Erstellung eines Blueprints im Bereich Scala/Lift auf der Basis einer Applikation für die Ferienplanung

Semesterarbeit

vorgelegt am: 15. November 2010

an der Hochschule für Technik in Zürich

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 aus einem Flex-Client [1] der via REST Schnittstelle auf die Services im Hintergrund zugreifft. Zur persistierung wurde die Java Persistence API durch die Implementation Hibernate verwendet und als Programmiersprache wurde Scala verwendet. 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

I Projektdetails **11**

1 Aufgabenstellung **13**
1.1 Ausgangslage 13
1.2 Ziel der Arbeit 13
1.2.1 Optionale Ziele 14
1.3 Aufgabenstellung 14
1.4 Erwartetes Resultat 15

II Umsetzung **17**

2 Analyse der Aufgabenstellung **19**
2.1 Idee und Ziele der Arbeit 19
2.1.1 Vorbereitung: Erarbeitung des Basiswissens 19
2.1.2 Design 20
2.1.3 Technische Umsetzung 20
2.2 Lieferumfang der Semesterarbeit 20
2.2.1 Dokumentation 20
2.2.2 Prototyp 20

3 Grundlagen **23**
3.1 Begriffserklärungen zur Klassifizierung von Programmiersprachen 23
3.1.1 Funktionale Programmierung 23
3.1.2 Statisch typisierte Sprachen 25
3.2 Scala 26
3.2.1 Scala Type Inferenz 27
3.2.2 Traits 27
3.2.3 Funktionen als Objekte 29

3.2.4 Currying . . . 30
3.2.5 Pattern Matching . . . 30
3.2.6 Tail Recursion . . . 31
3.2.7 Predef . . . 31
3.2.8 Implicit Conversion . . . 32
3.2.9 XML Datentyp . . . 32
3.2.10 Integration mit Java . . . 32
3.3 Liftweb Framework . . . 32
3.3.1 Erstellen eines Lift-Projektes . . . 33
3.3.2 Bootstrapping [6, p. 26] . . . 34
3.3.3 Site Rendering [6, p. 27-43] . . . 34
3.3.4 Formulare . . . 36
3.3.5 SiteMap [6, p. 61-70] . . . 36
3.3.6 Persistenz . . . 36
3.3.7 Internationalisierung . . . 39
3.3.8 Fazit . . . 40
3.4 Stax Cloud Plattform . . . 40

4 Design und Konzeption 41
4.1 Use Case Definitionen . . . 42
4.1.1 Aktoren . . . 43
4.1.2 Beschreibung der Use Cases . . . 43
4.2 Rollen-Konzept . . . 44
4.2.1 Anonymous . . . 44
4.2.2 Registrierte Benutzer . . . 44
4.2.3 Optional Aufteilung der Registrierten Benutzer . . . 45
4.3 Datenbank-Schema . . . 45
4.3.1 Entity Relationship Model . . . 45
4.4 Prozesse . . . 48
4.4.1 Person registrieren . . . 48
4.4.2 Ferien beantragen, planen . . . 48
4.4.3 Team administrieren . . . 49
4.5 Navigations-Konzept . . . 50
4.6 Architektur . . . 50

5 Implementation 51
5.1 Persistenz . . . 51
5.1.1 Auswahl Persistenz-Provider . . . 51
5.1.2 Domain Mapping . . . 52
5.1.3 Validation . . . 54

5.2 Benutzermanagement 55
5.3 Navigation 57
5.4 Flex Client 57
5.4.1 Architektur 57
5.4.2 Ferienplanung 60
5.4.3 Teammanagement 60
5.5 Mail-Versand (Notifikationen) 60

6 Entwicklung, Build und Deployment 61
6.1 Build-System 61
6.2 Entwicklungsumgebung 62
6.3 Test- und Produktiv-Umgebung 62

III Rückblick 65

7 Analyse der Arbeit auf der Basis der Aufgabenstellung 67
7.1 Prototyp 67
7.2 Optionale Ziele 67
7.2.1 Setup von Test-und Produktiv-Umgebung 67
7.2.2 Performance Tests 67
7.2.3 Internationalization 67
7.2.4 Search Engine Optimization 67
7.2.5 Usability 67

8 Fazit 69

IV Anhang 79

A Glossar 81

B Journal 83
B.1 Phase Implementation Backend 83
B.2 Phase Implementation Frontend 84
B.3 Phase Dokumentation 85

Einleitung

TODO

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

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

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 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

Das grundsätzliche Ziel hinter der vorgelegten Semesterarbeit war es, sich mit dem Lift Webframework und Scala auseinander zu setzen um im Anschluss daran eine Aussage darüber machen zu knnen, ob sich die beiden Technologien in naher Zukunft als Plattform zur Entwicklung einer Webseite anbieten würden. Zu Beginn steht auch die Erarbeitung des Knowhows in den 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 Punkten unterscheiden, mit Lift umsetzen lassen:

- Authentifizierung, Authorisierung
- Persistenz respektive Objekt-Relationales Mapping
- Internationalisierung

Die soeben beschriebenen Punkte sind im Abschnit 3.3 Liftweb Framework dokumentiert.

2.1.2 Design

Nachdem die Einarbeitung abgeschlossen ist, kann mit dem Design der Arbeit begonnen werden. Dieser Punkt beinhaltet zum einen die Definition der Prozesse und zum anderen das Design des Datenbank Schemas. Beide Punkte sind angesichts der überschaulichen Problemstellung des Ferienplaners zeitlich nicht sehr aufwändig.

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.

2.2 Lieferumfang der Semesterarbeit

2.2.1 Dokumentation

Ausgehend von der Aufgabenstellung muss die Dokumentation folgende Teile beinhalten:

- Konzepte
  - Navigationskonzept befindet sich im Abschnitt 4.5 Navigations-Konzept.
  - Rollenkonzept befindet sich im Abschnitt 4.2 Rollen-Konzept.
  - Die Definitionen der Prozesse befinden sich Abschnitt 4.4 Prozesse
- Implementationsdetails befinden sich im Abschnitt 5 Implementation

(aufgeteilt in Navigationskonzept, Rollenkonzept und Prozesse) und die Details zur Implementation (inklusive Begründung, weshalb wann welche Technologien oder Module verwendet wurden) beinhalten.

2.2.2 Prototyp

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 können 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

3.1 Begriffserklärungen zur Klassifizierung von Programmiersprachen

3.1.1 Funktionale Programmierung

Um das Prinzip der Funktionalen Programmierung zu verstehen hier ein kurzer Vergleich zwischen imperativer und deklarativer Programmierung.

Imperativ vs. Deklarativ

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

Als gutes Beispiel für eine deklarative Sprache ist SQL, die Structured Query Language zur Abfrage von Daten einer Datenbank, und ist deshalb ein gutes Beispiel für eine Sprache die unserem Denken entspricht.

Listing 3.1: Sql Deklaration

```
1 select first_name, last_name, zip, city
2 from tbl_user
3 where zip<=8000;
```

Tabelle 3.1: Resultat der deklarativen Abfrage

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

firstname	lastname	zip	city
Flavor	Flav	8000	Zürich

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 nicht einmal wissen, wie Computer funktionieren. Mit Hilfe der Abfragesprache können wir uns auf das Wesentliche konzentrieren und mit wenigen Anweisungen viel erreichen. [9]

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 höheren Sprachen wie zum Beispiel Scala wird die Berechnung der Summe auf deklarative Weise gemacht und sieht folgendermassen aus:

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 Top-Class Funktionen welche nicht nur definiert und aufgerufen werden können, sondern als Werte respektive Objekte herumgereicht werden können.

