich optional noch folgende Punkte ins Auge:

1. Setup von Test- und Produktiver Umgebung
2. Performance Testing
3. Internationalization
4. Search Engine Optimization
5. Usability

1.3 Aufgabenstellung

Erarbeitung einer Wissensbasis im Bereich Lift und Scala, um die Konzeption und Implementation des Prototypen zu ermöglichen. Setup der Entwicklungs-Infrastruktur. Dies beinhaltet das Projektsetup mit Maven, Versionskontrolle mit Git oder Subversion, Einrichten der Entwicklungsumgebung, Aufsetzen Infrastrukur für automatisierte Testläufe, Entwicklungsserver, allerdings werde ich auf den Einsatz einer Continuous Integration Software verzichten. Während der Konzeption werden Navigations- und Rollenkonzepte erstellt sowie die verschiedenen Prozesse definiert. Implementation des Prototypen beinhaltet unter anderem Folgendes:

Aufbau des Domain Modells - Implementation der Persistenz-Schicht - Security: Registrierung, Login, ... - Navigation - Mail-Versand - Evaluierung eines geeigneten, auf Javascript basierendem Kalender, um die Persönliche Ferienübersicht sowie die Teamübersicht darstellen zu können.

1.4 Erwartetes Resultat

Dokumentation der Semesterarbeit beinhaltet unter anderem folgende Teile:

- Konzepte
  - Navigationskonzept
  - Rollenkonzept
  - Prozesse Dokumentation
- Implementationsdetails

- Entscheide Software
- Lauffähige Software als Maven-Projekt

Teil II

Umsetzung

Kapitel 2

Analyse der Aufgabenstellung

2.1 Idee und Ziele der Arbeit

Grundsätzliche Idee der vorgelegten Semesterarbeit war, anhand der Entwicklung eines Prototypen Erfahrungen mit Scala und dem Lift Framework zu machen, und um heraus zu finden wie das Potential beider Technologien ist. Zu Beginn steht natürlich die Erarbeitung des Knowhows in beiden Bereichen Scala (siehe Abschnitt 3.2 Scala) , Lift (siehe Abschnitt "3.3 Liftweb Framework" im Zentrum. In der Folge sind die aus der Aufgabenstellung entstehenden Ziele definiert.

2.1.1 Vorbereitung: Erarbeitung des Basiswissens

Nebst dem Projekt-Setup und der Einarbeitung in Scala (siehe dazu Abschnitt "3.2 Scala") besteht ein wesentlicher Bestandteil dieses Punktes auch darin, herauszufinden wie die folgenden Problemstellungen, die sich bei der Idee des Ferienplaners nicht wesentlich von anderen Geschäftsideen unterscheiden, mit Lift umsetzen lassen:

- Authentifizierung, Authorisierung
- Persistenz respektive Objekt-Relationales Mapping
- Internationalisierung
- Architektur von Applikationen
- Testbarkeit

Die Ergebnisse sind im Abschnitt "3.3 Liftweb Framework" dargestellt.

2.1.2 Design

Nachdem die Einarbeitung abgeschlossen ist, kann mit dem Design der Arbeit begonnen werden. Dieser Punkt beinhaltet die Definition der Use Cases, der Prozesse des Navigationskonzepts, des Rollenkonzepts und das Design der Datenbank.

2.1.3 Technische Umsetzung

Im Anschluss an die Design-Phase wird der Prototyp umgesetzt. Die Details zur Umsetzung befinden sich im Abschnitt "5 Implementation des Prototypen".

2.2 Lieferumfang der Semesterarbeit

Nebst dem **Prototypen** besteht die Abgabe aus **Dokumentation**, die folgende Teile beinhalten soll:

- Konzepte
  - Navigationskonzept befindet sich im Abschnitt 4.4 Navigations-Konzept.
  - Rollenkonzept befindet sich im Abschnitt "4.2 Rollen-Konzept".
  - Die Definitionen der Prozesse befinden sich im Abschnitt "4.3 Prozesse"
- Implementationsdetails befinden sich im Abschnitt "5 Implementation des Prototypen"

Die Definition des Wortes Prototyp ist bekanntlich ein bisschen schwammig. In meinem Sinne sollte der Prototyp die folgenden Punkte erfüllen:

1. **Potential** - Anhand der Implementation des Prototypen solle eine qualifizierte Aussage darüber gemacht werden, wie viel Potential in Scala und Lift steckt.
2. **Erkenntnisse** - In den tangierten Bereichen (Persistenz, Webservices, usw.) sollen Ansätze von Best Practices erarbeitet werden respektive Aussagen über die Vor- und Nachteile von verschiedenen Technologien gemacht werden können. Dies betrifft auch die Bereiche Entwicklungsumgebung, Build-Tools, usw.
3. **Abdeckung des Funktionsumfangs** - Ich werde versuchen, einen möglichst grossen Funktionsumfang der Applikation zu implementieren.

Kapitel 3

Grundlagen

Zu Beginn dieser Semesterarbeit stand die Einarbeitung in Scala und das Lift Framework auf dem Programm. Im ersten Teil dieses Abschnitts beschreibe ich grundlegende Sprachkonzepte, die zur Klassifizierung von Scala und Programmiersprachen im Allgemeinen beitragen. Es handelt sich um Punkte, denen man beim Einstieg in die Sprache notgedrungen begegnet. Gerade weil sich mit Scala deklarativ (funktional) Programmieren lässt, sind einige dieser Grundlagen fürs Verständnis sinnvoll. Der zweite Teil konzentriert sich aber auf Sprachfeatures im Detail, für Javaprogrammierer könnte das eine oder andere relativ neu sein. Der dritte Teil widmet sich voll und ganz dem Lift Framework und beschreibt, wie Anforderungen heutiger Webapplikationen umgesetzt werden können.

3.1 Begriffserklärungen zur Klassifizierung von Programmiersprachen

3.1.1 Funktionale Programmierung

Das Prinzip der Funktionalen Programmierung ist nicht nur wegen der besseren Parallelisierbarkeit bekannt, sondern weil man mit diesen Sprachen vermehrt deklarativ programmieren kann und man deshalb bei diesen von Sprachen der 5. Generation spricht. Der Vergleich zur Imperativen Programmierung wird im folgenden Abschnitt erklärt.

Imperativ vs. Deklarativ

Bei den Imperativen[1] Sprachen wird der "Computer" angewiesen, wie er ein bestimmtes Resultat berechnen muss. Im Gegenzug ermöglichen deklarative Sprachen eine Trennung zwischen Arbeits- und Steuerungsalgorithmus. Wir formulieren, was wir haben wollen, und müssen dazu nicht wissen, wie es im Hintergrund "erarbeitet" wird.

[1] der Begriff Imperativ bezeichnet die Befehlsform (lat: imperare=Befehlen)

Als gutes Beispiel für eine deklarative Sprache gilt SQL. Code in der Structured Query Language entspricht der Art unseres Denkens. Wir erfragen was wir wollen und sind am darunter liegenden Prozess nicht interessiert:

Listing 3.1: Sql Deklaration

```
1 select first_name, last_name, zip, city
2 from tbl_user
3 where zip=5430;
```

Tabelle 3.1: Resultat der deklarativen Abfrage

firstname	lastname	zip	city
Raffael	Schmid	5430	Wettingen

Eine Sql-Anweisung ist im Normalfall auch ohne detaillierte Erklärung verständlich und man hat sich nicht mit dem Steuerungsalgorithmus im Hintergrund zu beschäftigen. Da die Queries nur auf Tabellen operieren, müssen wir die Funktionsweise des Computers nicht verstehen. Mit Hilfe der Abfragesprache können wir uns auf das Wesentliche konzentrieren und mit wenigen Anweisungen viel erreichen. [18]

Im Gegensatz zu dieser Deklaration ist beispielsweise die Aufsummierung aller Zahlen einer Liste in Sprachen wie Java, C++ oder C# imperativ:

Listing 3.2: Summe einer Liste in Java

```
1 List<Integer> summanden = asList(new Integer[] { 1, 2 });
2 int summe = 0;
3 for (int i = 0; i < summanden.size(); i++) {
4         summe = summe + summanden.get(i);
5 }
6 System.out.println(summe);
```

Imperative Sprachen haben unter anderem die folgenden Eigenschaften:

- Programme bestehen aus Anweisungen, die der Prozessor in einer bestimmten Reihenfolge abarbeitet. If-Else-Anweisungen werden durch Forwärtssprünge realisiert, Schleifen durch Rückwärtssprünge.
- Werte von Variablen verändern sich unter umständen kontinuierlich.

In der deklarativen Schreibweise höherer Sprachen kann die Aufsummierung folgendermassen aussehen:

Listing 3.3: Summe einer Liste in Scala

```
1 List(1,2,3).foldLeft(0)((sum,x) => sum+x)
```

Definition Funktionale Programmierung

Funktionale Programmierung besitzt die folgenden Eigenschaften:

- jedes Programm ist auch eine Funktion
- jede Funktion kann weitere Funktionen aufrufen
- Funktionale Sprachen haben Funktionen als "Top-Class-Citizen" diese können zusätzlich zum Aufruf auch als Objekte herumgereicht werden.
- Die theoretische Grundlage von Funktionaler Programmiersprachen basiert auf dem Lambda-Kalkül[2].

3.1.2 Statisch typisierte Sprachen

Statisch typisierte Sprachen zeichnen sich dadurch aus, dass sie im Gegensatz zu dynamisch typisierten Sprachen den Typ von Variablen schon beim Kompilierungsprozess ermitteln. Der Typ kann entweder durch explizite Deklaration oder Typ Inferenz ermittelt werden.

Explizite Deklaration und Typ Inferenz

Bei der expliziten Deklaration wird der Typ einer Variablen respektive der Rückgabetyp einer Funktion durch den Programmierer festgelegt und wird für die weitere Verwendung bekannt gemacht. Im Normalfall können diese expliziten Definitionen aus den restlichen Angaben hergeleitet werden und können in höheren Sprachen wie beispielsweise Scala weggelassen werden - dann Spricht man von Typ Inferenz. Die Fähigkeiten in Sachen Typ Inferenz sind bei den Compilern der aktuellen Sprachen sehr unterschiedlich.

Typ Inferenz in Java

In diesem Bereich ist der Java Compiler nicht besonders clever. Das kleine Bisschen Typ Inferenz von Java wird am folgenden Beispiel gezeigt:

Listing 3.4: Typ Inferenz in Java

```
1 public static void main(String[] args) {
2         List<String> list = newArrayList();
3 }
4 public static <T> List<T> newArrayList() {
5         return new ArrayList<T>();
6 }
```

Die Ermittlung des Rückgabetyps aufgrund des Variablen-Typs ist alles, was Java hier bieten kann.

[2]Der Lambda-Kalkül ist eine formale Sprache zur Untersuchung von Funktionen. Sie beschreibt Funktionsdefinitionen, das Definieren formaler Parameter sowie das Auswerten und Einsetzen aktueller Parameter. Jeder Ausdruck wird dabei als auswertbare Funktion betrachtet, so dass Funktionen als Parameter übergeben werden können[25]

Vorteile von statischer Typisierung

Trotz der geringeren Flexibilität von statisch typisierten Sprachen gibt es auch ein paar nicht zu unterschätzende Vorteile:

- Typ-Fehler werden bereits bei der Kompilierung erkannt.
- Das Testen von Applikationen erhält durch den Compile-Time check eine geringere Wichtigkeit.
- Die Performance von statisch typisierten Sprachen ist besser, weil die Ermittlung des Typs zur Laufzeit in den meisten Fällen vermieden werden kann.

Nachteile von statischer Typisierung

- Kompilieraufwand ist wesentlich grösser.
- Dynamische Sprachen ermöglichen eine höhere Flexibilität. Zum Beispiel können folgende Dinge in statischen Sprachen teilweise relativ schön, aber mit erhöhtem aufwand gemacht werden:
  - Einfügen von Methoden in Classen oder Objekte zur Laufzeit in Java ist beispielsweise mit AspectJ[3] möglich, herkömmliche Mittel erlauben dies nicht.
  - Interceptoren können mittels dem seit Java 1.3 verfügbaren java.lang.reflect.Proxy implementiert werden. Dabei wird vor jeder Methodenlogik der Interceptor-Code durchlaufen. Dynamische Sprachen auf der Java-Plattform greifen auf techniken der Byte-Code-Manipulation und stellen diese Funktionalität in wesentlich einfacherer Art zur Verfügung
  - Duck Typing[4]

3.2 Scala

Scala wird von Martin Odersky seit 2003 an der EPFL in Lausanne entwickelt. Obwohl Scala an einer Hochschule entwickelt wurde, handelt es sich dabei um eine Sprache mit dem Ziel des industriellen Gebrauchs. Dies ist unter anderem auch deshalb möglich, da die Sprache auf der Java Plattform aufbaut ist und deshalb andere Bibliotheken verwendet werden können. Die Ideologie hinter Scala lässt sich durch die folgenden beiden Begriffe umschreiben:

- **Concise**[5]

[3] AspectJ ist eine aspekt-orientierte Erweiterung von Java, bei Xerox Parc entwickelt und mittlerweile Teil des Eclipse Projektes

[4] Duck-Typing ist ein Konzept der objektorientierten Programmierung, bei dem der Typ eines Objektes nicht durch seine Klasse beschrieben wird, sondern durch das Vorhandensein bestimmter Methoden. http://de.wikipedia.org/wiki/Duck-Typing

[5] übersetzt prägnant - Geschätzte 10 Mal pro Seite wird dieser Begriff in Fachliteratur zu Scala verwendet.

- **Konsistent**

Im Anschluss werden verschiedene neue Konzepte[6] gezeigt, die wesentlich für die Schönheit dieser Sprache verantwortlich sind.

3.2.1 Scala Type Inferenz

Auch wenn es sich anhand der Syntax von Scala nicht darauf schliessen lässt, bei Scala handelt es sich um eine statisch typisierte Sprache. Der Unterschied zu herkömmlichen Sprachen befindet sich in der Typinferenz - bei Scala ist die Angabe des Variablen-Typs meist optional. So handelt es sich bei den folgenden Zeilen um gültige Scala-Ausdrücke:

Listing 3.5: Typeinferenz in Scala

```
1 val name = "Rudolf"            //Variable des Typs String
2 val age = 12                   //Variable des Typs Int
3 val l = List("a","b","c")      //typisierte Liste
4
5 def add(a:Int,b:Int)=a+b       //Methode (impliziter Typ)
```

3.2.2 Traits

Traits sind ein fundamentales Konzept in Scala für die Wiederverwendbarkeit von Code. Im Gegensatz zu der Klassenvererbung können unzählige Traits[7] eingemischt werden und aufgrund der Linearisierung dieser "mixins"können Probleme wie sie bei Mehrfachvererbung vorkommen vermieden werden. Auch hier zur Erklärung wieder ein Beispiel aus [17, p. 222-227]:

Listing 3.6: Klassen und Traits definieren

```
1 abstract class IntQueue{
2    def get():Int
3    def put(x:Int)
4 }
5 class BasicIntQueue extends IntQueue{
6    private val buf = new ArrayBuffer[Int]
7    def get() = buf.remove(0)
8    def put(x:Int){buf+=x}
9 }
10 trait Doubling extends IntQueue{
11    abstract override def put(x:Int){super.put(2*x)}
12 }
13 trait Incrementing extends IntQueue{
14    abstract override def put(x:Int){super.put(x+1)}
15 }
```

[6] aus der Sicht eines Java-Entwicklers

[7] bedeutet übersetzt Eigenschaft respektive Merkmal

Hier haben wir eine abstrakte Klasse IntQueue deklariert. In der Basis-Klasse BasicIntQueue implementieren wir die beiden Methoden. Doubling und Incrementing implementieren ebenfalls die Methode get zur Verdopplung und Inkrementierung. In Java verwenden wir für das selbe Verhalten das Delegate Pattern. Der folgende Codeausschnitt verdeutlicht was passiert:

Listing 3.7: Traits: Verschiedene Instanzen vom Typ IntQueue und die entsprechenden Auswirkungen

```
1 val diQ=new BasicIntQueue with Incrementing with Doubling
2 val idQ=new BasicIntQueue with Doubling with Incrementing
3 val iQ=new BasicIntQueue with Incrementing
4 val dQ=new BasicIntQueue with Doubling
5 diQ.put(2)
6 assert(5==diQ.get)
7
8 idQ.put(2)
9 assert(6==idQ.get)
10
11 iQ.put(2)
12 assert(3==iQ.get)
13
14 dQ.put(2)
15 assert(4==dQ.get)
```

Nun erstellen wir eine Doubling-Incrementing-Queue (diQ), bei welcher die übergebene Variable zuerst verdoppelt und dann inkrementiert. Für die umgekehrte Reihenfolge die Incrementing-Doubling-Queue, sowie die anderen Increment-Queue (iQ) und Doubling-Queue(dQ).

