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

Semesterarbeit

vorgelegt am: xx. November 2010

an der Hochschule für Technik in Zürich

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

Kapitel 1

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

Einleitung

TODO

Kapitel 2

Grundlagen

2.1 Funktionale Programmierung

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

2.1.1 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 2.1: Sql Deklaration

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

Tabelle 2.1: Resultat der deklarativen Abfrage

firstname	lastname	zip	city
Flavor	Flav	8000	Zürich

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

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. [1]

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

Listing 2.2: 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 2.3: Summe einer Liste in Scala

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

2.1.2 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 bergeben werden knnen.

2.2 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:

2.2.1 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 der Praxis

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

Listing 2.4: 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 }
```

Die Ermittlung des Rückgabetyps aufgrund des Variablen-Typs ist schon fast alles was Java in Sachen Typeinferenz zu bieten hat. Etwas mehr kann mit den höheren Sprachen wie Scala erreicht werden:

[2]Der Lambda-Kalkl 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

Listing 2.5: Typeinferenz in Scala

```
1 scala> val s = "inference"
2 s: java.lang.String = inference
3
4 scala> val l = List("a","b","c")
5 l: List[java.lang.String] = List(a, b, c)
```

2.2.2 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.

2.2.3 Nachteile von statischer Typisierung

- Dynamische Sprachen ermöglichen eine höhere Flexibilität. Zum Beispiel können folgende Dinge in statischen Sprachen nicht, mit erhöhtem Aufwand oder mit schlechtem Design gemacht werden:
  - Einfügen von Methoden in Classen oder Objekte zur Laufzeit.
  - Duck Typing[3]
- Kompilieraufwand ist wesentlich grösser.

2.2.4 Scala und die Typisierung

[3]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

Kapitel 3

Analyse

3.1 Programmiersprachen

3.1.1 Java

3.1.2 Groovy

3.1.3 Scala

3.2 Web-Frameworks

3.2.1 Grails

3.2.2 Liftweb

3.3 HTML vs. Flex vs. others

Kapitel 4

Evaluation

4.1 Programmiersprache

4.2 Web Framework

Kapitel 5

Implementation

5.1 Datenbank

5.1.1 Entity Relationship Model

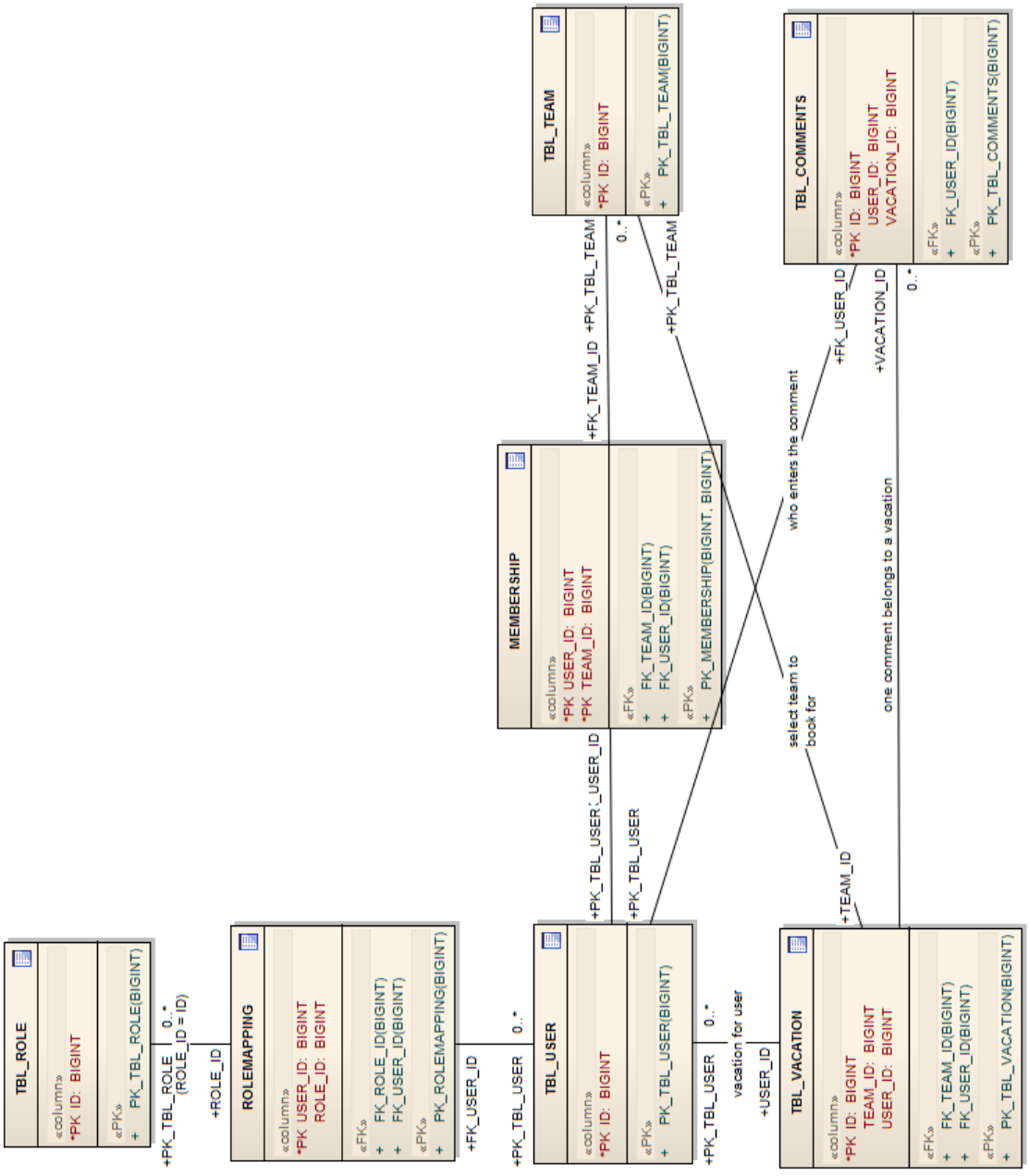

Abbildung 5.1: Entity Relationship Model

Kapitel 6

Fazit

Literaturverzeichnis

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

Abbildungsverzeichnis

Tabellenverzeichnis

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 aufgrund 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