- Die theoretische Grundlage von Funktionaler Programmiersprachen basiert auf dem Lambda-Kalkül[2] Jeder Ausdruck wird dabei als auswertbare Funktion betrachtet, so dass Funktionen als Parameter übergeben werden können.

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. Dies kann im wesentlichen durch 2 verschiedene Arten geschehen:

Explizite Deklaration und Typinferenz

Bei der expliziten Deklaration wird der Typ einer Variablen respektive der Rückgabetyp einer Funktion 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 Typinferenz. Die heutigen Programmiersprachen besitzen unterschiedliche Fähigkeiten in Sachen Typinferent.

Typinferenz in Java

In Sachen Typinferenz ist Java wenige begütert. Ein kleines Beispiel welches das kleine bisschen Typinferenz in Java aufzeigen soll:

Listing 3.4: Typeinferenz 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 }
```

[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. http://de.wikipedia.org/wiki/Lambda-Kalkül

Die Ermittlung des Rückgabetyps aufgrund des Variablen-Typs ist schon fast alles was Java in Sachen Typeinferenz zu bieten hat.

Vorteile von statischer Typisierung

- Bestimmte Fehler werden durch die Typprüfung während der Kompilierzeit vermieden.
- Grundsätzlich ist das akribische Testen von Code weniger wichtig.
- Die Performance von statisch typisierten Sprachen ist deshalb besser, weil die ermittlung des Typs zur Laufzeit in den meisten Fällen vermieden werden kann.

Nachteile von statischer Typisierung

- Dynamische Sprachen ermöglichen eine höhere Flexibilität. Zum Beispiel können folgende Dinge in statischen Sprachen teilweise relativ schn, aber mit erhhtem aufwand gemacht werden:
  - Einfügen von Methoden in Classen oder Objekte zur Laufzeit in Java ist beispielsweise mit AspectJ[3] möglich, herkmmliche 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]
- Kompilieraufwand ist wesentlich grösser.

3.2 Scala

Scala wird von Martin Odersky seit 2003 an der EPFL in Lausanne entwickelt. Trotzdem dass Scala an einer Hochschule entwickelt wurde, handelt

[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

es sich dabei um eine Sprache fr den industriellen Gebrauch. Dies ist unter anderem auch deshalb möglich, da die Sprache auf der Java Plattform aufbaut und deshalb andere Frameworks wie zum Beispiel JPA Bibliotheken verwendet werden knnen. Die Ideologie hinter Scala lässt sich durch die folgenden beiden Begriffe umschreiben:

- **Concise[5] - Dieser Begriff wird in Büchern über Scala pro Seite gefühlte 10 mal. Mehr dazu unter Ausdrucksstärke.**
- **Consistent[6] - Mehr dazu in der Sektion Consistenz.**

Im Anschluss werden fünf Konzepte von Scala vorgestellt, welche die Sprache und ihre Schönheit aufzeigen.

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 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 fr die Wiederverwendbarkeit von Code. Im Gegensatz zu der Klassenvererbung knnen unzhlige Traits[7] eingemischt werden und aufgrund der Linearisierung dieser "mixins"knnen bekannte Probleme wie sie in der Mehrfachvererbung vorkommen vermieden werden. Zur erklärung von Traits in Scala ein Beispiel[8, p. 222-227]:

Listing 3.6: Klassen und Traits definieren

[5] übersetzt prägnant
[6] übersetzt einheitlich
[7] bedeutet übersetzt Eigenschaft respektive Merkmal

```
1 import scala.collection.mutable.ArrayBuffer
2
3 abstract class IntQueue{
4    def get():Int
5    def put(x:Int)
6 }
7
8 class BasicIntQueue extends IntQueue{
9    private val buf = new ArrayBuffer[Int]
10   def get() = buf.remove(0)
11   def put(x:Int){buf+=x}
12 }
13
14 trait Doubling extends IntQueue{
15   abstract override def put(x:Int){super.put(2*x)}
16 }
17
18 trait Incrementing extends IntQueue{
19   abstract override def put(x:Int){super.put(x+1)}
20 }
```

Hier haben wir eine abstrakte Klasse IntQueue deklariert, welche keine beider Methoden implementiert. Anschliessend implementieren wir beide Methoden in einer Basisklasse BasicIntQueue und erstellen zwei, welche die übergebene Zahl in die put-Methode je inkrementieren respektive verdoppeln. In Java wurde ein solches Verhalten bis anhin vorzugsweise mit dem Delegate-Pattern implementiert. Wie wir die Traits verwenden sehen wir nun im Folgenden Code-Ausschnitt.

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 wird. Dann eine Incrementing-Doubling-Queue, bei welcher die übergebene Varialbe incrementiert und dann verdoppelt wird, eine Increment-Queue (iQ) und eine 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 durchgefhrt, damit Klassen nicht mehrmals aufgerufen werden fhrt der Scala Compiler bei der Instanzierung eine wie folgt definierte Linearisierung durch. Sofern eine Klasse in der Vererbungshierarchie mehrmals vorkommt, wird nur die erste 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**

Die Übergebenen Argumente werden nun zuerst verdoppelt und inkrementiert, bevor sie in die Queue gestellt werden.

3.2.3 Funktionen als Objekte

Scala erfüllt die wichtigsten Kriterien, die eine Sprache als Funktional bezeichnen lassen[9, 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 programmiert man Funktionen respektive Methoden mit Seiteneffekten.

3.2.4 Currying

Currying[8] 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 als nur die Anwendung aus Text - die Idee gibt es allerdings schon viel länger, wurde sie zum ersten Mal in ML[9] verwendet.

In Scala können unterschiedliche Typen von Mustern auf Objekte angewendet werden:

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

[9] http://de.wikipedia.org/wiki/ML_(Programmiersprache)

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

Mehr Informationen gibt es unter [8, p. 263-296] oder [9, p. 167-176]
Ursprünglich kommt die Idee von

3.2.6 Tail Recursion

Wie jede rekursive Funktion lassen sich Endrekursive[10] Funktion mittels einer Iteration darstellen, dabei sind die iterativen Varianten oft auch wesentlich sparsamer mit Ressourcen, da fr jeden Funktionsaufruf ein Frame auf dem Stack erstellt wird. Allerdings lassen sich gewisse Problemstellungen wesentlich lesbarer mit Rekursion darstellen. Sogar in Java gibt es Compiler die es schaffen, Endrekursive Funktionen zu optimieren. Der Scala Compiler wandelt diese in Iterationen um. Seit der Scala Version 2.8 bietet der Compiler die Möglichkeit, die Umwandlung Ëndrekursiv - Iteration"mit der Annotation **@endrec** zu Überprüfen.

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. Ein paar Beispiele, die via dieses Objekt implizit definiert sind:

- 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 ist es einigermassen erstaunlich, warum die Klasse java.lang.String plötzlich eine Methode reverse besitzt. Unbemerkt haben wir es hier mit einer Impliziten Konversion der Klasse Predef zu tun. Diese respektive deren Super-Typ LowPriorityImplicits besitzt die Methode mit der folgenden definition:

Listing 3.11: Implicit Conversions Method wrapString