Am Beispiel Doubling-Incrementing-Queue schauen wir uns an, was hinter den Kulissen passiert. Folgende Deklaration dient als Ausgangslage:

Listing 3.8: Traits: Deklaration Doubling-Incrementing-Queue

```
1 val diQ=new BasicIntQueue with Incrementing with Doubling
```

Die Delegation des Super-Calls wird bei dieser Deklaration von rechts nach links durchgeführt, damit Klassen nicht mehrmals aufgerufen werden führt der Scala Compiler bei der Instanzierung eine wie folgt definierte Linearisierung durch: Sofern eine Klasse in der Vererbungshierarchie mehrmals vorkommt gilt der Grundsatz, dass nur die welchem am ersten Auftritt verwendet (in Punkt 4 wird die Klasse BasicIntQueue beim ersten Mal ignoriert).

1. IntQueue - AnyRef - Any
2. BasicIntQueue - IntQueue - AnyRef - Any
3. Incrementing - BasicIntQueue - IntQueue - AnyRef - Any
4. **Doubling - ~~BasicIntQueue~~ - Incrementing - BasicIntQueue - IntQueue - AnyRef -Any**

3.2.3 Funktionen als Objekte

Scala erfüllt die wichtigsten Kriterien, die eine Sprache als Funktional bezeichnen lassen[18, p. 28]:

- Funktionen können anonym definiert werden. Das heisst, man kann Funktionen vereinbaren, ohne ihnen einen Namen zu geben.
- Funktionen werden wie alle anderen Daten behandelt. Das hat zur Folge, dass in einer statischen Sprache jede Funktion ein Typ hat.
- Funktionen sind First-Class Values und können anderen Funktionen übergeben oder als Resultate von anderer Funktionen zurückgegeben werden.
- Funktioneller Style ist unter anderem, dass Eingabewerte auf Ausgabewerte gemappt werden. Andernfalls spricht man von Methoden mit Side-Effects[8] welche Problematisch sind bei Nebenläufigkeit von Systemen.

3.2.4 Currying

Currying[9] wird in Scala mit Partieller Anwendung von Funktionen erreicht. Die Spezialisierung einer Funktion ist darauf angewiesen, dass Funktionen Konstanten zugewiesen werden können. Im folgenden Beispiel wird eine Funktion add definiert, um anschliessend die Partielle Anwendung mit der Funktion increment zu definieren.

Listing 3.9: Partielle Anwendung einer Funktion

```
1 //definition add
2 scala> def add(a:Int,b:Int) = a+b
3 add: (a: Int,b: Int)Int
4
5 //definition increment mittels Partieller Anwendung
6 scala> val increment = add(1,_:Int)
7 increment: (Int) => Int = <function1>
8
9 //Aufruf der Methode
10 scala> increment(3)
11 res4: Int = 4
```

3.2.5 Pattern Matching

Mustererkennung respektive Pattern Matching kennen die meisten von Regulären Ausdrücken, welche bestimmte Patterns in Texten erkennen können. In Scala geht die Mustererkennung wesentlich weiter. [10]

[8] Nebeneffekte

[9] Bezeichnet ein Konzept der Funktionalen Programmierung benannt nach dem Erfinder der Sprache Haskell: Haskell Brooks Curry

[10] Die Idee gibt es allerdings schon viel länger. Zum ersten Mal wurde Pattern Matching in dieser Art in ML verwendet.

Hier kann Pattern Matching auf unterschiedliche Patterns von Objekttypen angewendet werden:

- Konstante
- Platzhalter
- Tubel
- Variable
- Extraktoren
- Listen
- Typen

Mehr Informationen gibt es unter [17, p. 263-296] oder [18, p. 167-176]

3.2.6 Tail Recursion

Wie jede rekursive Funktion lassen sich Endrekursive[23] Funktion mittels einer Iteration darstellen, dabei sind die iterativen Varianten oft auch wesentlich ressourcenfreundlicher, da bei Rekursionen bei jedem Funktionsaufruf ein Frame auf dem Stack erstellt wird - und trotzdem ist die Rekursion wegen der besseren Lesbarkeit in vielen Situationen wünschenswert. Neuere Sprachen erkennen Endrekursionen und wandeln diese in Iterationen um. Sogar einzelne Java Compiler sind bereits in der Lage eine automatische Umwandlung durchzuführen. Der Scala Compiler bietet zusätzlich die Möglichkeit, mittels Annotation @endrec die Umwandlung des Compilers zu prüfen.

3.2.7 Predef

Das Predef Objekt stellt Definitionen zur Verfügung, die ohne explizite deklaration verfügbar sind und vom Compiler in die Klasse importiert werden. Es sind zum Beispiel die folgenden Punkte:

- Die Verwendung von List() liefert implizit eine Instanz vom Typ scala.collection.immutable.List. Das gleiche gilt für Set() und Map()
- println() ist implizit ein Aufruf an Console.println().

3.2.8 Implicit Conversion

Listing 3.10: Implicit Conversions am Beispiel String

```
1 scala> val s = "hello world!"
2 s: java.lang.String = hello world!
3
4 scala> println(s.reverse)
5 !dlrow olleh
```

In diesem Beispiel erstaunt, warum die Klasse java.lang.String plötzlich die Methode reverse besitzt. Unbemerkt haben wir es hier mit einer Impliziten Konversion der Klasse Predef zu tun. Deren Supertyp (LowPriorityImplicits) besitzt die Methode mit der definition:

Listing 3.11: Implicit Conversions Method wrapString

```
implicit def wrapString(s:String):WrappedString = {
   new WrappedString(s)
}
```

Sofern der Typ die aufgerufene Methode reverse nicht hat, wird im Gültigkeitsbereich nach einer Impliziten Conversion gesucht, die als Rückgabetyp ein Typ mit dieser Methode hat. Das ganze wird zur Kompilierzeit gemacht und hat entsprechend keine grossen Einfluss auf die Performance.

3.2.9 XML Datentyp

In Scala kann XML direkt in den Code eingebettet werden. Folgender Code ist deshalb völlig legitim:

Listing 3.12: XML Generierung mit Scala

```
object Test extends Application {

  val name = "planets"
  val elements = List("Merkur", "Venus", "Erde", "Mars",
                      "Jupiter", "Saturn", "Uranus",
                      "Neptun", "Pluto")
  val xml =
  <elements data={name}>
    {for (element <- elements)
        yield <element>{element}</element>
    }
  </elements>
  println(xml)
}
}
```

und produziert die Ausgabe:

Listing 3.13: Ausgabe XML Generierung mit Scala

```
<elements data="planets">
    <element>Merkur</element>
    <element>Venus</element>
    <element>Erde</element>
    <element>Mars</element>
    <element>Jupiter</element>
    <element>Saturn</element>
    <element>Uranus</element>
```

```
9       <element>Neptun</element>
10      <element>Pluto</element>
11    </elements>
```

Zusätzlich bietet Scala unzählige Möglichkeiten im Zusammenhang mit XML. Nebst der Serialisierung von Objekten können so XML-Daten deserialisiert werden und es kann Pattern Matching angewendet werden. Für weitere Informationen empfehle ich [17, p. 513 - 526]

3.2.10 Integration mit Java

Bei der Integration mit Java gibt es zwei verschiedene Möglichkeiten. Die eine, man verwendet Java Bibliotheken aus Scala Code, ist in den meisten Fällen möglich. Der umgekehrte weg ist dann schon einiges komplizierter. Aufwendig und unleserlich ist es insofern dann, wenn Methodennamen wie +, - etc. verwendet werden. Für weitere Informationen empfehle ich [17, p. 569 - 581]

3.3 Liftweb Framework

Die Evaluation eines Web Frameworks war aufgrund der Aufgabenstellung nicht nötigt. Bei der Erarbeitung der Grundlagen war vielmehr das Ziel Wege zu finden, mit denen die einzelnen Problemstellungen, die im übrigen praktisch in jeder Webapplikation auftreten, umgesetzt werden können. Im ersten Teil werde ich deshalb viele einzelne Aspekte des Lift Webframeworks beleuchten. Ich beschreibe wie man ein Projekt erstellen kann, wie das Rendering der Webseiten funktioniert, wie Formulare hergestellt werde können, Navigation und analysiere die verschiedenen möglichen Persistenz Provider.

3.3.1 Erstellen eines Lift-Projektes

Innerhalb des ganzen Lift-Ökosystems wird Maven[11] als das Build-System verwendet. Mittels der vordefinierten Maven Archetypen[12] können Projekte mit relativ geringem Aufwand erstellt werden und auch bestehende Erweiterungen können einfach durch Angabe der Abhängigkeit beansprucht werden. Momentan sind mehrere Archetypen für unterschiedliche Projekte vorhanden: zum Beispiel zur Erstellung eines Lift-Projektes basierend auf JPA (lift-archetype-jpa-basic), oder eines Lift-Projektes basierend auf Mapper[13] (lift-archetype-basic), usw. Auch wenn das Plugin Konzept verglichen mit beispielsweise Grails wesentlich weniger Leistungsfähig ist, die Maven-Konventionen [14] vereinfachen die

[11] http://maven.apache.org

[12] Archetypes in Maven sind vordefinierte Templates mit welchen Maven-Projekte erstellt werden können.

[13] Mapper ist nebst Record und der JPA-Integration eine der ORM-Libraries für Relationale Datenbanken

[14] Convention over Configuration

verschiedensten Phasen der Softwareentwicklung ungemein. Es ergeben sich dadurch viele Möglichkeiten, zum Beispiel können Maven-Projekte ohne Aufwand in Continuous Integration Systeme importiert werden.

Um mit dem Setup einer Lift-Applikation zu beginnen muss folgendes Kommando ausgeführt werden:

Listing 3.14: Erstellung eines Lift-Projektes

```
1 mvn archetype:generate
```

Anschliessend ist die Auswahl des Archetypen sowie von Gruppe, Name, Default-Package und Version notwendig. Lift-Archetypen befinden sich aktuell zwischen Position 21 und 39.

Nun lässt sich das Projekt in den "gängigen IDEs"[15] importieren. Meistens ist allerdings zusätzlich das Maven-Plugin (Eclipse) und das Scala-Plugin (IntelliJ, Eclipse, Netbeans) zu installieren. Für die Entwicklung kann es einen gewissen Vorteil bringen, wenn man SBT[16] (Simple Build Tool) verwendet. SBT ist ein Build Tool für Scala und unterstützt den Software-Entwicklungsprozess erheblich. Es stellt Funktionalitäten wie Continuous Compilation und Testing, Parallel Test Execution, usw. zur Verfügung. Die Installation ist ebenfalls relativ einfach und kann unter [15] nachgeschaut werden.

3.3.2 Bootstrapping [13, p. 26]

Das Bootstrapping der Applikation kann zusätzlich durch die Klasse Boot.scala ergänzt werden. In dieser Klassen können Dinge wie das Setup einer Navigation, die Definition der Zugriffskontrolle sowie Url-Rewriting konfiguriert werden. Die Boot.scala Datei wird bei der Projekterstellung im Package bootstrap.liftweb angelegt, was sich via web.xml anpassen lassen würde.

3.3.3 Site Rendering [13, p. 27-43]

Das Rendering einer Webseite lässt sich in verschiedene Schritte unterteilen:

1. Als erstes werden Url-Rewritings vorgenommen. Sofern eine Url nach aussen unter einem Alias verfügbar sein soll, wird dieser Alias in den Internen Pfad übersetzt.

2. Nun wird geprüft, ob es für die Url eine spezifische Dispatch-Funktion gibt. Dies kann beispielsweise dann der Fall sein, wenn ein Chart oder ein Bild generiert werden soll, im Gegensatz zu einem Template- oder View-Rendering.

3. Falls dies nicht der Fall war, wird ein passendes Template oder eine View für die vorhandene Url gesucht.

[15] Eclipse, IntelliJ und Netbean
[16] http://code.google.com/p/simple-build-tool

Rendering mit Templates

Templates sind vordefinierte XML-Dateien, welche HTML und Lift-Tags enthalten können. Anhand der aufgerufenen Url (Beispiel: /path/file) wird nacheinander versucht, die Dateien mit Name "file_de-CH" (Locale: de-CH), "file_de" (Locale: de) oder "file" mit je den Endungen .html, .htm und .xhtml aufzulösen. Diese Funktionalität kann zur Internationalisierung der Webapplikation verwendet werden. Eine datei könnte folgenden Inhalt aufweisen:

Listing 3.15: Lift Template Surround

```
<lift:surround with="default" at="content">
        <head><title>Hello!</title></head>
        <lift:Hello.world/>
</lift:surround>
```

Die Tags werden von aussen nach innen transformiert, entsprechend wird hier das Default-Template verwendet. Innerhalb dieses Templates, das ansonsten HTML-Code enthält, befindet sich der Tag zum einfügen des Seiteninhaltes.

Listing 3.16: Lift Template Binding

```
<lift:bind name="content"/>
```

Die Klasse Hello ist ein Snippte, befindet sich im Package "snippet" und hat eine oder mehrere Methoden mit dem Rückgabetyp scala.xml.Element.

Listing 3.17: Snippet

```
class Hello {
  def world = <h1>Hello World</h1>
}
```

Rendering mit Views

Anstelle von Views (die sich in Html-Datein befinden) kann wie oben beschrieben auch das Dispatching verwendet werden. Bei eingetragenen Dispatchern handelt es sich um Klassen vom Typ LiftView in welchen die Methode dispatch überschrieben wird. Hier auch wieder ein Beispiel aus [13]:

Listing 3.18: Views

```
class ExpenseView extends LiftView{
    override def dispatch = {
       case "enumerate" => doEnumerate _
    }
    def doEnumerate() :NodeSeq:{
        ...
        <lift:surround with="default" at="content">
            {expenseItems.toTable}
        </lift>
```

```
10      }
11  }
```

Views müssen sich im Package "views" befinden. Wenn man den Pfad der View auf "ExpenseView/enumerate" mapt wird sichergestellt, dass nicht alle Methoden via eine Url ansprechbar sind. Das Resultat wird nachträglich gerendert.

3.3.4 Formulare

Mit dem oben beschriebenen Mechanismus können ebenfalls Formulare definiert werden. Dabei werden in den Snippets zusätzliche Callback-Funktionen definiert, die beim Übermitteln des Post-Requests ausgeführt werden. Mehr dazu kann unter [13, p. 47-58] nachgeschlagen werden.

3.3.5 SiteMap [13, p. 61-70]

Im Lift Framework wird das Menu in form der Klasse SiteMap definiert und kann auf die Seite ebenfalls automatisch integriert werden. Des weiteren bietet aber die SiteMap noch eine Vielzahl anderer Funktionen:

- Hierarchie und Gruppierungen von Navigationselementen, damit können auch nur einzelne Äste auf der Seite angezeigt werden
- Zugriffskontrolle auf den einzelnen Elementen durch LocParams[17]
- Request-Rewriting

3.3.6 Persistenz

Relationale Datenbanken

Im Bereich der Persistenz mit relationalen Datenbanken habe ich mir die zwei neueren Framworks von Lift plus JPA angeschaut. Die Entwickler hinter dem Lift-Framework versuchten eigene OR-Mapper auf der Basis von Scala zu entwickeln:

Mapper - Das erste Framework, namentlich Mapper, ist bereits seit längerem verfügbar. Mit ihm lassen sich die gängigen Relationen (many-to-many, one-to-many) abbilden und es stellt dafür alle CRUD[18]-Operationen für Objekte bereit und sticht im Bereich des Scaffoldings[19] heraus.

Ein Beispiel für das Mapping einer User-Klasse[20] würde folgendermassen aus-

[17] LocParam ist eine Klasse, von der es verschiedene Subtypen gibt Bei der Implementation eines "net.liftweb.sitemap.Loc.If" kann man definieren, ob der Zugriff für den Benutzer auf diese Url gewährleistet wird oder nicht.

[18] Create, Read, Update, Delete

[19] Scaffolding bedeutet soviel wie die Generierung von View-Komponenten aus den in den Modellklassen existierenden Informationen.

[20] Quelle ist das Mapper Framework

sehen:

Listing 3.19: Beispiel Mapping User Klasse mit Mapper

```
trait ProtoUser[T <: ProtoUser[T]]
       extends KeyedMapper[Long, T] with UserIdAsString {
  self: T =>

  override def primaryKeyField = id

  // the primary key for the database
  object id extends MappedLongIndex(this)

  def userIdAsString: String = id.is.toString

  // First Name
  object firstName extends MappedString(this, 32) {
    override def displayName =
        fieldOwner.firstNameDisplayName
    override val fieldId = Some(Text("txtFirstName"))
  }

  def firstNameDisplayName = ??("first.name")

  //...
}
```

Die Verwendung von Typen aus dem Persistenz-Framework als Properties führt zu einer **hohen Kopplung**, insbesondere auch in Service-und Controller-Klassen.

Record - [21]Die überarbeitet Version dieser Bibliothek bietet ähnliche Funktionen an. Im wesentlichen unterscheiden sich die beiden Frameworks durch die Art- und Weise der Konfiguration.

JPA - Nebst den beiden genannten Objekt-Relationalen Mappern gibt es aber auch die Möglichkeit, die auf der Basis von scalajpa verfügbare lift-jpa Library zu verwenden. Als JPA-Implementation ist somit auch die Implementation durch Hibernate möglich. Die Mapping-Konfiguration kann mittels den bereits bekannten Java-Annotationen in den Domänenklassen gemacht werden. Nachteil ist allerdings, dass Scala Typen (Map, Set, etc.) nicht verwendet werden können.

Listing 3.20: Property Mapping mit JPA

```
@Column(name = "FIRST_NAME", nullable = false)
@NotNull
@NotEmpty
@BeanProperty
```

[21]Ich vermute, der Terminus Record stammt vom Active Record Design Pattern, das durch Martin Folwer definiert wurde.

```
5 var firstname: String = _
```

Kompliziertere Mappings werden in Java mittels Nested-Annotations[22] erstellt. Seit Scala 2.8 können diese nun folgendermassen definiert werden:

Listing 3.21: Relation Mapping mit JPA

```
1  @ManyToMany
2    @JoinTable(
3      name = "MEMBERSHIP",
4      joinColumns = Array(
5                             new JoinColumn(
6                               name = "USER_ID",
7                               referencedColumnName = "ID")
8                             ),
9      inverseJoinColumns = Array(
10                            new JoinColumn(
11                              name = "TEAM_ID",
12                              referencedColumnName = "ID")
13                            )
14                          )
15   @BeanProperty
16   var memberOf=new _root_.java.util.HashSet[Team]()
```

NoSql-Datenbanken

In den vergangenen ein bis zwei Jahren haben Objektorientierte- und Dokumentenorientierte-Datenbanken[23] stark an allgemeinem Interesse gewonnen. Man[26] begründet dies damit, dass Relationale Datenbanken mit den Eigenschaften der Zugriffen wie sie heutige Webapplikationen[24] machen an ihre Leistungsgrenzen gelangen können. Für Scala gibt es bereits eine grosse Anzahl an Adapter für diese Datenbanken und verschiedene Plattformen[25] setzen sie bereits ein:

- MongoDB[26]
- CouchDB[27]
- BigTable

3.3.7 Konfiguration

Fast alle Applikationen benötigen früher oder späger eine umgebungsabhängige Konfiguration. Bereits Maven gibt einem die Möglichkeit, verschiedene Konfigurationen Profilabhängig in den Klassenpfad zu laden. Lift Applikationen haben

[22] Annotation innerhalb einer anderen Annotation
[23] Zusammengefasst spricht man von NoSql (Not only Sql) Datenbanken
[24] hohe Anzahl an Datenänderungen bei gleichzeitig hohen Datenvolumen führt zu vielen grossen Rollback Segmenten
[25] Beispiele: Foursquare, Novell
[26] http://github.com/mongodb/mongo-java-driver
[27] http://code.google.com/p/scouchdb

zusätzlich einen Run-Mode, der sich über das System-Property “run.mode” definieren lässt. Grundsätzlich sind die Run-Modes Test, Staging, Production, Pilot, Profile möglich. Properties-Dateien im Root- oder “props”-Verzeichnis des Klassenpfades werden, sofern sie einer Namenskonvention entsprechen, nach einem bestimmten Schema geladen. Der Code befindet sich in der Klasse net.liftweb.util.Props und ist selbsterklärend:

Listing 3.22: Konfigurationsschema von .properties-Dateien im Lift Framework

```
1 lazy val toTry: List[() => String] = List(
2     () => "/props/" + _modeName + _userName + _hostName,
3       () => "/props/" + _modeName + _userName,
4       () => "/props/" + _modeName + _hostName,
5       () => "/props/" + _modeName + "default.",
6       () => "/" + _modeName + _userName + _hostName,
7       () => "/" + _modeName + _userName,
8       () => "/" + _modeName + _hostName,
9       () => "/" + _modeName + "default.")
```

Für das Beispiel Umgebung=Test, Server=Localhost, Benutzer=rschmid werden die Konfigurationsdateien in folgender Reihenfolge geladen:

1. /props/test.rschmid.localhost.properties
2. /props/test.rschmid.properties
3. /props/test.localhost.properties
4. /props/test.default.properties
5. /test.rschmid.localhost.properties
6. /test.rschmid.properties
7. /test.localhost.properties
8. /test.default.properties

Innerhalb der Applikation kann man dann mit dem folgenden Befehl eine Eigenschaft aus diesen Properties laden:

Props.get(”key”) liefert als erstes eine Box[28]

[28] Box ist ein in Lift oft verwendetes Pattern um ständigen Not-Null-Tests aus dem Weg zu gehen. Im Fall dass die Box leer ist (z.Bsp. das Property wurde nirgends konfiguriert) wird beim öffnen mittels der Methode open_! eine Exception geworfen. openOr übernimmt ein Argument und verwendet dies als Default-Wert.

3.3.8 Dependency Injection mit dem Lift Framework

Dreh- und Angelpunkt des guten Designs von Applikationen ist heutzutage oft die Art und Weise, wie Abhängigkeiten zwischen Komponenten, insbesondere Objekten, aufgelöst werden. In den letzten Jahren haben sich auch deshalb Dependency Injection (DI) Frameworks (Spring, Guice, Weld) und seit 2009 nun auch Standards (JSR 330, JSR 299) etabliert, welche zur Laufzeit die Abhängigkeiten auflösen. Die dadurch entstehenden Architekturen zeichnen sich durch eine gute Erweiterbarkeit[29] und gerade deshalb durch eine gute Testbarkeit[30] aus. Die Scala und Lift-Community beschreitet in den meisten Fällen andere, neue Wege und verwendet für solche Anforderungen das Cake-Pattern[12][16] und damit sprachinterne Mittel. Im Bereich des Lift Frameworks gibt es zusätzlich den Trait SimpleInjector um die DI-Mechanismen applikationsweit zu lösen. Dazu ein kleines Beispiel aus [2]:

Listing 3.23: Dependency Injection mit dem Lift Framework - ein Beispiel

```
1  import net.liftweb.common.Box
2  import net.liftweb.util.SimpleInjector
3
4  abstract class Thing
5  class TestThing extends Thing
6  class ProdThing extends Thing
7  object Injection extends SimpleInjector
8  trait Unboxing{
9    implicit def unboxing[T](input:Box[T]):T = input.open_!
10 }
11
12 object Test extends Application with Unboxing{
13   Injection.registerInjection[Thing]{()=>
14     new TestThing
15   }
16
17   val myThing1:Thing = Injection.inject[Thing]
18   println(myThing1.getClass.getName)
19 }
```

In diesem Beispiel gibt es zwei Implementation (TestThing und ProdThing) der Klasse Thing. Das Objekt Test ist Startpunkt der Applikation. Java-Entwickler suchen in diesem Beispiel vergeblich nach der main-Methode, weil diese sich im Trait Application befindet. Der Ablauf funktioniert folgendermassen:

1. Die main-Methode innerhalb Application instanziert das Objekt Test.
2. Der Konstruktor wird aufgerufen, Konstruktor ist in diesem Fall alles was sich zwischen den beiden geschweiften Klammern von Test befindet. Injec-

[29] die Abstraktion der verschiedenen Abhängigkeiten durch Beispielsweise Interfaces lässt einzelne Komponenten einfach austauschen

[30] Unit-Testing

tion, ein Objekt vom Typ SimpleInjector, wird konfiguriert und speichert sich die Factory-Methoden intern in einer Map.

3. Mittels Injection.inject[Thing] wird die eben konfigurierte Factory-Methode ausgeführt und das Resultat vom Typ Box[Thing] wird zurückgegeben.

4. Mittels der impliziten Konversion im Typ Unboxing wird der Typ automatisch ausgepackt und der Variablen zugewiesen.

5. Ausgabe: TestThing

Zu diesem Thema beschreibt Abschnitt "5.1.1 Dependency Injection", wie ich Dependency Injection im Prototypen eingesetzt habe.

3.3.9 Internationalisierung

Die Möglichkeiten zur Internationalisierung von Web-Applikation unterscheiden sich im Wesentlichen nicht von den Möglichkeiten anderer Applikationen und basieren ebenfalls auf java.util.Locale, die wir aus der Java-Entwicklung bereits kennen.

Ressourcen werden in sogenannten Properties Dateien (Resource Bundles) im Klassenpfad abgelegt und enthalten Key-Value-Pairs. Das jeweilige Bundle wird anhand der berechneten Locale geladen. Es gibt wie bereits unter "3.3.3 Rendering mit Templates" beschrieben die Möglichkeit, unterschiedliche Templates zu definieren, welche beim rendern der Seite Locale-abhängig ausgewählt werden.

Eine Methode mit der folgende Signatur könnte im Bootstrap konfiguriert werden, um die Berechnung der Locale wie sie per default durchgeführt wird, zu überschreiben:

Listing 3.24: Überschreibung der Locale-Berechnung

```
1 def localeCal(request : Box[HTTPRequest]): Locale = {...}
2
3 //Konfiguration im Boot.scala
4 LiftRules.localeCalculator = localeCalc _
```

Diese Anforderung würde bespielsweise bestehen, wenn man trotz der eingestellten Browsersprache initial die Deutsche Sprache laden möchte.

Kapitel 4

Design und Konzeption

Nach der Analyse der Aufgabenstellung und der Erarbeitung des Knowhows im Bereich von Scala und Lift ging es an Design und Konzeption. In dieser Phase entstanden auf der Basis der Use Cases das Konzept der Benutzerrollen, der Navigation und das Design der Prozesse. Als Basis für die Implementation diente das am Ende beschriebene Entity Relationship Model.

4.1 Use Cases Beschreibung

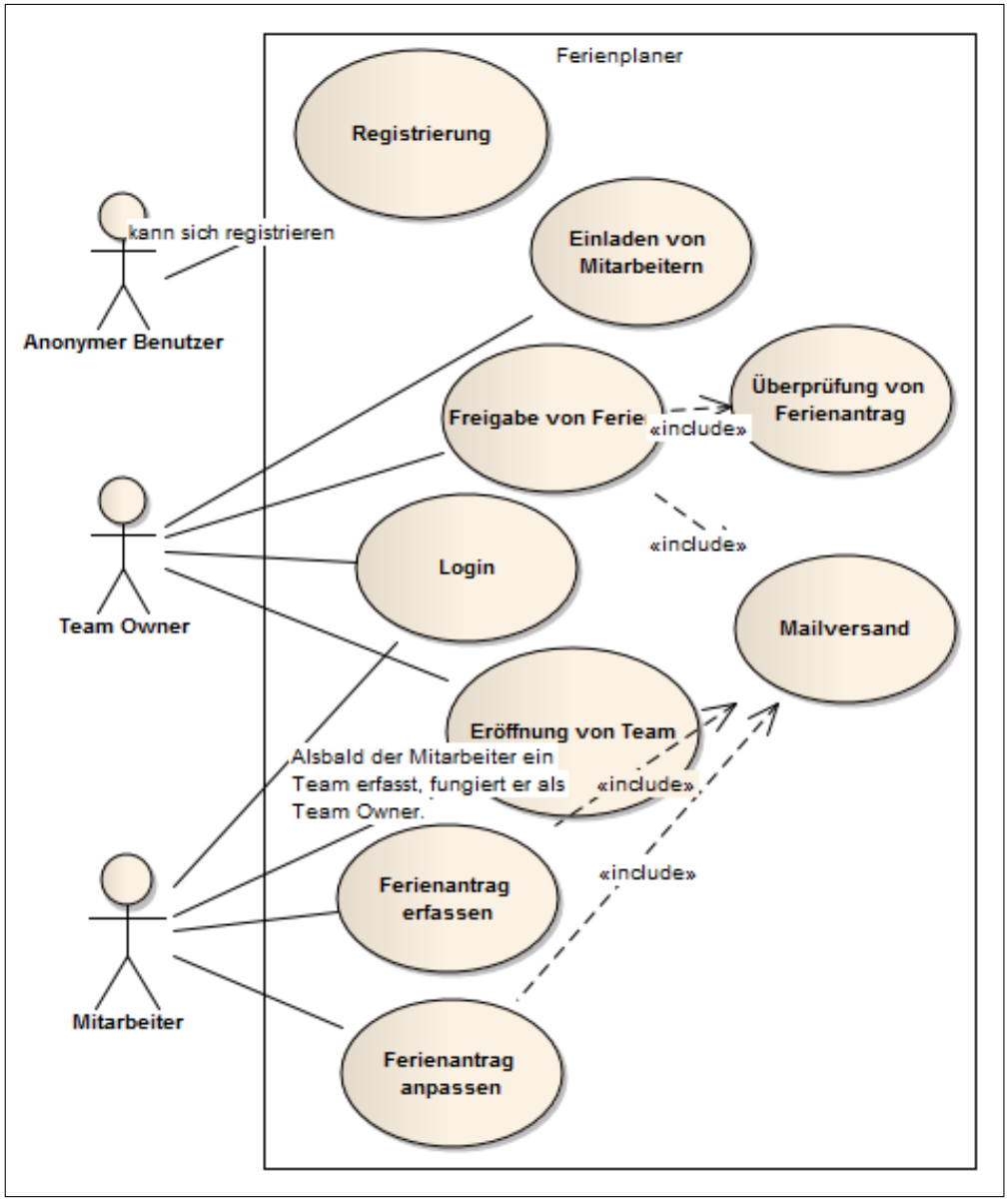

Abbildung 4.1: Use Case Diagramm

4.1.1 Aktoren

Mitarbeiter

sind registrierte Benutzer und können von Team Ownern zu den entsprechenden Teams hinzugefügt werden.

Owner - Team Owner

sind grundsätzlich auch Mitarbeiter, die allerdings ein eigenes Team administrieren und für dieses (es können auch mehrere sein) deshalb zusätzliche Kompetenzen besitzen.

4.1.2 Beschreibung der Use Cases

Projekt eröffnen

Nach dem Login erfasst der Administrator (Project Manager, Teamleiter, usw.) für sein Team ein Projekt. Grundsätzlich kann ein Projekt mehreren Administratoren besitzen. Ich werden diesen Fall aber nicht berücksichtigen und anstelle nur einem Administrator zuweisen.