```
1 implicit def wrapString(s:String):WrappedString = {
2    new WrappedString(s)
3 }
```

Sofern der Typ die aufgerufene Methode reverse nicht hat, wird im Gültigkeitsbereich nach einer Impliziten Conversion gesucht die zu einem Rückgabetyp mit dieser Methode führt. Das ganze wird zur Kompilierzeit gemacht.

3.2.9 XML Datentyp

TODO

3.2.10 Integration mit Java

TODO

3.3 Liftweb Framework

Im Prinzip war die Entscheidung, welches Web Framework zu verwenden ist, bereits in der Aufgabenstellung definiert. Ziel war es viel mehr, eine Analyse des als Grundlage definierten Lift Frameworks zu erstellen und 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. Im Anschluss an diese Punkte werde ich zusätzlich versuchen,

den Vergleich mit Grails, eines auf Java, Spring, Hibernate und Groovy basierenden Webframeworks, herzustellen.

3.3.1 Erstellen eines Lift-Projektes

Innerhalb des ganzen Lift-Ökosystems wird Maven[10] als das Build-System verwendet. Mittels der vordefinierten Maven Archetypen[11] können Lift-Projekte mit relativ geringem Aufwand erstellt 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[12] (lift-archetype-basic), usw. Die Standardisierung an die sich Maven-Projekte halten[13] vereinfachen die verschiedensten Phasen der Software-Entwicklung. Zum Beispiel können Maven-Projekte in die meisten Continuous Integration Systeme ohne erheblichen Aufwand importiert werden.

Mittels folgendem Befehl lässt sich ein Wizard starten mittels welchem man ein neues Maven-Projekt erstellen kann. Die Auswahl des Archetypen kann zu Beginn gemacht werden. Aktuell befinden sich die Lift-Archetypen zwischen ca. Position 21 und 39.

Listing 3.12: Erstellung eines Lift-Projektes

```
1 mvn archetype:generate
```

Lift- respektive Maven-Projekte lassen sich in den "gängigen"[14] importieren. Es sind dafür noch 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[15] (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 [7] nachgeschaut werden.

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

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

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

[13] Convention over Configuration

[14] damit wird Eclipse, IntelliJ und Netbeans gemeint

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

3.3.2 Bootstrapping [6, 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, Url-Rewriting konfiguriert werden. Die Boot.scala Datei befindet sich per Default im Verzeichnis bootstrap.liftweb, was sich in besonderen Fällen[16] via web.xml anpassen lässt.

3.3.3 Site Rendering [6, 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, und nicht ein Template oder eine View angezeigt werden sollen.

3. Im letzten Schritt wird gesucht, ob ein Template oder eine View für die Url vorhanden ist und diese entsprechend gerendert.

Rendering mit Templates

Templates sind vordefinierte XML-Dateien, welche HTML und vordefinierte Lift-Tags enthalten können. Anhand der hineinkommenden Url (Beispiel: /path/file) wird nacheinander versucht, die Dateien template_de-CH (Locale: de-CH), template_de (Locale: de) oder template mit je den Endungen .html, .htm und .xhtml aufzulösen. Die Templates können zum Beispiel folgenden Inhalt enthalten:

Listing 3.13: Lift Template Surround

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

Die Tags werden von aussen nach innen transformiert, entsprechend wird hier das Default-Template angezogen und beim Tag

[16] davon wird aber ziemlich vehement abgeraten

Listing 3.14: Lift Template Binding

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

der Output der Methode world von der Klasse Hello eingefügt. Bei der Klasse Hello spricht man von einem Snippet. Es handelt sich allerdings um eine normale Scala Klasse, die in der Methode world eine Objekt vom Typ scala.xml.Elem zurück gibt.

Listing 3.15: Snippet

```
1 class Hello {
2   def world = <h1>Hello World</h1>
3 }
```

Rendering mit Views

Als Alternative zu Template Dateien kann HTML-Code auch direkt aus den Methoden generiert werden. Um das Dispatching auf die entsprechenden Klassen und Methoden zu ermöglichen, wird am Besten der Trait LiftView verwendet und die Methode dispatch überschrieben. Ein Beispiel sieht folgendermassen aus:

Listing 3.16: Views

```
1 class ExpenseView extends LiftView{
2     override def dispatch = {
3        case "enumerate" => doEnumerate _
4     }
5     def doEnumerate() :NodeSeq:{
6         ...
7         <lift:surround with="default" at="content">
8             {expenseItems.toTable}
9         </lift>
10    }
11 }
```

Views müssen sich im Package default-namespace.views befinden. Hier wird eine Dispatch-Funktion von "/ExpenseView/enumerateäuf die Methode doEnumerate durchgeführt. Mit dieser zusätzlichen Definition stellt man sicher, dass nicht alle Methoden via eine Url ansprechbar sind. Das Resultat wird nachträglich prozessiert und äquivalent wie Templates behandelt - der Rückgabewert des Beispiels kann ebenfalls wieder Lift-Tags enthalten.

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 [6, p. 47-58] nachgeschlagen werden.

3.3.5 SiteMap [6, p. 61-70]

Grundsätzlich stellt die Lift SiteMap die Menu-Struktur mit entsprechenden Links zur Verfügung und kann mittels der Unordered List auf der Webseite eingefügt werden. Des weiteren bietet aber die SiteMap noch eine Vielzahl anderer Funktionen:

- Hierarchien und Gruppierungen von Elementen der Navigation, somit können auch nur einzelne Äste der Navigation angezeigt werden.
- Zugriffskontrolle auf die einzelnen Elemente
- Request-Rewriting

3.3.6 Persistenz

Relationale Datenbanken

Im Bereich der Persistenz mit relationalen Datenbanken habe ich mir die drei Verfügbaren Ansätze angeschaut. Die Entwickler hinter dem Lift-Framework gingen den Weg, dass sie vorallem zu Beginn versuchten, eigene OR-Mapper respektive eigene Persistenz Frameworks auf der Basis von Scala zu entwickeln.

Mapper - Das originale 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[17]-Operationen für Objekte und sticht vorallem im Bereich des Scaffoldings[18] heraus.

Ein Beispiel für das Mapping einer User-Klasse[19] sieht folgendermassen aus:

[17] Create, Read, Update, Delete

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

[19] Quelle ist das Mapper Framework

Listing 3.17: 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 an den Persistenz-Layer**, insbesondere auch in Service-Klassen und Controller, sehr gross wird.

Record - Die überarbeitet Version dieser Bibliothek Record [20] 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. Es bietet sich dabei sogleich an, als JPA-Implementation Hibernate zu verwenden. Lift-JPA-Applikationen können mit dem Maven Archetype "lift-archetype-jpa-basic" erzeugt werden. Wie im folgenden Beispiel beschrieben können die Annotationen wie anhin in Java Domänenklassen verwendet werden:

Listing 3.18: Property Mapping mit JPA

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

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

Kompliziertere Mappings werden in Java mittels Nested-Annotations[21] erstellt. Seit Scala 2.8 können diese nun folgendermassen definiert werden:

Listing 3.19: Relation Mapping mit JPA

```
@ManyToMany
  @JoinTable(
    name = "MEMBERSHIP",
    joinColumns = Array(
                          new JoinColumn(
                            name = "USER_ID",
                            referencedColumnName = "ID")
                          ),
    inverseJoinColumns = Array(
                          new JoinColumn(
                            name = "TEAM_ID",
                            referencedColumnName = "ID")
                          )
                        )
  @BeanProperty
  var memberOf=new _root_.java.util.HashSet[Team]()
```

NoSql-Datenbanken

In den vergangenen 1-2 Jahren haben Objektorientierte- und Dokumentenorientierte-Datenbanken[22] stark an Wichtigkeit gewonnen. Man[11] begründet die dies vorallem damit, dass Relationale Datenbanken mit bestimmten Charakteristiken heute vorkommender Daten-Manipulationen[23] an Leistungsgrenzen gelangen. Die Scala-Entwickler haben im Bereich der NoSql Datenbanken für relativ viel Wirbel gesorgt, dies sicherlich auch deshalb, weil grosse Plattformen wie Novel oder Foursquare auf diesen beiden Technologien aufgebaut sind. Für die meisten dieser Datenbanken sind Treiber in Scala oder mindestens Java vorhanden:

[21] Annotation innerhalb einer anderen Annotation

[22] Zusammengefasst spricht man von NoSql (Not only Sql) Datenbanken

[23] Mehrheitlich haben sie Probleme mit hoher Anzahl an Datenänderungen und gleichzeitig hohen Datenvolumen

- MongoDB[24]
- CouchDB[25]
- BigTable

Die Integration von MongoDB wird via den oben erwähnten Java-Treiber gemacht, auf die CouchDB wird auf der Basis einer Thirdparty Library via RESTful Http Requests zugegriffen, aber auch hier bietet das Lift-Framework einen dünnen Wrapper um die Libraries.

3.3.7 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 erhalten 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 3.20: Überschreibung der Locale-Berechnung

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

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

[24] http://github.com/mongodb/mongo-java-driver
[25] http://code.google.com/p/scouchdb

3.3.8 Fazit

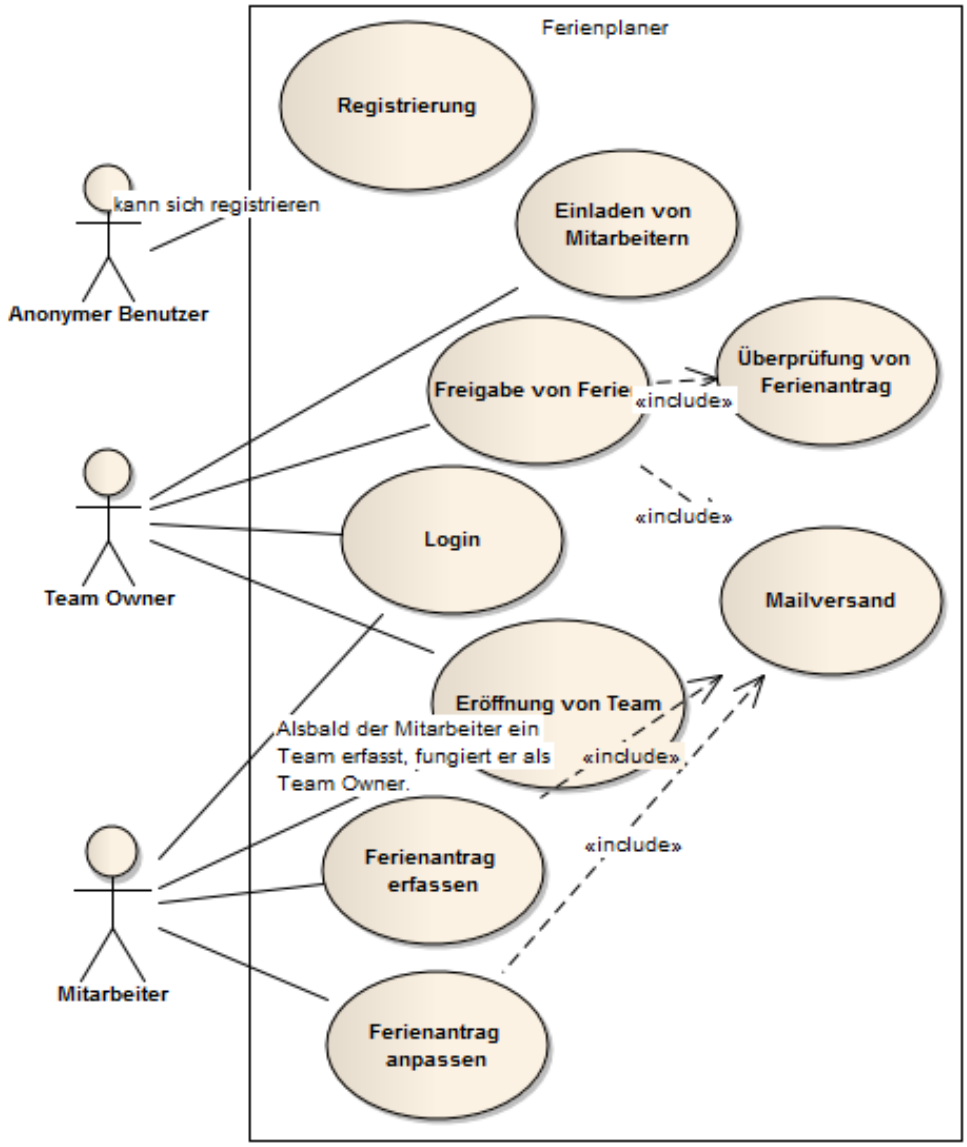
3.4 Stax Cloud Plattform

Die Stax Cloud Plattform[4] 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.

Kapitel 4

Design und Konzeption

4.1 Use Case Definitionen



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 gehört ein Projekt mehreren Administratoren, in einem 2. Projekt wird die zusätzliche Möglichkeit implementiert, anderen Benutzer Administrator-Rechte fürs eigene Projekt zu geben.

Mitarbeiter in Projekt erfassen

Projekte alleine sind leere "Behälter" für Mitarbeiter. Um Mitarbeiter ins eigene Projekt zu nehmen, sucht der Administrator nach einem bestehenden User. Sofern es den gesuchten Benutzer im System noch nicht gibt, wird er durch den Administrator neu erfasst. Der neu erfasste Benutzer wird mittels Mail auf diese Aktion aufmerksam gemacht.