Mitarbeiter in Projekt erfassen

Projekte alleine dienen der Gruppierung der Mitarbeiter, anhand dieser werden die Berechtigungen für die Administratoren gegeben. Um Mitarbeiter ins eigene Projekt zu nehmen, sucht der Administrator nach einem bestehenden User. Eine zusätzliche Möglichkeit wäre die direkte Erfassung der Mitarbeiter und entspechende Validierung durch den Betroffenen.

Ferienwünsche bearbeiten

Die vom Mitarbeiter erfassten Ferien können durch den Administrator bezüglich Status und Termin verändert werden. Die Bewilligung von Ferien wird mittels des Status confirmed / bestätigt erteilt. Ansonsten gibt es folgende Möglichkeiten:

- erwünscht - requested
- abgelehnt - rejected
- bestätigt - confirmed

Ferienwunsch erfassen

Der Mitarbeiter kann seine Ferienwünsche pro Projekt erfassen. Diese befinden sich zu Beginn im Status "requested" und können vom Administrator in die Status "rejected" und "confirmed" geändert werden. Falls der Mitarbeiter sich in mehreren Projekten befindet muss er aufgrund der Zuständigkeit für beide einen Antrag stellen.

Ferienwunsch bearbeiten

Sollte ein Ferienwunsch abgelehnt werden, kann er erneut bearbeitet werden. Jede Änderung wird allen beteiligten per Mail mitgeteilt.

4.2 Rollen-Konzept

Im Grunde handelt es sich bei dem Ferienplaner um einen Prototypen ohne eigentliche Geschäftsidee. Es war einfach eine Gute Aufgabenstellung um sich das Lift Framework und Scala anzusehen. Aus diesem Grund sehe ich als Anfang nur 2 Rollen vor. Zum einen ist es der Anonyme Benutzer, der zwar die Webseite Besuchen kann, sich für die weiteren Schritte allerdings registrieren muss, zum anderen ist es der Registrierte Benutzer, der nicht nur Ferien erfassen, sondern auch eigene Teams mit eigenen Mitarbeitern bilden kann.

4.2.1 Anonymous

Folgende Funktionalitäten stehen dem Anonymous zur Verfügung:

- Ansicht von öffentlichen Seiten
- Registrierung

4.2.2 Registrierte Benutzer

Nebst den Funktionen des Anonymous stehen dem Registrierten Benutzer folgende Dinge zur Auswahl:

- Login
- Administration von Teams und Zuweisung von Mitarbeitern
- Einladen von Mitarbeitern
- Erfassen von Ferien für Teams welchen der Registrierte Benutzer angehört.

4.2.3 Optional Aufteilung der Registrierten Benutzer

Es bestünde in einem weiteren Schritt die Möglichkeit, die Rolle des Registrierten Benutzers in Mitarbeiter und Team Owner aufzuteilen. In diesem Falle hätte man die Möglichkeit, eine Kostenpflicht zu implementieren. Als Beispiel würden die Rollen folgendermassen aussehen:

Mitarbeiter

Folgende Funktionalitäten stehen dem Mitarbeiter zur Verfügung:

- Erfassen von Ferien für Teams welchen der Registrierte Benutzer angehört.

Team Owner

Zusätzlich zu den Funktionen des Mitarbeiters kann der Team Owner folgendes tun:

- Administration von Teams und Zuweisung von Mitarbeitern
- Einladen von Mitarbeitern

4.3 Prozesse

4.3.1 Person registrieren

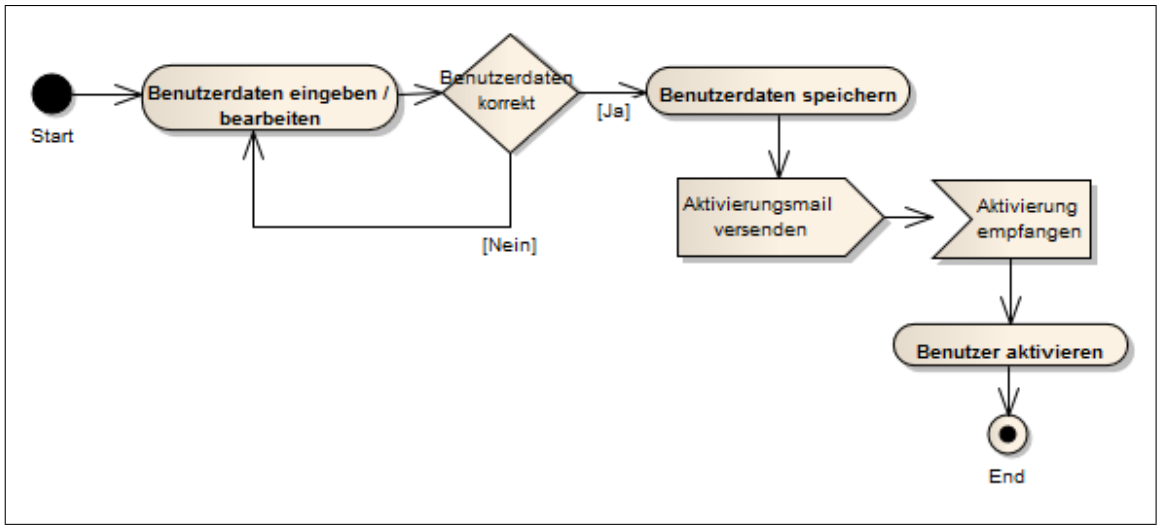

Abbildung 4.2: Prozess Member Administration Webpage

4.3.2 Ferien beantragen, planen

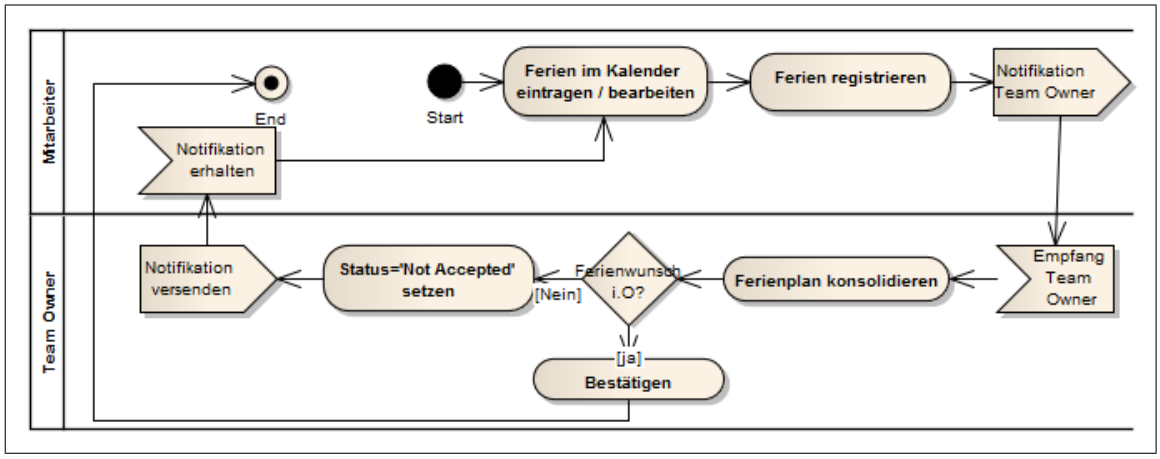

Abbildung 4.3: Prozess Ferien beantragen, planen

4.3.3 Team administrieren

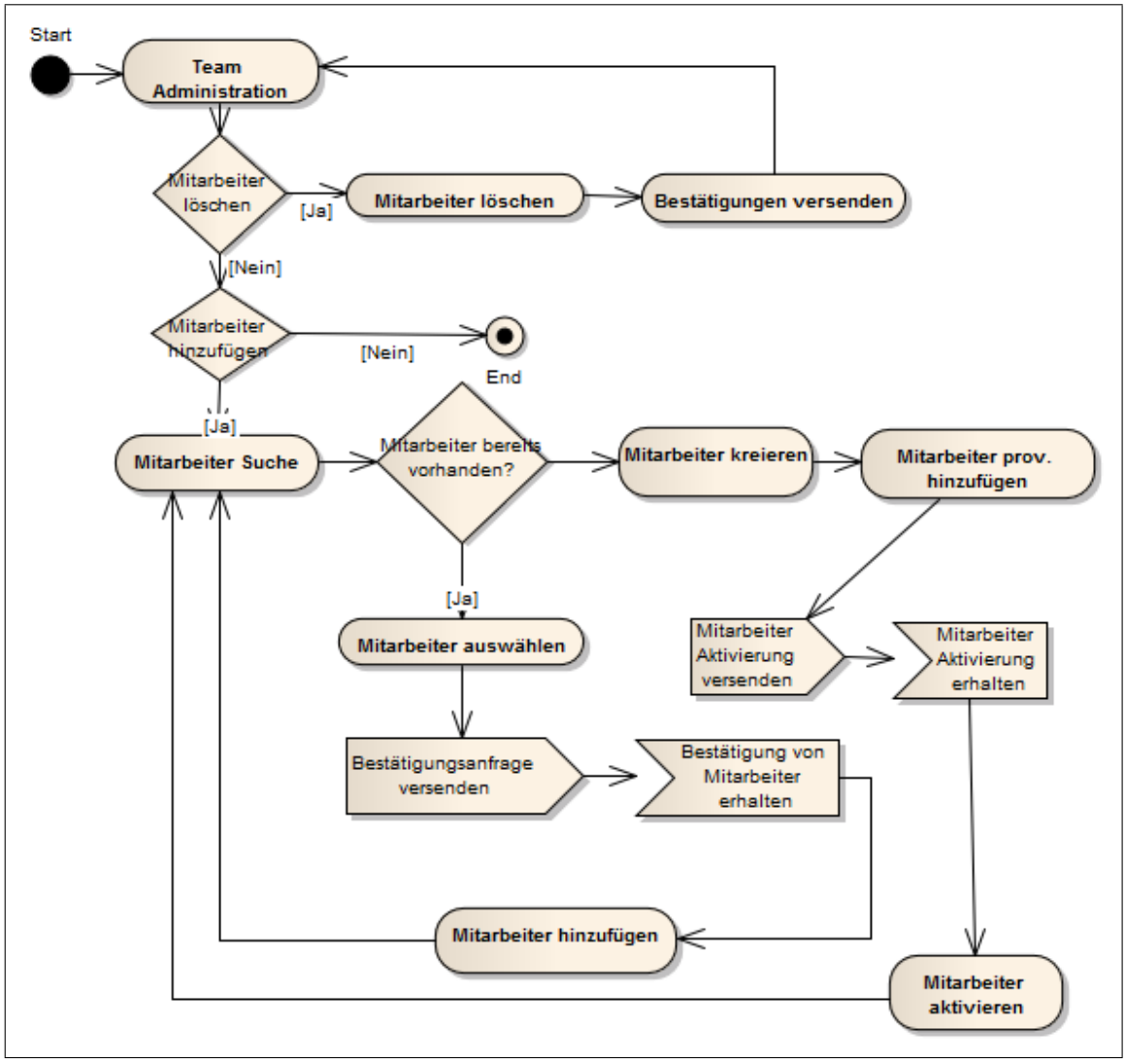

Abbildung 4.4: Prozess Member Administration Webpage

4.4 Navigations-Konzept

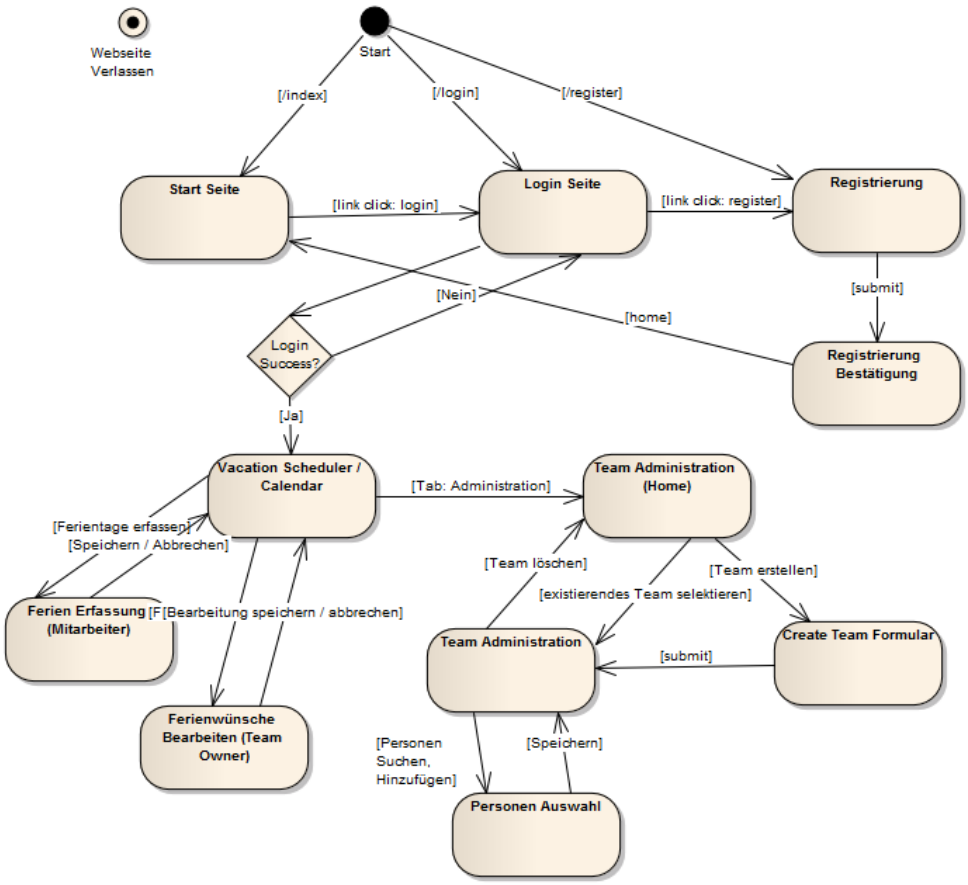

Abbildung 4.5: Navigation Webpage

4.5 Datenbank-Schema

4.5.1 Entity Relationship Model

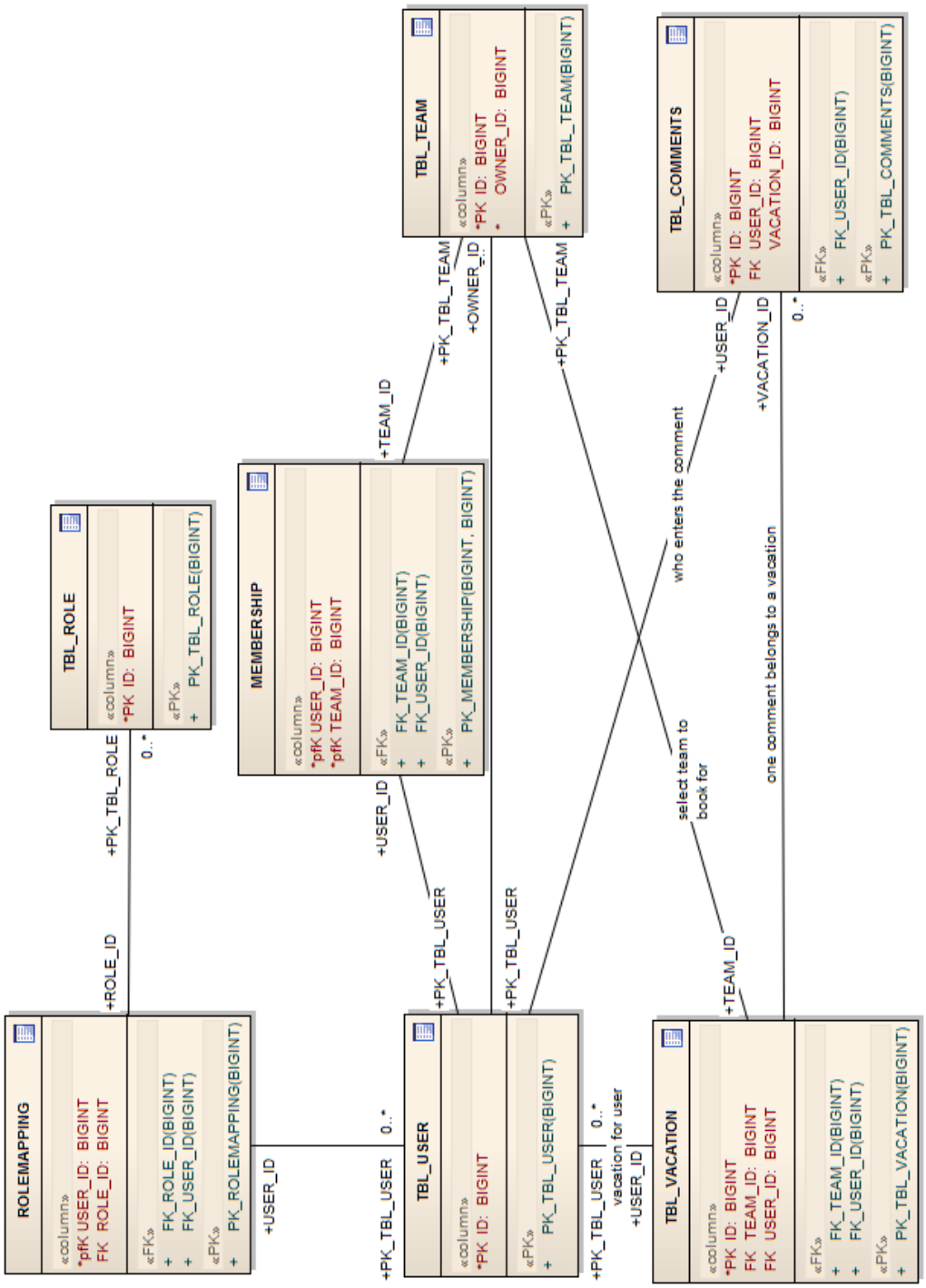

Abbildung 4.6: Entity Relationship Model

4.5.2 Beschreibung

Das Entity Relationship Model zeigt alle Tabellen, wie sie auf Ebene der Datenbank geplant sind. Many-to-Many Beziehungen sind in aufgelöster Form dargestellt.