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.

Ferienwunsch bearbeiten

Sollte ein Ferienwunsch abgelehnt werden, kann er erneut bearbeitet werden, was eine erneute Anfrage beim Administrator zur Folge hat.

4.2 Rollen-Konzept

Im Grunde handelt es sich bei dem Ferienplaner um einen Prototypen ohne eigentlichen Business Case im Hintergrund. 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. Als Business Case könnte ich mir vorstellen, dass es eine Trennung des Owners vom Mitarbeiter gibt, und dafür spezielle Bedingungen (z.B. Bezahlung, etc.) erfüllt werden müssen.

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

Bei allfälligem Business Case bestünde die Möglichkeit, die Rolle des Registrierten Benutzers in Mitarbeiter und Team Owner aufzuteilen. In diesem Falle hätte man die Möglichkeit, bestimmte Richtlinien (Zahlungsrichtlinien, etc.) zu errichten. 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 Datenbank-Schema

4.3.1 Entity Relationship Model

Das Entity Relationship Model zeigt alle Tabellen, wie sie fr die Datenbank geplant werden. Many-to-Many Beziehungen sind also bereits in relationaler Form aufgelöst.

User und Rollen

Beginnen wir mit den Tabellen User und Rollen: Für die Modellierung der Rollen gibt es meines Erachtens zwei vertretbare Alternativen. Erstens wie hier im ERM aufgezeigt, werden die Rollen via eine Join Table dem User zugewiesen. Die Erweiterung davon wäre dann eine Zusätzlichen Self-Join auf der Tabelle TBL_ROLE, damit man eine Hierarchie der Rollen abbilden kann, und Rollen den Benutzern auch in Gruppen zuweisen kann. Aufgrund der zusätzlichen Komplexitätsstufe habe ich mich für Variante eins entschieden.

Teammitgliedschaft, Membership

Die Zuteilung von Membern zu Teams muss ebenfalls über eine Many-to-Many Assoziation realisiert werden. Im Diagramm ersichtlich an der Tabelle MEMBERSHIP.

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 Ersteller und 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 relativ einleuchten, da diese immer einem Teilnehmer zugeordnet werden. Die Relation zu TBL_TEAM kommt daher, weil ich Ferien immer für ein spezifisches Team erfassen muss. Ob das dann manuell durch den Benutzer oder Applikatorisch für alle Teams durchgeführt wird, sei dahin gestellt. Jedenfalls, sofern ein Benutzer mehreren Teams zugeordnet ist, müssen für all diese Teams Ferien-Records vorhanden sein.

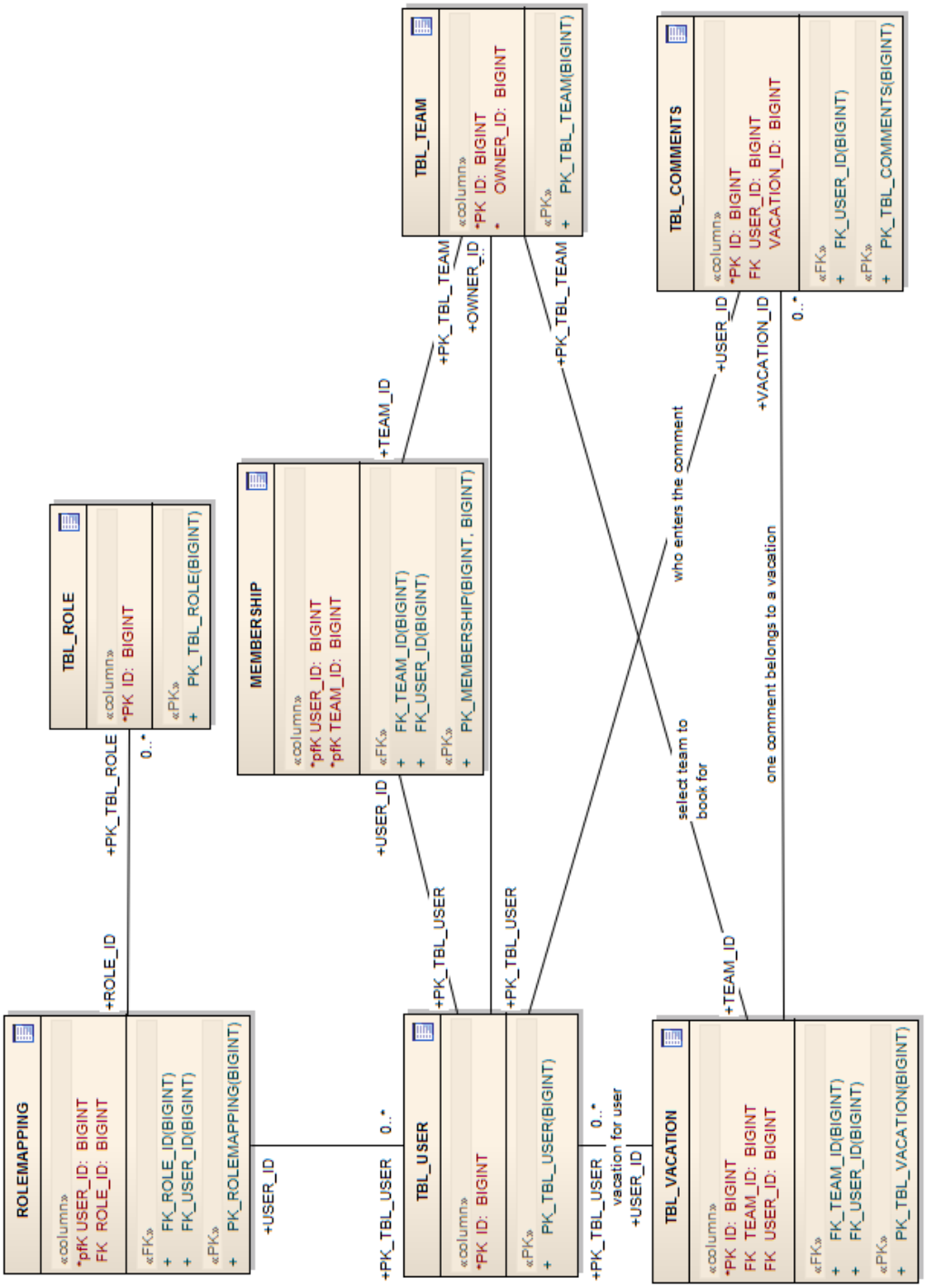


Abbildung 4.2: Entity Relationship Model

4.4 Prozesse

4.4.1 Person registrieren

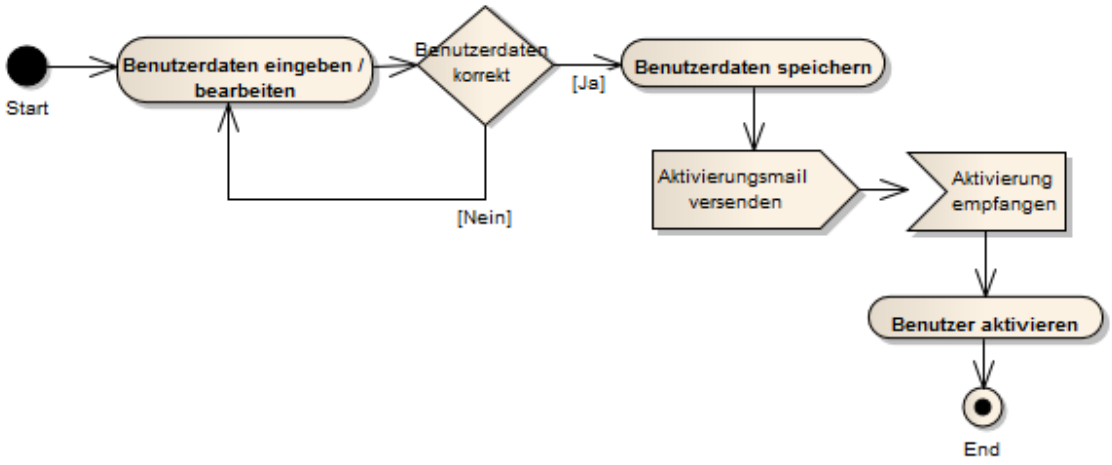


Abbildung 4.3: Prozess Member Administration Webpage