User und Rollen

Weil Benutzer verschiedene Rollen besitzen und Rollen umgekehrt verschiedenen Benutzern zugeteilt werden können, ist die Beziehung TBL_USER-TBL_ROLE mittels einer Join-Tabelle ROLEMAPPING aufgelöst. Ich habe auf die Implementierung einer Rollenhierarchie verzichtet, trotzdem müsste man an deren Implementierung bei einer Business-Anwendung denken.

Teammitgliedschaft, Membership

Die Zuteilung von Membern zu Teams muss ebenfalls über eine Many-to-Many Assoziation realisiert werden. Im Diagramm ist diese durch die Tabelle MEMBERSHIP ersichtlich.

Teamzugehörigkeit

Im von mir modelierten einfacheren Fall, bei dem ein Team nur einem Verantwortlichen zugeteilt wird, wird die Beziehung zwischen TBL_TEAM und TBL_USER mittels einer One-To-Many Relation hergestellt. Nebst der Einfachheit hat dies den Nachteil, dass Ferien nur vom Administrator (Team-Owner) bewilligt werden können, er kann also keine Stellvertreter dafür nominieren.

Ferien

Ferien werden als Einträge in der Tabelle TBL_VACATION definiert. Die Beziehung zu TBL_USER ist insofern logisch, da sie immer einem Teilnehmer zugeordnet werden. Die Relation zu TBL_TEAM definiert, für welche Gruppe ich meine Ferien beantrage respektive von welchem Verantwortlichen die Ferien bewilligt werden können.

Kapitel 5

Implementation des Prototypen

Dieser Abschnitt beschreibt die Phase der Implementation. Zu Beginn sind es verschiedene technologische Aspekte und Themen zur Architektur: Dependency Injection, Mapping der Domäne, Validierung, etc. Anschliessend gehe ich auf die spezifischen Anforderungen der Applikation ein: Navigation, Email-Versand und die Implementation der verschiedenen Use Cases.

5.1 Architektur, Technologiewahl

5.1.1 Dependency Injection

Das Dependency Management wurde mittels dem im Abschnitt "3.3.8 Dependency Injection mit dem Lift Framework" beschriebenen Verfahren implementiert. Mittels der folgenden Bean-Definition[1] ist es möglich, je nach definiertem Environment (test, prod) die Objekte innerhalb des Bean-Contextes unterschiedlich zu definieren. Somit können beispielsweise während dem Test andere Security Implementationen verwendet werden.

Listing 5.1: Umgebungsabhängigkeit

```
1 object Beans {
2   lazy val env = System.getProperty("run.mode")
3   def init = {
4     env match {
5       case "test" =>
6         Context.put[SecurityService]{()=>
7             MockSecurityService}
8       case _ =>
9         Context.put[SecurityService]{()=>
```

[1] Die Bean-Definition befindet sich im Objekt bootstrap.config.Beans

```
10              SecurityServiceImpl}
11      }
12    }
13 }
```

Die oben beschriebenen Closures werden bei jedem Injection-Vorgang aufgerufen und liefern das entsprechende Resultat. In diesem Fall werden Scala Objekte[2] zurückgegeben. Bei Spring oder ähnlichen Frameworks spricht man vom "Singleton Scope". Mit diesem Mechanismus kann man auch Prototype-, Request- und Session-Scoped Objekte injizieren. Um die Referenz einer Variable auf ein Objekt im Context zu setzen, wird innerhalb des Prototypen die folgende Anweisung ausgeführt:

Listing 5.2: Anfrage einer Variable aus dem Lift Bean Context

```
1 val securityService = Context.inject_![SecurityService]
```

5.1.2 Persistenz

Evaluation des Persistenz Frameworks

Bereits zu Beginn stellte sich die Frage, aus welchen Gründen man welchen Persistenz-Provider verwenden soll. Siehe Abschnitt "3.3.6 Persistenz".

Ich analysierte, wie weit die drei Ansätze hinsichtlich Dokumentation, aktive Weiterentwicklung und Reifegrad der Implementation sind.

- **Aktive Weiterentwicklung:** Ein Blick auf die Code-Änderungen[3] an den Persistenz Modulen Mapper[4] und Record[5] zeigt schnell auf, dass an beiden Frameworks aktiv nicht sonderlich viel gemacht wird. Änderungen im Bereich der Persistenz-Libraries finden momentan vorallem im Bereich der NoSQL Datenbanken statt.

- **Dokumentation:** Während die Dokumentation von Mapper[6] noch einigermassen anspricht, kann man zum Mapping mit dem Record Framework ausser ganz wenigen Beispielen in [13, p. 79 - 113] zum aktuellen Zeitpunkt nichts finden.

- **Reifegrad:** Der Reifegrad der im Lift Framework enthaltenen Persistenz-Bibliotheken ist meines Erachtens gering. Anforderungen wie das Mappen von Hierarchien (Beispiele in Hibernate Table per Klasse, Table per Hierarchie) fehlen gänzlich. Die Konfigurationsmöglichkeit von Attributen und Relationen zwischen Klassen sind unflexibel und genügen höchstens bei Projekten die auf der "Grünen Wiese" beginnen. Der dritte wichtige

[2] Ein Scala-Sprachfeature zur Implementation von Singletons
[3] Commit-History
[4] http://github.com/lift/lift/commits/master/framework/lift-persistence/lift-mapper
[5] http://github.com/lift/lift/commits/master/framework/lift-persistence/lift-record
[6] http://www.assembla.com/wiki/show/liftweb/Mapper

Mangel ist, dass mit der Verwendung von Mapper und Record eine Kopplung der gesamten Applikation ans Persistenz-Framework passiert. Siehe dazu Abschnitt "3.3.6 Relationale Datenbanken".

Fazit: Meines Erachtens liefern Mapper und Record nicht das, was wir uns von bereits existierenden Frameworks wie Hibernate gewohnt sind. Ich habe mich aus oben beschriebenen Gründen dazu entschieden, JPA 2.0 und Hibernate in der Version 3.5.1 im Persistenz-Layer zu verwenden.

Domain Mapping

Ich habe die Mappings der Domänenklassen mittels im Abschnitt "3.2.10 Integration mit Java" kurz angesprochenen Annotationen gemacht. Ich gehe im folgenden auf das Mapping der User Klasse ein, werde aber an dieser Stelle auf die Beschreibung der Mappings aller Klassen verzichten.

Die Mapping Informationen für Hibernate respektive JPA befinden sich in Klasse ch.plannr.model.User. Die Klasse verwendet als Mixins folgende Traits:

- **MegaBasicUser -** stellt in Anlehnung an MegaProtoUser[7] die Funktionalitäten, die im Zusammenhang mit der Registrierung, Login, Passwort-Reset benötigt werden, zur Verfügung.
- **Domain -** definiert abstrakte Methoden zur Umwandlung von Objekten in XML und in die JSON[8]-Notation.
- **Persistent -** Stellt in Anlehnung an das Active Record Design Pattern Methoden zum Persistieren, Löschen, Editieren von Objekten zur Verfügung. Die Klasse Persistent verwendet für die beschriebenen Operationen eine ThreadLocal basierten EntityManager. Mit diesem Konstrukt wird die Thread-Safety dieses EntityManagers sichergestellt. Gleichzeitig entspricht der EntityManager dem First-Level-Cache für bereits instanzierte Objekte.

Zu persistierende Objekte werden als Entities bezeichnet und müssen in JPA mit

Listing 5.3: User: ScalaJPA Entity Definiton

```
1 @Entity
2 @Table(name = "TBL_USER")
```

annotiert werden. Mit Table kann man den Default-Tabellennamen überschreiben. Was auch ein Mapping auf Legacy Datenbanken ermöglichen würde.

Die Id des Benutzers kann bei bestimmten Datenbanken automatisch ermittelt werden. In meinem Fall MySQL wird das mittels

[7] Mega ProtoUser ist die Basis-User-Klasse für das Mapper Framework und stellt eine Basis für die Benutzerverwaltung zur Verfügung

[8] Javascript Object Notation

Listing 5.4: User: ScalaJPA Id mit Auto-Increment

```
@Id
//@GeneratedValue(
//       strategy = GenerationType.SEQUENCE,
//       generator="user_seq")
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "ID")
var id: Long = _
```

gemappt. Im Fall von Oracle müsste der GenerationType.SEQUENCE verwendet werden. Normale Properties benötigen eigentlich keine Annotation mehr, per Default werden in JPA alle Properties als Spalten angelegt, mit der @Column Annotation kann man allerdings die Defaults (Constraints, Spaltenname) überschreiben. @NotNull und @NotEmpty sind Annotationen zur Validierung von Objekten (siehe Abschnitt "5.1.2 Validierung"). Die Annotation @BeanProperty ist Bestandteil von Scala und sorgt dafür, dass für ein Feld Getter- und Setter-Methoden erstellt werden. Dies ist vorallem zur Interoperabilität mit Java-Frameworks teilweise notwendig - in diesem Fall wird es ebenfalls für die Validierung benötigt.

Listing 5.5: User: ScalaJPA firstname Mapping

```
@Column(name = "FIRST_NAME", nullable = false)
@NotNull
@NotEmpty
@BeanProperty
var firstname: String = _
```

Des weiteren sind die Annotationen @Embedded[9] und die verschiedenen Beziehungen zu anderen Tabellen @ManyToOne, @OneToMany und @ManyToMany[10] interessant.

Validierung

Annotationen wie @NotNull, @Null, @Past, @Future sind seit gut einem Jahr definiert über den Standard JSR330 (Bean Validation 1.0). Mittels Bean Validation lassen sich Validierungen auf allen Ebenen des Systems durchführen. Zum Beispiel lassen sich User-Objekte bereits ohne Speicherung validieren und entsprechend für jedes Property Fehlermeldungen in der View bereitstellen. Auf der Ebene der Datenbank können anhand dieser annotierten Properties automatisch Constraints generiert werden.

Alle Domänenklassen die als Mixin die bereits erwähnte Klasse Persistent[T] verwenden können mittels der methode validate validiert werden. Die Validie-

[9] @Embedded bietet die Möglichkeit, Attribute zwar in der selben Tabelle (embedded) zu speichern, allerdings im Objekt-Orientierten Modell in ein andere Instanz wie zum Beispiel in ein Adress-Objekt abzurufen

[10] Mittels @ManyToOne, @OneToMany, @ManyToMany können uni- und bidirektionale Beziehungen realisiert werden.

rung wird vollumfänglich anhand der in den Klassen enthaltenen Annotationen gemacht. Als Rückgabewert werden die Constraints-Verletzungen als Set zurückgeliefert. Ich verwende diese Constraints-Verletzungen weiter für die Anzeige in der View. Zum Beispiel der Validierung bei der Registrierung werden bei unvollständiger Eingabe folgende Fehler angezeigt:

Abbildung 5.1: Formular Validierung: Registration

Die Implementation der Methode validate basiert wie bereits gesagt auf dem Validation Framework und sieht folgendermassen aus:

Listing 5.6: Validation innerhalb der Klasse Persistence

```
1  @Transient
2  private val validatorFactory =
3                  Validation.buildDefaultValidatorFactory();
4
5  @Transient
6  private val validator = validatorFactory.getValidator();
7
8  def validate() = {
9    validator.validate(this)
10 }
```

5.1.3 Verwendung und Aufbau des Flex Clients

Für die Administration der Teams sowie auch für die Ferienplanung habe ich mich wegen der besseren Benutzungsfreundlichekeit und dem besseren Software-Design im Frontend für Adobe Flex entschieden. Ich werde im nächsten Abschnitt aufzeigen, wie die Architektur von Flex Clients mit dem Einsatz eines Flex-Frameworks umgesetzt werden kann.

Architektur

Die meisten User-Interfaces sind wegen der hohen Kopplung zwischen einzelnen Komponenten sehr schlecht wartbar. Dies betrifft nebst Swing- und Flex-Clients auch viele Frontends in Javascript und HTML. Zu dieser Kopplung gehört auch

das Problem, dass Observer für Events nicht via Konfiguration sondern programmatisch registriert werden. Des weiteren gibt es generelle Anforderungen an Zentrale Daten (Contexte) und die Anbindung an Backend-Systeme. Damit solche Problemstellungen und die im Zusammenhang damit auftretenden Fragen für diesen Client minimiert werden können, habe ich eine kurze Evaluation bestehender Flex Dependency Injection Frameworks gemacht. Bei den meisten geht allerdings der Funktionsumfang weit über DI hinaus.

- **Cairngorm[1] -** unter Flex 2 und 3 war Cairngorm noch ein Framework mit Fokus auf die MVC-Architektur innerhalb Flex. Mittlerweile handelt es sich allerdings um ein breiteres Set von Guidelines, Tools und Bibliotheken. Welche sich frameworkunabhängig einsetzen lassen. Mittlerweile basieren viele dieser Bibliotheken auf anderen DI- und MVC-Frameworks wie Parsley, Swiz, oder Spring ActionScript

- **Mate[7] -** in erster Linie wurde Mate für das Event-Handling entwickelt. Die Art und Weise der Konfiguration findet deklarativ via ein MXML statt und ist relativ unflexibel:

Listing 5.7: Event-Deklaration mit dem Mate Framework

```
<EventHandlers type="{MessageEvent.GET}">
  <RemoteObjectInvoker
    instance="{services.helloService}"
    method="sayHello"
    arguments="{event.name}">
    <resultHandlers>
      <CallBack
        method="handleResult"
        arguments="{resultObject.text}"/>
    </resultHandlers>
    <faultHandlers>
      <CallBack
        method="handleFault"
        arguments="{fault.faultDetail}"/>
    </faultHandlers>
    </RemoteObjectInvoker>
</EventHandlers>
```

- **Switz[11] -** ist meines Erachtens das flexibelste und am elegantesten zu konfigurierende Framework. Es stellt folgende drei Hauptfunktionalitäten zur Verfügung:

  – **Dependency Injection** als Basis zur Inversion of Control anhand eines Beispieles. Beans und somit der applikationsweite Context werden in einer MXML-Datei definiert:

Listing 5.8: Swiz: Bean Deklaration

```
<swiz:BeanProvider ...>
  <fx:Declarations>
    <core:Context id="context"/>
  </fx:Declarations>
</swiz:BeanProvider>
```

Objekte können nun in beliebeigen MXML-Dateien oder ActionScript Klassen injected werden - selbst two-way Bindings sind möglich:

Listing 5.9: Swiz: Bean Injection

```
[Inject(source="context.selectedTeam",
        bind="true",
        twoWay="true")]
public var selectedTeam:Team = null;
```

- **Event Handling** zur entkopplung von Mediator und Observer. Events werden mit der Methode dispatchEvent(Event e) des Interfaces IEventDispatcher geworfen werden. MXML sind grundsätzlich von diesem Typ, in normalen ActionScript-Klassen kann ein entsprechender Dispatcher injected werden. Um sich für Events zu registrieren ist eine Methode mit folgender Signatur und Annotation zu implementieren:

  Listing 5.10: Swiz: Event Observer

  ```
  [Mediate( event="Events.SEARCH_USERS",
            properties="term" )]
  public function searchUsers( term:String) : void
  {
  //...
  }
  ```

  Hier wird das Property term des geworfenen Events zusätzlich der Methode als Argument übergeben.
- Einfacher Lifecycle für asynchrone Aufrufe auf Remote Systeme.