4.4.2 Ferien beantragen, planen

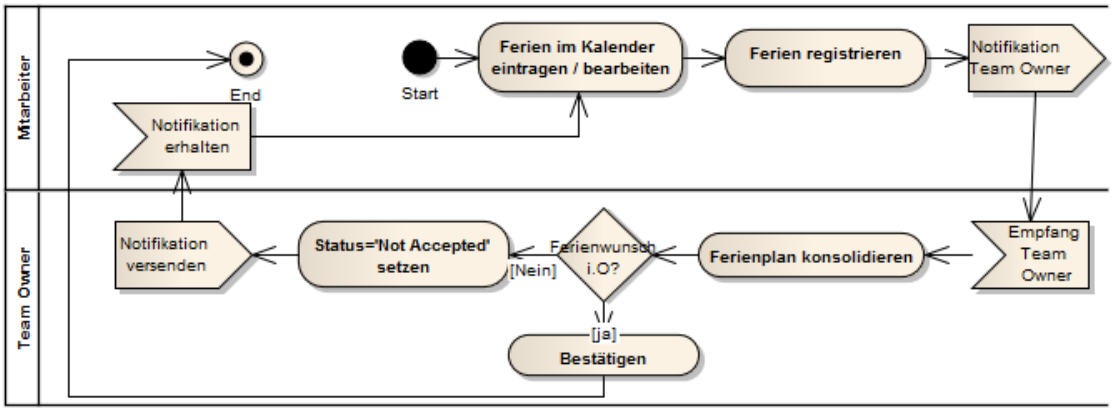


Abbildung 4.4: Prozess Ferien beantragen, planen

4.4.3 Team administrieren

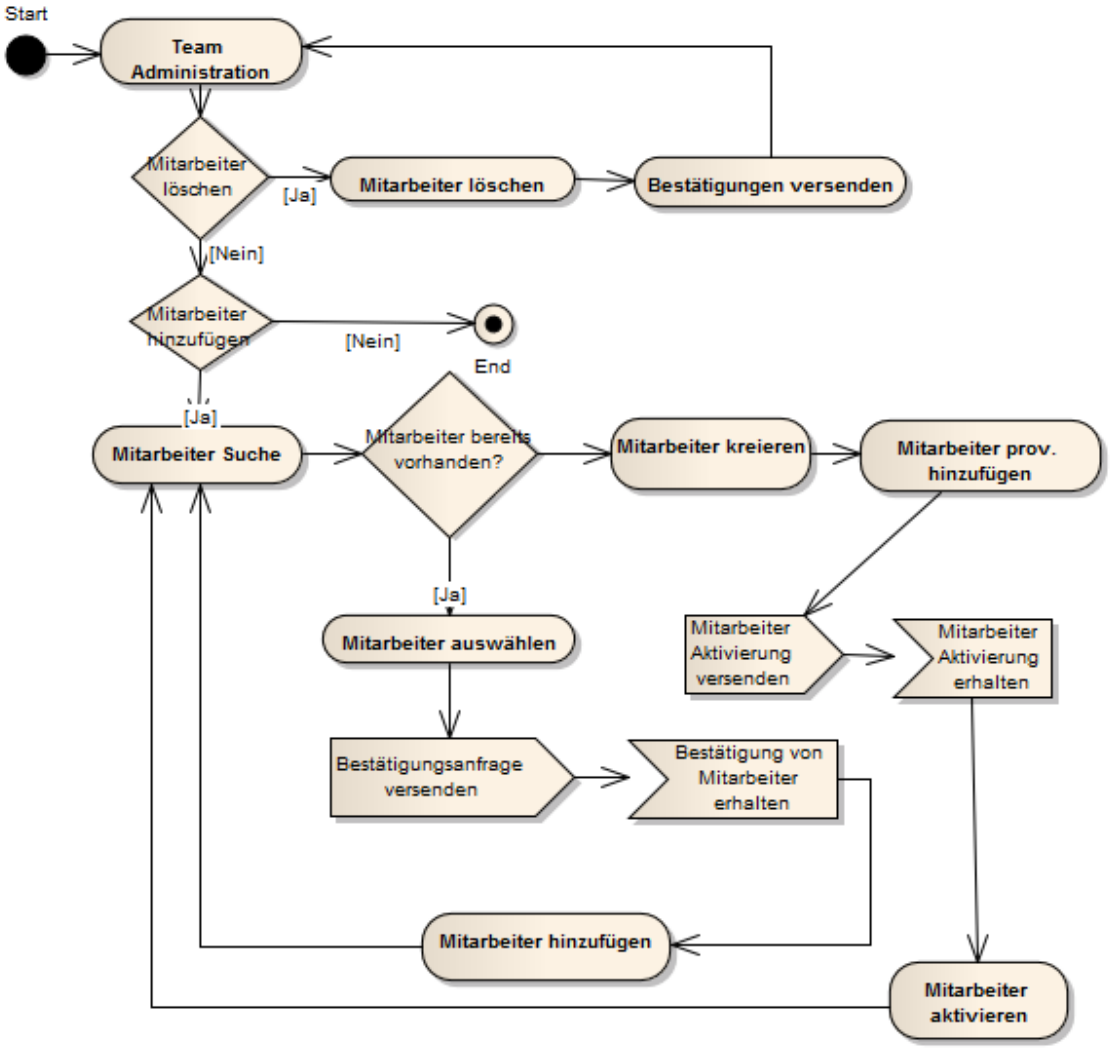


Abbildung 4.5: Prozess Member Administration Webpage

4.5 Navigations-Konzept

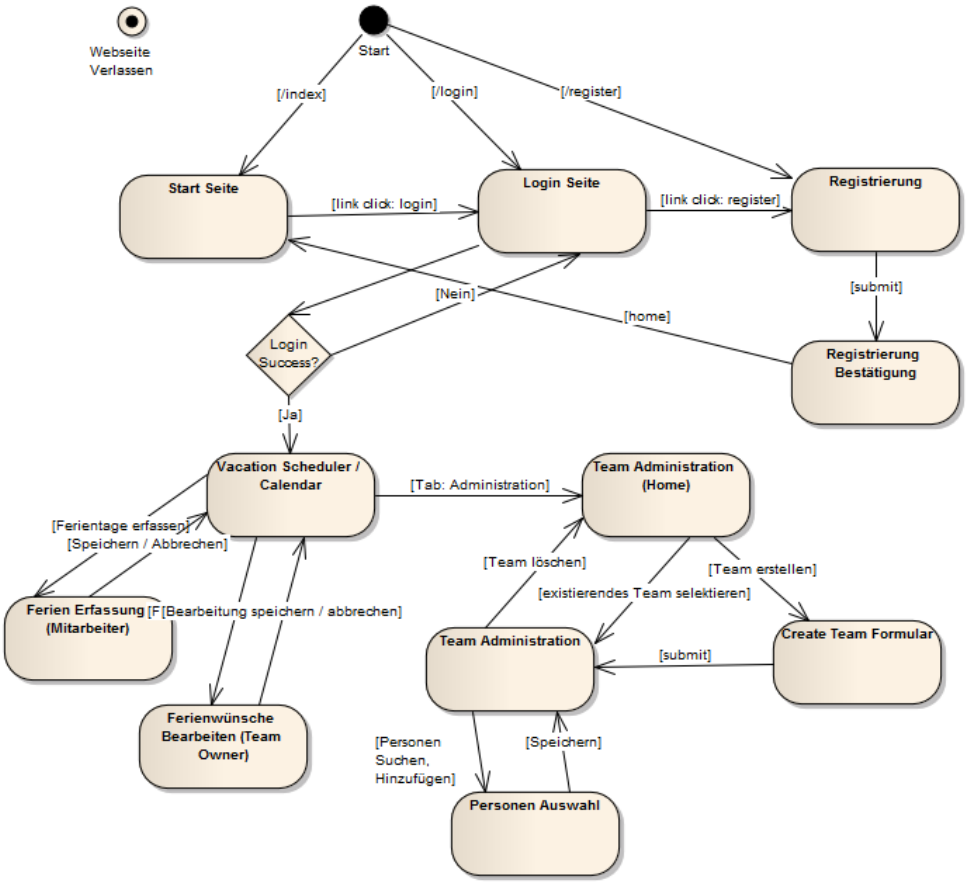


Abbildung 4.6: Navigation Webpage

4.6 Architektur

Kapitel 5

Implementation

5.1 Persistenz

5.1.1 Auswahl Persistenz-Provider

Bereits zu Beginn stellte sich die Frage, warum man welche der drei möglichen und im Abschnitt "3.3.6 Persistenz" aufgelisteten Persistenz-Provider Mapper, Record oder ScalaJPA verwenden sollte.

Ich versuchte herauszufinden und auszuführen, wie weit Ansätze hinsichtlich Dokumentation, aktive Weiterentwicklung und Reifegrad der Implementation sind.

Aktive Weiterentwicklung

Ein Blick auf die verschiedenen Commit Histories der Persistenz Module Mapper[1] und Record[2] zeigt schnell auf, dass in beiden Frameworks aktiv nicht sonderlich viel geschieht. Änderungen im Bereich der Persistenz-Libraries finden momentan vorallem im Bereich der NoSQL Datenbanken statt.

Dokumentation

Während die Dokumentation von Mapper[3] noch einigermassen anspricht, kann man zum Mapping mit dem Record Framework ausser ganz wenige Beispiele in [6, p. 79 - 113] nicht viel auffinden.

[1] http://github.com/lift/lift/commits/master/framework/lift-persistence/lift-mapper
[2] http://github.com/lift/lift/commits/master/framework/lift-persistence/lift-record
[3] http://www.assembla.com/wiki/show/liftweb/Mapper

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 Relationen zwischen den Klassen sind relativ unflexibel und genügen höchstens, wenn man ein Projekt auf der "Grünen Wiese" starten kann. Der dritte wichtige 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.

5.1.2 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.

Klasse User

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

- **MegaBasicUser -** verfügt in Anlehnung an MegaProtoUser[4] über die Funktionalitäten, die im Zusammenhang mit der Registrierung, Login, Passwort-Reset benötigt werden.

- **Domain -** definiert abstrakte Methoden zur Umwandlung von Objekten in XML und in die JSON[5].

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

[5]Javascript Object 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 sicher gestellt und gleichzeitig ein lokaler Context für die Attachten Klassen definiert.

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

Listing 5.1: User: ScalaJPA Entity Definiton

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

annotiert werden. Mit Table kann man den Tabellennamen spezifizieren. 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

Listing 5.2: User: ScalaJPA Id mit Auto-Increment