Aufgrund der Einfachheit des Swiz-Frameworks, der Flexibilität und der eleganten Konfiguration habe ich mich gegen Cairngorm und Mate entschieden und das Event-Handling sowie Context und Dependency Injection mit Swiz implementiert.

Kommunikation mit dem Backend

Flex-Applikationen werden als Flash-Dateien dem Browser gesendet und laufen innerhalb der Flash Runtime Umgebung. Verglichen mit HTML 4.0.1 und

XHTML 1.0 hat man mit Flex die Möglichkeit, Daten respektive Zustände clientseitig zu Speichern - mindestens so lange die Applikation nicht neu geladen wird. In meinem Fall wird der Flex-Client innerhalb der Lift-Applikation eingebettet und kommuniziert mit dem Backend via **RESTful Webservices**. REST ist ein Architekturstil für Hypermedien wie das World Wide Web und legt nahe, jede Ressource mit einer eigenen URI anzusprechen. Operationen auf Ressourcen werden mittels HTTP Methoden wie PUT, DELETE, POST,GET, HEAD, etc. manipuliert[27]. Innerhalb der Browser-Umgebung ist die Verwendung von POST und GET möglich, die anderen Methoden stehen allerdings nicht in allen Browsern zur Verfügung. Zu diesem Zweck werden bei meinem Prototypen PUT- und DELETE-Requests mit dem HTTP-Header "X-HTTP-Method-Override" ausgestattet und als eigentliche POST-Requests gesendet. Die Applikation merkt serverseitig von diesem Header nichts, da er bereits in der Filter Chain zuvor die richtige Methode auf dem Request setzt. Für weitere Informationen siehe:

- ch.plannr.common.http.WrappedRequest.java (server)
- Klasse ch.plannr.common.http.RequestWrapperFilter.scala (server)
- web.xml (server)
- ch.plannr.service.HttpServiceFactory (client)

REST definiert grundsätzlich nicht, in welchem Format daten übertragen werden. Mögliche Formate wären JSON und XML. Es war naheliegend, aufgrund der XML-Unterstützung in Scala, dieses Format zu nehmen. Für eine weitere Applikation würde ich den Einsatz von BlazeDS[6][28] oder GranitDS[5] prüfen.

Zu Beginn habe ich die **Authentifizierung** der Webservice-Aufrufe über Basic Authentication gemacht. Dies hätte in der Praxis den wesentlichen Nachteil, dass prinzipiell alle Aufrufe verschlüsselt werden müssen[11]. Nun habe ich die Authentifizierung auf eine Form-Authentication umgestellt, bei der die Benutzer-Identifikation respektive in meinem Fall die Email-Adresse in der Session gespeichert wird. Somit muss nur der Login-Formular Submit via das Https-Protokoll laufen.

5.1.4 Navigation

Da die Ferienplanung und Administration der Benutzer vollumfänglich in Flex implementiert ist, basiert auch die Navigation dieser Beiden Use Cases auf Flex Komponenten wie Tab-Panels sowie States und Transitions (siehe: "5.2.2 Ferienplanung und Teammanagement"). Ausserhalb ist die Navigation mittels der Lift-Sitemap implementiert. Wie bereits unter "3.3.5 SiteMap [13, p. 61-70]" angekündigt bezieht sich die Lift-Sitemap nicht nur auf das Verlinken von Seiten, sondern auch für die Zugriffskontrolle. Die Struktur des Menus sieht folgendermassen aus:

[11] Bei Basic Authentication wird Benutzernamen und Passwort in der Form "username:password" im Base-64 Format übermittelt

Tabelle 5.1: Navigation und Menustruktur

Name	Link	Constraints
Home	/index	keine
Manager	/manager	Benutzer angemeldet
Login	/user_mgt/login	Benutzer nicht angemeldet
Benutzer registrieren	/user_mgt/sign_up	Benutzer nicht angemeldet
Passwort verloren	/user_mgt/lost_password	Benutzer nicht angemeldet
Logout	/user_mgt/logout	Benutzer angemeldet
Passworter reset	/user_mgt/reset_password	Benutzer angemeldet
Benutzer editieren	/user_mgt/edit	Benutzer angemeldet
Passwort ändern	/user_mgt/change_password	Benutzer angemeldet

Das Menu wird aus einer Liste von Menu-Einträgen in der Klasse Boot.scala definiert und in die SiteMap gesetzt. Die Zugriffssteuerung wird über die LocParams implementiert gewährleistet.

5.2 Details zur Implementation der Use Cases

5.2.1 Benutzermanagement

Sofern man bei der Entwicklung von Webapplikationen mittels Lift den Persistenz-Provider Mapper auswählt, erhält man Benutzerverwaltung inklusive Login-, Registrierung- und Passwort-Reset-Mechanismus mitgeliefert und kann diesen relativ flexibel erweitern. Da ich mich allerdings aus den bereits beschriebenen Gründen für JPA und gegen Mapper entschieden habe, musste ich diese Funktionalität selber implementieren. Dies habe ich auf der Basis des ProtoUsers, MegaProtoUsers und MetaMegaProtoUser vom Mapper-Framework getan. Zur Beschreibung wie die Benutzerdaten zwischen Formularen und dem Backend ausgetauscht werden, eine kurze Erläuterung wie die Funktionalität Passwort-Reset funktioniert:

Listing 5.11: Implementation Funktionalität Passwort-Reset

```
1  def passwordReset(id: String) =
2      findById(id.toLong) match {
3        case Full(user) =>
4          def finishSet() {
5            val violations = user.validate
6            if (violations.size == 0)
7              user.merge
8            else {
9              val node: NodeSeq = violations
10             S.error(node)
11             S.redirectTo("/")
12           }
```

```
13          }
14
15          bind("user", HtmlTemplate.passwordResetXhtml,
16            "pwd" ->
17              password_*("", (p: List[String]) =>
18                user.password = p.head),
19            "submit" ->
20              submit(S.??("set.password"), finishSet _))
21        case _ =>
22          S.error(S.??("password.link.invalid"));
23          S.redirectTo(homePage)
24      }
```

Die Methode zum Reset des Passworts wird vom Menu respektive der Lift-Internen SiteMap Funktionalität (siehe unter: "3.3.5 SiteMap [13, p. 61-70]") aufgerufen. Möglich ist dies nur, wenn der Benutzer angemeldet ist.. Der Rückgabewert der passwordReset-Methode ist ein Objekt vom Typ scala.xml.NodeSeq, welches das eigentliche Formular repräsentiert. Eingabefelder werden nicht in reinem HTML definiert, sondern via die Verwendung der Methoden text, password, usw. Diese Methoden der Klasse SHtml registrieren zum einen eine Callback-Funktion, zum anderen geben sie das auszugebenden Xml-Element als scala.xml.Elem zurück. Beim Übermitteln des Formulars wird dann die entsprechend zu einer definierten ID registrierte Callback-Funktion mit dem vom Benutzer eingegebenen Wert aufgerufen. Zum einen wird somit für die Textfelder der Wert hinterlegt, zum anderen via die Methode finishSet der Benutzer validiert und gespeichert.

Alle Formulare der Benutzerverwaltung sind auf diese Weise implementiert.

5.2.2 Ferienplanung und Teammanagement

Die Prozesse der Teamadministration und Ferienplanung sind unter "4.3.2 Ferien beantragen, planen" und "4.3.3 Team administrieren" beschrieben. Aufgrund der folgenden drei Gründe habe ich mich entschieden, beide Teile in Flex zu implementieren:

- **Usability:** Usability bezeichnet die vom Nutzer erlebte Nutzungsqualität (respektive User Experience) bei der Interaktion mit einem System[22]. Heutzutage erreicht man dies mit der Verwendung von Technologien wie Ajax, Flex, Silverlight, oder andere. Der Benutzer erfährt die Webseite wie als eine auf seinem Rechner installierte Applikation:
  - kurze Ladezeiten
  - Interaktion mit dem Server findet statt, indem nur Teile der Seite ausgetauscht werden.
- **Aufwand:** Der Aufwand um Administrationsoberflächen zu implementieren ist vergleichsweise geringer. Man muss allerdings auch berücksichtigen,

dass die Kommunikation mit dem Backend respektive das Testen der Schnittstelle zum Backend trotzdem eine Menge Zeit in Anspruch nimmt.

- **Kalender:** Die Verwendung des Flex Kalenders "IBM Eloxier" erschien mir als angemessen (Siehe Abschnitt "5.2.2 Evaluation Kalender")

Evaluation Kalender

Unter dem Namen "Elixier" verkauft IBM Flex Komponenten. Da ich bereits im Besitz der Standard-Lizenz war, habe ich mich für diesen Kalender entschieden. Die Enterprise Version beinhaltet zusätzlich noch ein Gantt-Chart [24], der für private doch relativ hohe Preis (dieser liegt momentan bei 3130 USD) hat mich allerdings von der Verwendung dieser Komponente gehindert. Andere Widgets habe ich aufgrund meiner Meinung nach mangelnder Qualität, Stabilität oder Funktionalität nicht verwendet:

- Simile Timeline[8]
- Flex Calendar Components [3]

Der Kalender sieht auf dem Prototypen folgendermassen aus:

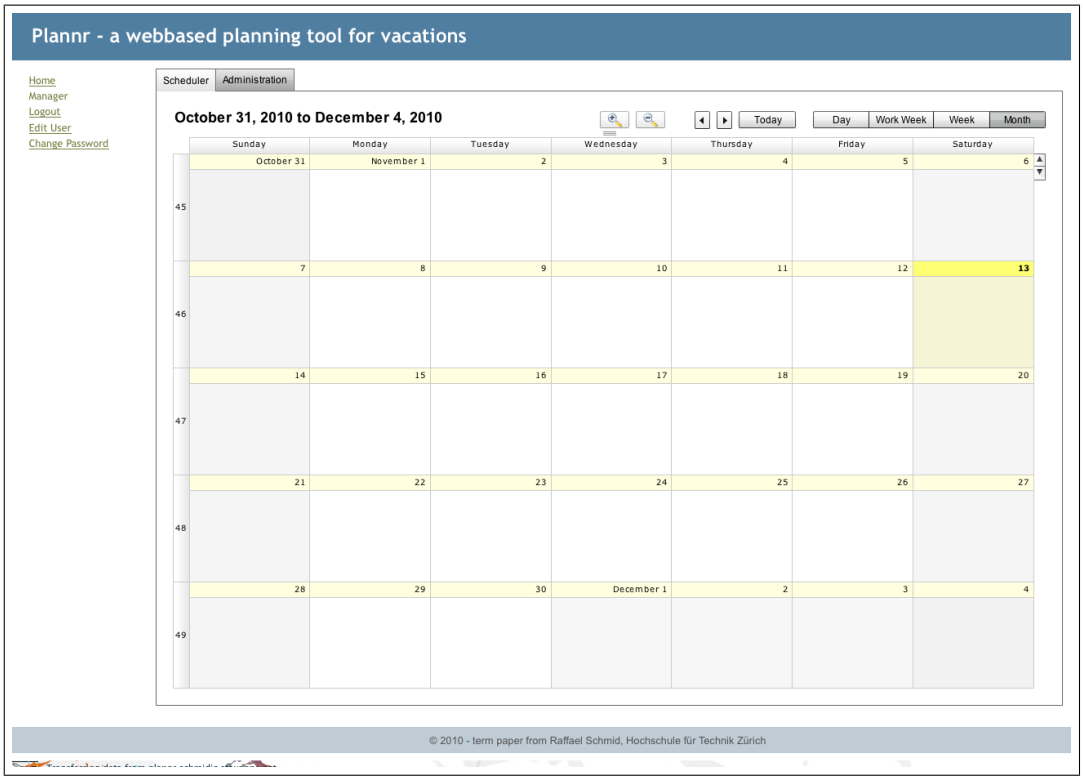

Abbildung 5.2: Ansicht des verwendeten Kalenders

5.2.3 Email-Versand (Notifikationen)

Use Cases

Zu unterschiedlichen Zwecken müssen Emails an die Benutzer versendet werden. Die Email-Adressen gelten systemweit als Benutzer Identifikatoren und sind deshalb sowieso bereits beim Benutzer gespeichert. Nachfolgend eine Liste der Use Cases mit Email und der Stand der Implementation im Prototypen:

Tabelle 5.2: Use Cases Email Versand

Aktion	**Beschreibung**	**Status Mail Versand**
Signup	Nach der Registrierung wird zur Email-Validierung ein Email inklusive einem Link an den Benutzer versendet, über welchen er den Account aktivieren kann. Im Link enthalten ist eine Zufallszahl so dass kein Missbrauch betrieben werden kann.	Nicht implementiert
Passwort vergessen	Sofern das Passwort vergessen geht, wird ein Email an den Benutzer versendet. Über dieses Email kann das Passwort neu gesetzt werden.	Implementiert
Erfassung von Ferien durch den Mitarbeiter	Nachdem der Benutzer seine Ferienwünsche erfasst hat, geht ein Email an den Vorgesetzten, damit dieser die Einträge überprüfen kann	Nicht implementiert
Statusänderung durch Vorgesetzten	Nachdem der Vorgesetzte die Ferien überprüft und bewilligt, abgelehnt hat, wird der Mitarbeiter informiert	Nicht implementiert

Technische Umsetzung

In meinem Fall lässt sich der Lift-Mail-Support, der sich innerhalb der Klasse net.liftweb.util.Mailer befindet, nicht nur über die definition der Properties (mail.smtp.host, mail.username, mail.password) konfigurieren - ich muss zusätzlich den Authenticator (Mailer.authenticator) des Objekts Mailer setzen. Deshalb wird der Mailer beim Startup entsprechend konfiguriert.

Mails können nun mit dem Aufruf der folgenden Methode versendet werden:

```
1     Mailer.sendMail(From(emailFrom),
2                     Subject(signupMailSubject),
3                     (To(user.email) ::
4                     xmlToMailBodyType(msgXml) ::
5                     (bccEmail.toList.map(BCC(_)))): _*)
```

Das erste Argument ist die Absender-Adresse, das zweite das Subject und das dritte eine Liste bestehend aus Empfänger, Carbon Copy (CC), Blind Carbon Copy (BCC) und Mailbodies (HTML, Plain Text). Die Liste wird vom

Lift Framework beim Verwenden wieder in ihre Bestandteile aufgelöst und mit Empfänger und Subject versendet.