```
1 @Id
2 //@GeneratedValue(
3 //       strategy = GenerationType.SEQUENCE,
4 //       generator="user_seq")
5 @GeneratedValue(strategy = GenerationType.AUTO)
6 @Column(name = "ID")
7 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 “5.1.3 Validation”). Die Annotation @BeanProperty sorg dafür, dass für ein Feld Getter- und Setter-Methoden erstellt werden. Dies ist vorallem zur Interoperabilität mit Java-Frameworks teilweise möglich - in diesem Fall wird es ebenfalls für die Validierung benötigt.

Listing 5.3: User: ScalaJPA firstname Mapping

```
1 @Column(name = "FIRST_NAME", nullable = false)
```

```
2 @NotNull
3 @NotEmpty
4 @BeanProperty
5 var firstname: String = _
```

Des weiteren sind die Annotationen @Embedded[6] und die verschiedenen Beziehungen zu anderen Tabellen @ManyToOne, @OneToMany und @ManyToMany[7] interessant.

5.1.3 Validation

Annotationen wie @NotNull, @Null, @Past, @Future sind definiert über den 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 Validierung wird vollumfänglich anhand der in den Klassen enthaltenen Annotationen gemacht. Als Rückgabewert werden die Constraints-Verletzungen innerhal eines Sets 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:

[6] @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

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

Abbildung 5.1: Formular Validierung: Registration

Die Implementation der Methode validate ist relativ einfach und sieht folgendermassen aus:

Listing 5.4: 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 }
```

5.2 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. Sofern man sich für JPA entscheidet, muss dies - insbesondere weil die Kopplung zwischen Domänenklassen und Mapper respektive Record so hoch ist - selber implementieren. In anlehnung an ProtoUser, MegaProtoUser und MetaMegaProtoUser vom Mapper-Framework habe ich Funktionalität übernommen und MegaBasicUser respektive MetaMegaBasicUser für das Mapping mit JPA implementiert. In

Sachen LOC[8] haben die Klassen noch etwas Optimierungspotential, in Sachen Registrierung, Login, Validierung, Passwort Reset, Benutzer Editierung tun sie aber ihren Zweck. Als kurze Erläuterung die Beschreibung zur Funktionalität für das Passwort-Reset beispielhaft:

Listing 5.5: Implementation Funktionalität Passwort-Reset

```
def passwordReset(id: String) =
    findById(id.toLong) match {
      case Full(user) =>
        def finishSet() {
          val violations = user.validate
          if (violations.size == 0)
            user.merge
          else {
            val node: NodeSeq = violations
            S.error(node)
            S.redirectTo("/")
          }
        }

        bind("user", HtmlTemplate.passwordResetXhtml,
          "pwd" ->
            password_*("", (p: List[String]) =>
              user.password = p.head),
          "submit" ->
            submit(S.??("set.password"), finishSet _))
      case _ =>
        S.error(S.??("password.link.invalid"));
        S.redirectTo(homePage)
    }
```

Die Methode zum Reset des Passworts wird vom Menu respektive der Lift-Internen SiteMap Funktionalität (siehe unter: "3.3.5 SiteMap [6, p. 61-70]") aufgerufen. Dies allerdings nur, wenn der Benutzer in der Http-Session 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

[8] Lines of Code

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 in etwa auf diese Weise implementiert.

5.3 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: "5.4.2 Ferienplanung"). Ausserhalb ist die Navigation mittels der Lift-Sitemap implementiert. Wie bereits unter "3.3.5 SiteMap [6, 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:

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 die sich in den Menu Instanzen befinden.

5.4 Flex Client

5.4.1 Architektur

Die meisten User-Interfaces zeichnen sich, insbesondere durch die hohe Kopplung zwischen Komponenten, durch eine extrem schlechte Wartbarkeit aus. Eine weitere Schwierigkeit bei diesen Applikationen sind die angemessene

und zentrale Behandlung von auftretenden Events, die Verfügbarkeit von Contexten und die Unterstützung in der Anbindung an Backend-Systemen. Damit solche Problemstellungen für diesen Client minimiert werden können, habe ich eine kurze Evaluation bestehender Flex-Frameworks durchgeführt und dabei die folgenden angeschaut:

- **Cairngorm[1] -** unter Flex 2 und 3 war Cairngorm noch ein Frameworks mit Fokus auf die MVC-Architektur von Client-Applikationen. 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 Dependency Injection und MVC Frameworks wie Parsley, Swiz, oder Spring ActionScript

- **Mate[2] -** 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 5.6: 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[5] -** 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 5.7: 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 5.8: 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 5.9: 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.

5.4.2 Ferienplanung

Evaluation Kalender

5.4.3 Teammanagement

5.5 Mail-Versand (Notifikationen)

Kapitel 6

Entwicklung, Build und Deployment

6.1 Build-System

Maven als Build-System basiert ebenfalls auf einem Plugin-Konzept und unterstützt in den verschiedensten Bereichen des Build-Managements. Nicht zuletzt dank dem Grundsatz "Convention over Configuration" lässt es sich auf allen Plattformen im Nu 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 Entwicklungsumgebungen Adapter oder Plugins dafür entwickelt haben vomit sich solche Projekte automatisch innerhalb der IDE konfigurieren. Dazu gehört das Hinzufügen von Abhängigkeiten in den Source- und Build-Pfad, das korrekte setzen von Ausgabe-Ordner.
- **Clean, Compile, Package:** Der gesamte Build-Prozess fürs Test-System findet mit Maven statt. Bei der Entwicklung werden die Klassen allerdings von der Entwicklungsumgebung kompilliert - diese beschriebenen Targets sind deshalb da nicht nötig.
- **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 unter-

schiedlichen Umgebungen laufen. Dann gilt es meistens, unterschiedliche Konfigurationen zu verwenden. Profile brauche ich um folgende Probleme zu lösen:

  – Der Datenbank-Zugriff wird in der Entwicklungsumgebung direkt über JDBC gemacht, in der Testumgebung auf der Stax-Cloud wird eine von Servern verfügbare 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. Die meisten 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[3] 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 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[4] 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.

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

```
<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.

Teil III

Rückblick

Kapitel 7

Analyse der Arbeit auf der Basis der Aufgabenstellung

7.1 Prototyp

7.2 Optionale Ziele

7.2.1 Setup von Test-und Produktiv-Umgebung

7.2.2 Performance Tests

7.2.3 Internationalization

7.2.4 Search Engine Optimization

7.2.5 Usability

Kapitel 8

Fazit

Literaturverzeichnis

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

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

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

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

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

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

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

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

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

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

[11] Wikipedia. Nosql — wikipedia, die freie enzyklopdie. http://de.wikipedia.org/w/index.php?title=NoSQL&oldid=79154375, 2010. [Online; Stand 11. Oktober 2010].

Listings

3.1 Sql Deklaration 23
3.2 Summe einer Liste in Java 24
3.3 Summe einer Liste in Scala 24
3.4 Typeinferenz in Java 25
3.5 Typeinferenz in Scala 27
3.6 Klassen und Traits definieren 27
3.7 Traits: Verschiedene Instanzen vom Typ IntQueue und die entsprechenden Auswirkungen 28
3.8 Traits: Deklaration Doubling-Incrementing-Queue 29
3.9 Partielle Anwendung einer Funktion 30
3.10 Implicit Conversions am Beispiel String 32
3.11 Implicit Conversions Method wrapString 32
3.12 Erstellung eines Lift-Projektes 33
3.13 Lift Template Surround 34
3.14 Lift Template Binding 35
3.15 Snippet 35
3.16 Views 35
3.17 Beispiel Mapping User Klasse mit Mapper 37
3.18 Property Mapping mit JPA 37
3.19 Relation Mapping mit JPA 38
3.20 Überschreibung der Locale-Berechnung 39
5.1 User: ScalaJPA Entity Definiton 53
5.2 User: ScalaJPA Id mit Auto-Increment 53
5.3 User: ScalaJPA firstname Mapping 53
5.4 Validation innerhalb der Klasse Persistence 55
5.5 Implementation Funktionalität Passwort-Reset 56
5.6 Event-Deklaration mit dem Mate Framework 58
5.7 Swiz: Bean Deklaration 59
5.8 Swiz: Bean Injection 59
5.9 Swiz: Event Observer 59

6.1 Konfiguration Maven mit Stax 63
6.2 Befehl Deployment Stax 63

Abbildungsverzeichnis

4.1 Use Case Diagramm 42
4.2 Entity Relationship Model 47
4.3 Prozess Member Administration Webpage 48
4.4 Prozess Ferien beantragen, planen 48
4.5 Prozess Member Administration Webpage 49
4.6 Navigation Webpage 50

5.1 Formular Validierung: Registration 55

Tabellenverzeichnis

3.1 Resultat der deklarativen Abfrage 23

5.1 Navigation und Menustruktur 57

A.1 Glossar 81

B.1 Journal Implementation Backend 83
B.2 Journal Implementation Frontend 84
B.3 Journal Dokumentation 85

Teil IV

Anhang

Anhang A

Glossar

Tabelle A.1: Glossar

Wort	Beschreibung
TODO	TODO

Anhang B

Journal

B.1 Phase Implementation Backend

Tabelle B.1: Journal Implementation Backend

Datum	Eintrag
7. August 2010	• Projekt Setup Client und Server • Initialer Commit ins Git Repository unter http://github.com/schmidic
13. August 2010	• Installation des Persistenz Providers (Hibernate und JPA)
14. August 2010	• Authentifizierung und Authorisierung via Basic Authentication • Implementation REST Support in für Browser, abfrage des X-HTTP-Method-Override Headers, da Browser nicht wirklich PUT und DELETE requests unterstützen.
16. August 2010	• Registrierung und Login Webservice
17. August 2010	• Erarbeiten des Entity Relationship Models • Mapping der Domain-Klassen auf das ERM via JPA

18. August 2010	• Webservices zur Administration der Teams und User • Laden von Fixtures mittels import.sql
20. August 2010	• Erweiterung der bestehenden Webservices
22. August 2010	• Webservice zur Administration der Ferien • Anpassung des Persistenz Mappings
27. August 2010	• Konfiguration von Maven-Profilen für das Deployment auf STAX[1]

B.2 Phase Implementation Frontend

Tabelle B.2: Journal Implementation Frontend

Datum	Eintrag
24. August 2010	• Evaluation unterschiedlicher Actionscript Frameworks (Mate[2], Swiz [3], Cairngorm[4]) für Dependency Injection und Event Handling. Entscheidung zugunsten Swiz augrund der folgenden Eigenschaften: Flexibilität, Leistungsfähigkeit (Context, TwoWay-Bindings, Injection, Event-Dispatching), Annotation-Support, Service Integration. • Initialer Commit ins Git Repository unter http://github.com/schmidic
26. August 2010	• Browser sendet 401 bei Nicht-Authorisierung - dies führt zu einem Browser-Popup für Benutzername und Passwort. Noch keine Lösung gefunden.

[1] http://stax.net
[2] http://mate.asfusion.com
[3] http://swizframework.org
[4] http://opensource.adobe.com/wiki/display/cairngorm/Cairngorm

27. August 2010	• Fertigstellung der Administrations-Ansicht • Integration des Schedulers von ILOG Elixier

B.3 Phase Dokumentation

Tabelle B.3: Journal Dokumentation

Datum	Eintrag
19. September 2010	Grundlagen
19. September 2010	Entity Relationship Model
20. September 2010	Grundlagen
21. September 2010	Grundlagen