Kapitel 6

Entwicklung, Build und Deployment

6.1 Build-System

Maven als Build-System basiert ebenfalls auf einem Plugin-Konzept und unterstützt die Entwicklung in den verschiedensten Bereichen des Build-Managements. Nicht zuletzt dank dem Grundsatz "Convention over Configuration" lässt es sich auf allen Plattformen sehr schnell deployen. Maven übernimmt bei diesem Projekt eine vielzahl an Funktionen:

- **Dependency Management:** Maven lädt die im pom.xml definierten Abhängigkeiten von öffentlichen Repositories.
- **Support Entwicklungsumgebung:** Die grosse Verbreitung von Maven hat dazu geführt, dass für Entwicklungsumgebungen Adapter/Plugins entwickelt wurden, womit sich Projekte ohne weiteres importieren lassen. Beim import werden Abhängigkeiten in den Source- und Build-Pfad gesetzt, die richtige Runtime gesetzt, Ordner für kompilierte Klassen gesetzt, etc.
- **Clean, Compile, Package:** Der gesamte Build-Prozess fürs Produktiv-System wird mittels Maven gemacht, innerhalb der Entwicklung wird dies allerdings nicht benötigt - das übernimmt die IDE.
- **Profil Management -** Maven ist von Grund auf fähig, verschiedene Profile zu definieren. Profile kann man beispielsweise dann verwenden, wenn Applikationen für Entwicklung, Test, Produktion in unterschiedlichen Umgebungen laufen. Dann gilt es meistens, unterschiedliche Konfigurationen zu verwenden. Bei mir sind folgende Dinge Profil-Abhängig:
  - Der Datenbank-Zugriff wird in der Entwicklungsumgebung direkt über JDBC gemacht, in der Testumgebung auf der Stax-Cloud wird eine vom Servern bereitgestellte Datasource verwendet.

6.2 Entwicklungsumgebung

Während der Entwicklung hilft Maven vorallem im Bereich des Dependency Managements - Abhängigkeiten werden automatisch geladen, das Projekt kann nach der Öffnung sofort kompiliert und gestartet werden. Ein paar Funktionalitäten im Backend habe ich Test-Driven mit Scala-Specs entwickelt, ohne dass ich die Funktionalität jedesmal im Browser nachvollzog. Tests kann man auf verschiedene Arten ausführen:

- Maven per Default unterstützt die Ausführung von JUnit-Tests, mit Plugins können allerdings auch Scala-Specs oder andere Tests ausführen.
- Entwicklungsumgebungen welche Scala unterstützen, bringen meistens auch Funktionalität zum Testen (Ausführung von Scala Specs oder Scala Unit Tests).
- Die Scala-Gemeinde richtet sich mehr und mehr auf das Simple Build Tool[9] aus. Es handelt sich dabei um ein Tool welches ebenfalls Dependency Management, Test, Build, usw. unterstützt und sich gegebenenfalls auch mit Maven kombinieren lässt. Bei der Entwicklung hat das Tool mir geholfen, die Tests[1] auszuführen, indem es kontinuierlich auf Änderungen wartet, neu Kompiliert und entsprechend die Tests neu ausführt. Das somit schnelle Feedback kann den Entwicklungsprozess massiv beschleunigen.

6.3 Test- und Produktiv-Umgebung

Um die Applikation auch der Breiten Masse verfügbar zu machen, habe ich sie auf der Cloud-Plattform namens Stax[10] deployt. Bei der Verfolgung von Twitter-Nachrichten verschiedener Personen und Diensten bin ich auf diese Plattform aufmerksam geworden und die Einfachheit des Deployments hat mich sehr überrascht. Die Stax Cloud Plattform[10] bietet die Möglichkeit, Applikationen, die bevorzugt Maven und Ant als Build-Instrumente verwenden, auf die Amazon EC2 Plattform zu deployen. Die Verwendung der Stax Test Cloud ist gratis und kann zu einem beliebigen Zeitpunkt auf die Premium Java Cloude aufgerüstet werden. Bei der Premium Cloud bezahlt man anhand benutzer Bandbreite und grösse der CPU Instanz.

Für das Deployment auf diese Plattform musste ich folgende Änderungen durchführen:

- **Stax-Konfiguration für Maven -** Für Maven ist ein Stax-Plugin vorhanden, das mittels der folgenden Zeilen im pom.xml konfiguriert werden muss:

Listing 6.1: Konfiguration Maven mit Stax

[1] Continuous Testing

```
<build>
  <plugins>
    <plugin>
      <groupId>net.stax</groupId>
      <artifactId>stax-maven-plugin</artifactId>
    </plugin>
  ...
  </plugins>
</build>
```

Schlussendlich ist es möglich, die Applikation mittels dem folgenden Kommando zu deployen:

Listing 6.2: Befehl Deployment Stax

```
mvn clean stax:deploy -Pprod -Dstax.appid=plannr
```

- **Stax-Konfiguration des Web Archives -** Wars für die Stax Plattform müssen ein zusätzliches Xml mit dem namen stax-web.xml beinhalten.
- **Datasource Anbindung -** Via das Web-Interface von Stax kann eine neue MySql Datenbank auf der Cloud angelegt werden. Um der Applikation den Zugriff auf diese Datenbank zu ermöglichen, wird diese einerseits im stax-web.xml als auch im web.xml definiert. Die Auflösung der Datasource wird zur Laufzeit über den JNDI-Namen gemacht, der im persistence.xml definiert ist.
- **Konfiguration JPA -** Damit sich der Persistenz-Provider nicht wie während der Entwicklungszeit direkt über JDBC mit Url, Benutzernamen und Passwort verbindet, wird im Profilabhängigen persistence.xml für Test- und Produktiv-Umgebung die im web.xml als Ressource definierte Datasource verwendet.

Der Pfad für diese Dateien werden nur mit der Aktivierung über das Maven-Profil geladen und überschreiben die bereits bestehenden.

6.4 Source Code Management

Zur Versionskontrolle während der Zeit dieser Semesterarbeit habe ich Git [4] verwendet. Die Sourcen befinden sich unter [19].

Teil III

Rückblick

Kapitel 7

Analyse der Arbeit auf der Basis der Aufgabenstellung

7.1 Zielerreichung Prototyp

7.1.1 Funktionsumfang für anonyme Benutzer

Auf den folgenden drei Bildern sind die öffentlichen Seiten des Prototypen abgebildet. Es handelt sich also um Funktionen die allen Benutzern zur Verfügung stehen, sofern sie nicht angemeldet sind. Die Funktionalitäten sind grundsätzlich fertig implementiert, bräuchten aber für eine produktive Anwendung noch einige Anpassungen:

- Mit dem Gedanken im Hinterkopf, dass es sich um einen Prototypen handelt, habe ich den Fokus weg vom grafischen Design genommen und mich auf den Funktionsumfang konzentriert. Hier wäre es für eine produktive Applikation wichtig, noch einige Zeit zu investieren.
- Der Code der Klasse BasicUser müsste überarbeitet werden.
- Für eine mehrsprachige Anwendung müsste man die bestehenden Texte extrahieren und durch mehrsprachige Properties-Dateien ersetzen.

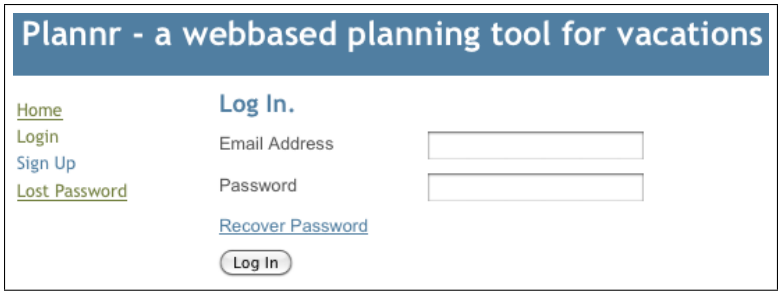

Abbildung 7.1: Login

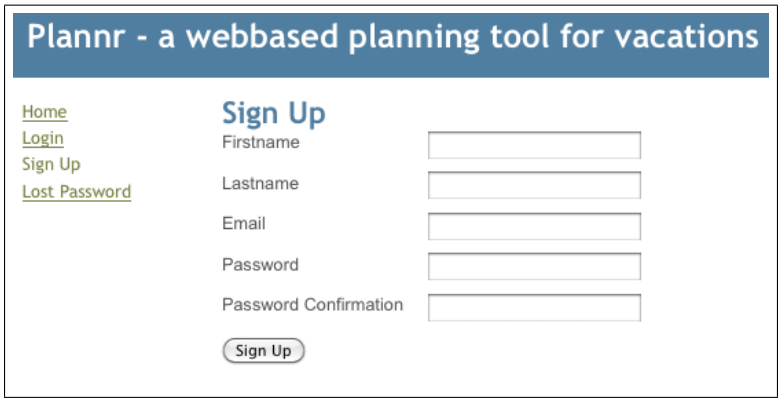

Abbildung 7.2: Registrierung

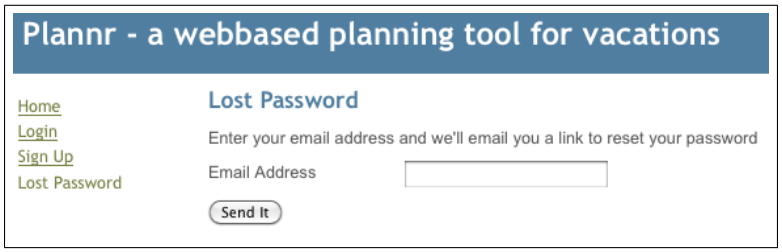

Abbildung 7.3: Lost Password

7.1.2 Funktionsumfang für registrierte Benutzer

Nun kommen wir zu den Seiten, die nur dem angemeldeten Benutzer zur Verfügung stehen. Die Seiten zur Benutzeradministration und Änderung des Passworts sind ebenfalls in HTML und CSS umgesetzt.

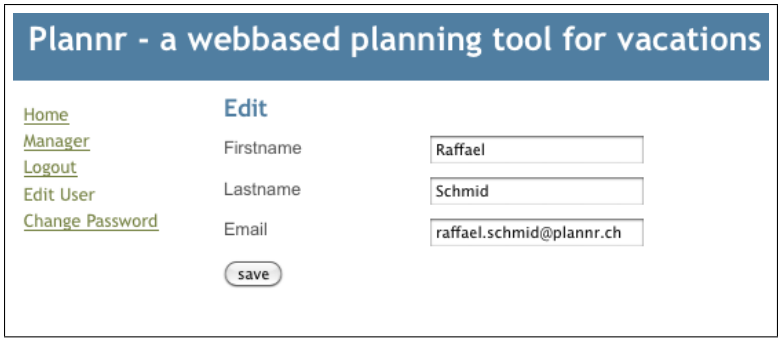

Abbildung 7.4: Edit User

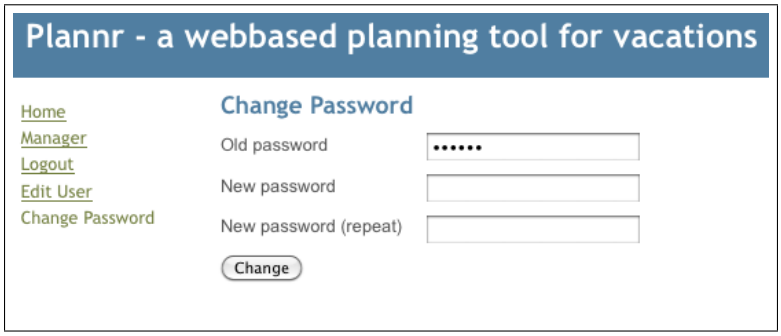

Abbildung 7.5: Change Password

Ab hier sind die Funktionalitäten der in Flex implementierten Seiten Teamadministration und Kalender abgebildet. Im Grossen und Ganzen sind die Anforderungen funktional implementiert. **Hingegen funktioniert die Freischaltung durch den Administrator noch nicht - Ferien können zwar erfasst, editiert und gelöscht werden, hingegen können die Ferien seitens Administrator noch nicht freigeschaltet werden. Es ist mir ein Dorn im Auge, aber ich musste die dafür notwendige Zeit in die Dokumentation und die Umstellung der Architektur auf Dependency Injection investieren und kam nicht mehr zur Fertigstellung dieses Use Cases.**

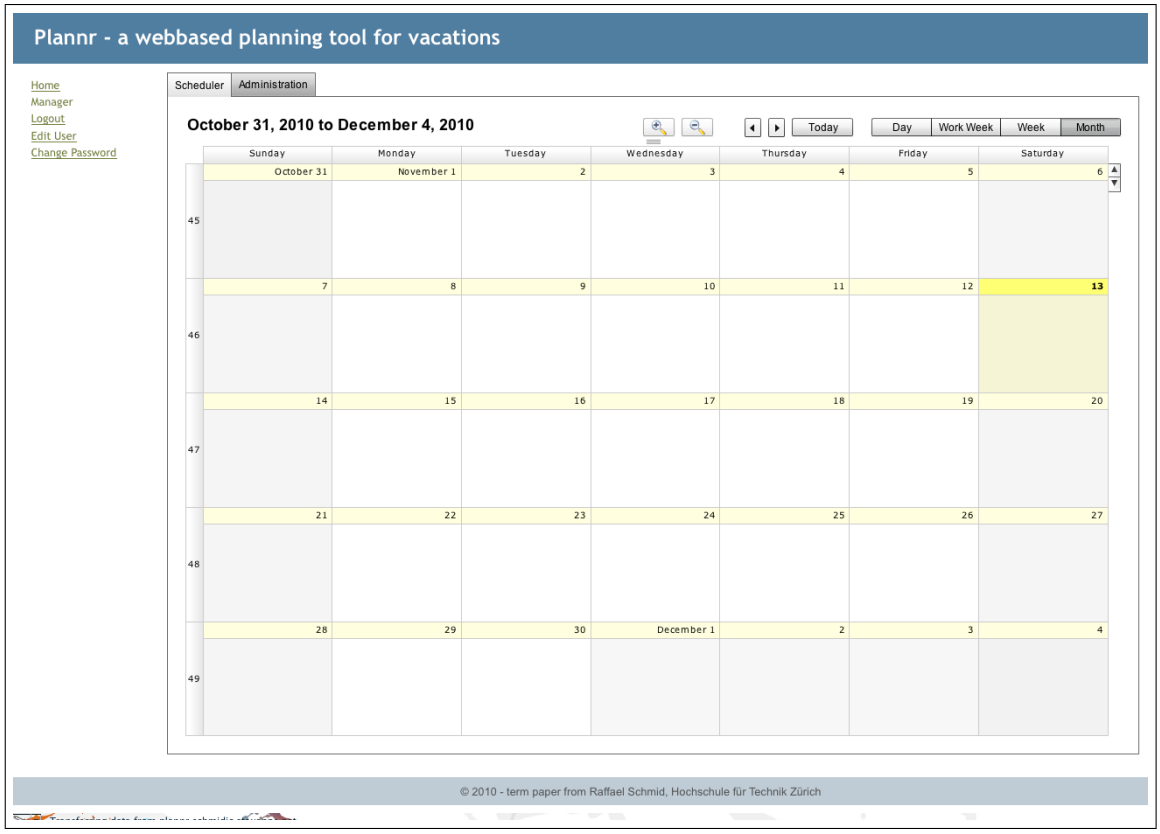

Abbildung 7.6: Ferienplanung

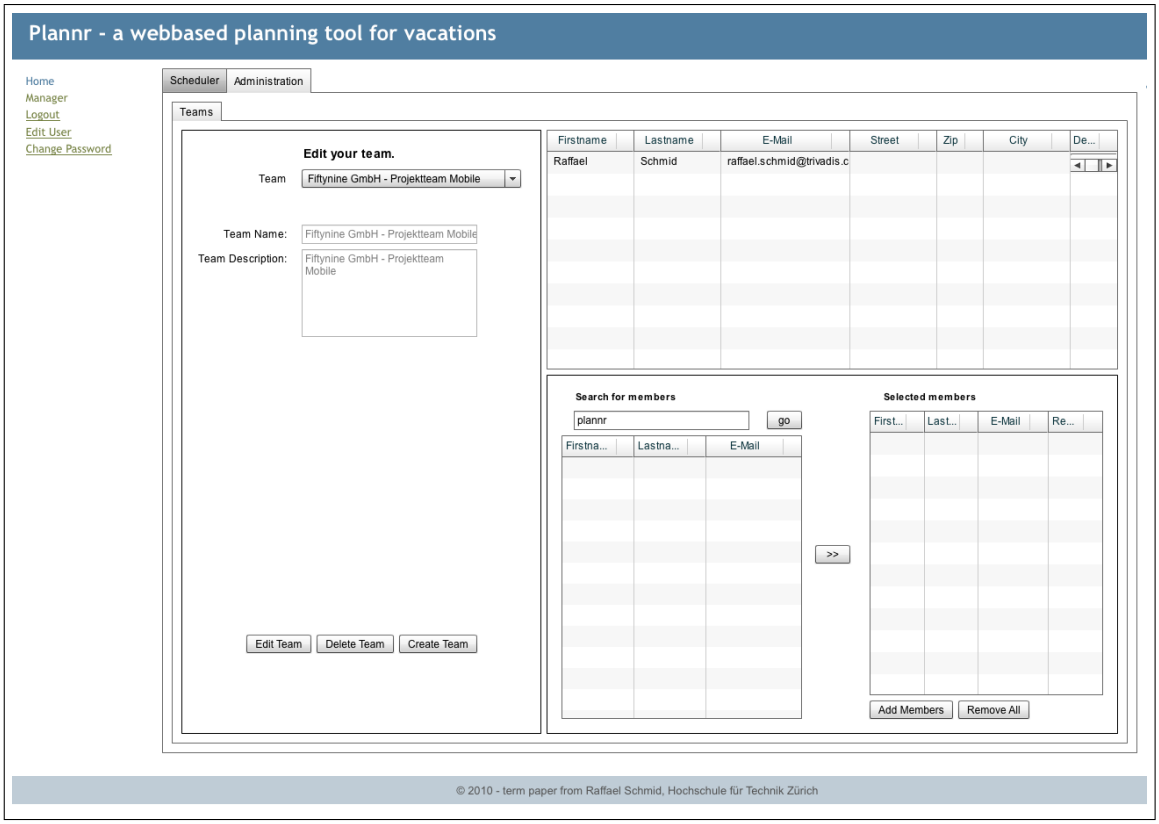

Abbildung 7.7: Teamadministration

7.2 Optionale Ziele Prototyp

Die von mir in der Aufgabe definierten optionalen Ziele sind Kriterien, welche heutzutage zum Erfolg einer Applikation beitragen. Hier wäre es deshalb interessant gewesen, wie das Lift Framework in diesen Bereichen abschliesst. Im Bereich von Internationalisierung, Search Engine Optimization und Usability trotzdem ein paar Feststellungen:

- **Internationalisierung:** Internationalisierung lässt sich in Lift wie in anderen Java-Frameworks auf der Basis von Properties einfach lösen. Siehe Abschnitt “3.3.9 Internationalisierung”
- **Search Engine Optimization:** Aus meiner Sicht haben die meisten SEO-Massnahmen mit den folgenden, unabhängig vom verwendeten Framework, aufgelisteten Punkten einer Webseite zu tun und sind deshalb für die Auswahl dessen nicht besonders relevant:
  - Inhalt

  - Meta-Daten
  - Mehrsprachigkeit
  - Sprechende Links

- **Usability:** Auch hier geht es m.E. in erster Linie um nicht Framework-relevante Punkte wie:
  - Design und Inhalt der Seite
  - Navigationsstruktur
  - Fehlertoleranz

Kapitel 8

Fazit der verwendeten Technologien

8.1 Scala

Subjektiv gesehen hat man stark das Gefühl, dass Funktionale Sprachen, insbesondere Sprachen auf der Java- oder Dot-Net-Plattform in letzter Zeit das Interesse der Entwicklergemeinde auf sich gezogen haben. Auch der Tiobe Index [21] hat diese Tendenz in den letzten Monaten bestätigt. Die erwähnten Sprachen haben oft eine "smartere" Syntax (siehe Abschnitt "3.1.1 Imperativ vs. Deklarativ"), lassen sich im Grossen und Ganzen besser parallelisieren und können in vielen Bereichen von bestehenden Bibliotheken und Frameworks ihrer bereits etablierten Plattformen profitieren. Die Zukunft gehört diesen Sprachen, da Problemstellungen damit in wesentlich weniger LOC implementiert werden können. Trotzdem behaupte ich, dass der Toolsupport noch lange nicht den Reifegrad wie bei anderen Sprachen erreicht hat und deshalb die Verbreitung der Sprache in der Industrie nur schleppend vonstatten geht. Auch die Nachfrage von Scala Entwicklern bei IT-Consulting unternehmen ist deshalb noch nicht so ausgeprägt. Trotzdem erweckt die Syntax und Semantik der Sprache bei vielen Begeisterung und ich möchte in den folgenden Abschnitten nochmals kurz darau eingehen.

8.1.1 Typisierung

Typisierung ist einer der ersten Kriterien zur Klassifizierung von Sprachen, und bekanntlich scheiden sich hier auch bereits die Geister. In der Folge habe ich die am wohl verbreitetsten Argumente aufgelistet und möchte anhand der statischen Sprache Scala zeigen, dass die meisten Argumente fälschlicherweise mit dem Thema Typisierung in Verbindung gebracht werden, obwohl sie eigentlich mit Typ Inferenz und Funktionen als "first-class citizens" zusammenhängen.

- Man sagt, dass statische Sprachen das DRY[1] verletzten. Als Beispiel wird genannt, dass man beispielsweise die Typen auf beiden Seiten des Gleichheitszeichen angeben müsse. Mit impliziter Typisierung sind bei Scala die Typen bereits bei Kompilierzeit bekannt.
- Man sagt, dass statische Sprachen eine hohe Verbosität haben. Dieser Punkt wird zum Beispiel im Zusammenhang mit anonymen Innerklassen genannt. Diese Aussagen kommen daher, dass typisierte Sprachen wie Java Funktionsobjekte und Closures bis jetzt leider noch nicht kennen.

Trotzdem, es gibt auch wirkliche Argumente die für die Programmierung mit dynamischen Sprachen sprechen. Unter anderem die geringere Kopplung zwischen Komponenten, die Verwendung von Interfaces kann per se reduziert werden und Klassen können auch aus diesem Grund wesentlich einfacher getestet werden.

8.1.2 Syntax

Bei der Einarbeitung in Scala fällt einem sofort folgendes auf:

- Scala ist statisch, stark typisiert, hat hervorragende Typ Inferenz
- Scala wurde von vielen Sprachen beeinflusst: Java, ML, Haskell
- Scala wurde von Personen entwickelt, die schon lange im Sprachdesign tätigt sind. Ich begründe diese Aussage mit der auffällig klaren Konsistenz in allen Bereichen und insbesondere
  - hinsichtlich Operator Overloading, respektive alles ist eine Methode.
  - hinsichtlich der Verwendung von Collections wie List und Map.
- Scala beinhaltet grossartige Konzepte wie
  - Pattern Matching.
  - Implizit Conversions.

8.1.3 Tool Support

Nebst der Tatsache, dass Scala Code um Längen mehr concise[2] ist als Java, wird man während der Entwicklung mit der Tatsache konfrontiert, dass das Tooling der Sprache hinterher hinkt:

- der Scala Compiler ist aktuell noch nicht genügend schnell, es ist viel warten angesagt
- die Code-Vervollständigung erkennt nur ganz einfache Zusammenhänge.

[1] Don't Repeat Yourself, ein Grundsatz für guten Code Smell
[2] englisches Wort für prägnant, tönt einfach besser

8.1.4 Verbreitung

Trotz der Schönheit und Kosistenz dieser Sprache, bin ich sehr gespannt, wie sie sich in der nahen Zukunft auch in der Industrie festsetzen wird. Ich bin der Meinung, dass die Verbreitung von Scala ohne ein Framework im Bereich des Internets sehr schwierig sein wird. Web Frameworks können sich nur dann verbreiten, wenn durch eine augeklügelte Erweiterbarkeit ein Ökosystem rundherum entsteht. So dass es einerseits eine grosse Community gibt und Webseiten in kurzer Zeit und entsprechend kostengünstig entstehen können. Als Paradebeispiel in diesem Bereich sehe ich Rails aber auch Grails, Lift ist meines Erachtens aber noch zu wenig weit. Siehe Abschnitt "8.2 Lift Framework"

8.2 Lift Framework

Mit Maven wurde ein ausgeklügeltes Build-Tool gewählt, das die Basis für eine grosse Plattform werden könnte. Trotzdem ist der Einstieg in Lift auch wegen der für viele Leute neuen Programmiersprache. Trotzdem unterscheiden sich die Probleme und Fragen zu Beginn nicht von denen bei anderen Frameworks:

- Wie kann ich die Applikation innerhalb einer IDE bauen?
- Welche IDE ist die richtige?
- Wie kann ich die Applikation starten und debuggen?
- Wie kann ich die Unit-Tests starten und debuggen?
- Wie komme ich an die Sourcen des Lift Frameworks[3]?

8.2.1 Lift geht eigene Wege

Für die Implementation einer Applikation muss man Lift schon relativ gut kennen. Das Framework ist in vielen Bereichen sehr innovativ und geht entsprechend auch immer wieder andere Wege. Dies ist insofern verständlich, da bereits durch die Auswahl der Programmiersprache eine Fülle anderer Möglichkeiten entstehen. Ich kritisiere trotzdem diesen Entscheid im Bereich des **Objekt-Relationalen-Mappings** - da wo es offensichtlich, trotz den vielen guten Ideen nicht gelang, eine adäquate Alternative zu finden. Die JPA Anbindung hätte diese Ideen und Energie sicherlich gut gebrauchen können. Auch im Bereich des **Dependency Injections** gibt es ein Paar Frameworks, die diese Funktionalität besser und flexibler bereitstellen. Nebst Spring gibt es auch den Standard "Context and Dependency Injection for Java" , wo die Konfigurierbarkeit der Abhängigkeiten eine in meinen Augen noch nie da gewesene Flexibilität ermöglicht.

[3]Die Source-Dateien von Lift sind noch nicht sehr lange im Maven Repository.

8.2.2 Test Unterstützung

Die unterstützung für **Unit-Tests** ist in erster Linie nicht eine Frage des Frameworks, sondern hat mit dem Aufbau und der Architektur der Applikation zu tun. Klassen lassen sich dann gut testen, wenn die Kopplung zum Umsystem gering ist und es wird entsprechend schwierig, wenn die Abhängigkeiten der UUT[4] schlecht abstrahiert und von aussen nicht gesetzt werden können. Trotzdem gibt es ein paar interessante Punkte:

- Tests in Scala können auf der Basis fast aller bekannter Frameworks erstellt werden: JUnit, TestNG, ScalaTest, ScalaSpecs. Ich habe mich für die ScalaSpecs entschieden, diese lassen es zu, menschenlesbare Spezifikationen zu schreiben. Mittlerweile finde ich den durch die Specs entstehenden erhöhten Aufwand nur in wenigen Fällen adäquat.
- Es gibt Mock-Klassen für HttpServletRequest, HttpServletResponse und HttpSession, die sich im Lift TestFramework befinden. Für weitere Mocks siehe Package net.liftweb.mocks des Lift-Testkits.

Bei der Frage nach dem Framework-Support bei Integration Tests[5] trennt sich üblicherweise die Spreu vom Weizen. Die Unterstützung durch Lift ist relativ bescheiden - Frameworks wie Grails und Rails bieten dafür einen wesentlich grösseren Funktionsumfang.

8.2.3 Fazit

Um mich aufgrund der Kritik nicht falsch zu verstehen, ich werde Lift mit Sicherheit wieder einmal verwenden, allerdings würde ich zum aktuellen Zeitpunkt aufgrund der in meinen Augen noch mangelnden Reife vom Gebrauch für eine Business-Lösung abraten. Lift und Scala machen Spass, sind Hobby. Ich bin gespannt wie es weitergeht.

[4] Unit under Test

[5] dabei meine ich das Testen gesamter Gruppen, beinhaltet unter anderem auch die Anbindung an die Datenbank

Literaturverzeichnis

[1] Cairngorm. http://sourceforge.net/adobe/cairngorm/home.

[2] Dependency injection. http://www.assembla.com/wiki/show/liftweb/Dependency_Injection.

[3] Flex calendar components. http://code.google.com/p/flexcalendar.

[4] Git - fast version control system. http://git-scm.com.

[5] Granit data services (flex & java ee). http://www.graniteds.org.

[6] Integrating flex, blazeds, and scala/lift. http://opensource.adobe.com/wiki/display/blazeds/BlazeDS.

[7] Mate flex framework. http://mate.asfusion.com.

[8] Simile widgets timeline. http://www.simile-widgets.org/timeline.

[9] Simple build tool. http://code.google.com/p/simple-build-tool/.

[10] Stax. http://www.stax.net.

[11] Swiz framework for adobe flex. http://swizframework.org.

[12] Jonas Bonr. Real-world scala: Dependency injection (di). http://jonasboner.com/2008/10/06/real-world-scala-dependency-injection-di.html.

[13] D. Chen-Becker, M. Danciu, and T. Weir. The Definitive Guide to Lift. Springer, 2009.

[14] Lift Community. Lift web framework. http://liftweb.net.

[15] Liftweb. Using sbt. http://www.assembla.com/wiki/show/liftweb/Using_SBT, 2010. [Online; Stand 10. Oktober 2010].

[16] Matthias Zenger Martin Odersky. Scalable component abstractions. http://lamp.epfl.ch/ odersky/papers/ScalableComponent.pdf.

[17] M. Odersky, L. Spoon, and B. Venners. Programming in Scala. Artima Inc, 2008.

[18] Lothar Piepmeyer. Grundkurs funktionale Programmierung mit Scala. Hanser Fachbuchverlag, 6 2010.

[19] Raffael Schmid. Git repository - plannr. http://github.com/schmidic/plannr.

[20] Raffael Schmid. plannr - a web based planning tool for vacations. http://plannr.schmidic.staxapps.net.

[21] TIOBE. Tiobe software: Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[22] Wikipedia. Benutzerfreundlichkeit — wikipedia. http://de.wikipedia.org/w/index.php?title=Benutzerfreundlichkeit, 2010. [Online; Stand 7. November 2010].

[23] Wikipedia. Endrekursion — wikipedia. http://de.wikipedia.org/w/index.php?title=Endrekursion&oldid=78741723, 2010. [Online; Stand 3. Oktober 2010].

[24] Wikipedia. Gantt-diagramm — wikipedia, 2010. [Online; Stand 7. November 2010].

[25] Wikipedia. Lambda-kalkl — wikipedia, 2010. [Online; Stand 12. November 2010].

[26] Wikipedia. Nosql — wikipedia. http://de.wikipedia.org/w/index.php?title=NoSQL, 2010. [Online; Stand 11. Oktober 2010].

[27] Wikipedia. Representational state transfer — wikipedia. http://de.wikipedia.org/w/index.php?title=Representational_State_Transfer, 2010. [Online; Stand 6. November 2010].

[28] Derek Wischusen. Integrating flex, blazeds, and scala/lift. http://flexonrails.net/?p=103.

Listings

Abbildungsverzeichnis

Tabellenverzeichnis

Teil IV

Anhang

Anhang A

Informationen

A.1 Inhalt des Datenträgers

Pfad	Beschreibung
/	
/documentation	Beinhaltet alle Teile der Dokumentation: Semesterarbeit, Protokolle von Design-Review und Kickoff und die Aufgabenstellung.
/workspace-client	Workspace für den Flash-Builder, mittels welchem die Teile Teamadministration, Kalender für die Ferienplanung in Flex implementiert wurden.
/workspace-server	Hier befinden sich die Teile für das Backend und die Seiten, welche in HTML, CSS implementiert wurden.
/workspace-server/intellij-project	Projektdateien für die Entwicklungsumgebung
/workspace-server/plannr	Hier befinden sich die Source-Dateien für das Backend. Es beinhaltet zwei Projekte, "plannr-test" in welchem sich wenige Klassen zu Testzwecken befindet und "spa", in welchem sich die ganze Backend-Applikation befindet.

A.2 Diverses

Git Repository

Das Git Repository der Semesterarbeit befindet sich unter [19].

Url

Die Applikation befindet sich auf Stax und kann unter [20] gefunden werden.[1]

[1] Normalerweise dauert der Seitenaufbau beim ersten Aufruf relativ lange, da sich Applikationen auf Stax nach einem Tag wieder in den Hibernate Modus versetzen.