

# *Webapplikation mit Scala/Lift*

---



---

## Autoren

Daniel Hobi ([dhobi@hsr.ch](mailto:dhobi@hsr.ch))

Ralf Muri ([rmuri@hsr.ch](mailto:rmuri@hsr.ch))

---

## Zeitraum

22. Februar 2010 bis 18. Juni 2010

---

## Auftraggeber

Paul Bernet ([paul.bernet@crealogix.com](mailto:paul.bernet@crealogix.com)), Crealogix E-Business AG

Prof. Hans Rudin ([hrudin@active.ch](mailto:hrudin@active.ch)), Hochschule Rapperswil

# Änderungsgeschichte

---

Die folgende Tabelle gibt eine Übersicht der Änderungsgeschichte des Dokuments.

Version	Datum	Änderung	Autor
0.1	12.03.10	Erstellung	dh
0.2	14.03.10	Zusammenführung der Berichte, Erstellung des Dokumentes Technologiestudium	dh
0.3	16.03.10	Erstellen der Kapitel Maven & Git	rm
0.4	17.03.10	Vervollständigen der Kapitel Maven & Git	rm
0.5	23.03.10	Struktur für Userverwaltung, HTML Formular binden und auswerten	dh
0.6	30.03.10	Userverwaltung Allgemein & ProtoUser	rm
0.7	08.04.10	Sitemap Funktionalitäten	rm
0.8	23.04.10	Lift Best Practice	rm
0.9	25.04.10	Persistenz	dh
1.0	02.05.10	Vorlage, Strukturierung des Dokuments	rm
1.1	12.05.10	GUI Widget, Strukturanpassungen	dh
1.2	17.05.10	Validierung	rm
1.3	18.05.10	Comet	dh
1.4	21.05.10	Ajax	rm
1.5	01.06.10	Überarbeitung gemäss Review von pernet und hrudin	dh, rm
1.6	01.06.10	Beispielapplikation	dh
1.7	02.06.10	Einführung Lift	rm
1.8	03.06.10	Einführung Scala	dh
1.9	05.06.10	Vervollständigung der Einführung in Scala, Beispielapplikation um Architekturdokumentation erweitert.	rm
2.0	08.06.10	URL Rewriting & Java Interoperabilität hinzugefügt	rm
2.1	08.06.10	Scala Frameworks, Properties & Internationalisierung	dh
2.2	10.06.10	Auswertung und Erfahrungsberichte	rm
2.3	11.06.10	Einleitungskapitel	dh
2.4	16.06.10	Überarbeitung und Abschluss des Dokuments	dh

# Inhalt

---

1	Abstract .....	5
2	Einleitung.....	6
2.1	Zweck des Dokuments.....	6
2.2	Aufbau des Dokuments .....	6
2.3	Zielpersonen / Leserkreis .....	6
3	Einführung in Scala .....	8
3.1	Was ist Scala? .....	8
3.2	Scala/Lift oder Lift/Scala?.....	8
4	Einführung in Lift .....	10
4.1	Grundkonzepte.....	10
4.2	View First im Detail.....	11
4.3	Erste Schritte mit Lift.....	12
4.4	Lift und andere Web Frameworks.....	17
4.5	Weiterführende Lift Ressourcen .....	18
5	Technische Aspekte .....	19
5.1	Infrastruktur .....	20
5.2	Userverwaltung .....	31
5.3	Persistenz .....	44
5.4	Validierung .....	49
5.5	GUI Widgets.....	55
5.6	Ajax & Comet.....	59
5.7	REST .....	69
6	Lift Best Practices und HowTos .....	74
6.1	Wichtige Klassen im Überblick .....	75
6.2	Objektübergabe zwischen Requests .....	77
6.3	Fortgeschrittenes Binding .....	80
6.4	URL Rewriting .....	83
6.5	Internationalisierung.....	86
6.6	Verwendung von Properties.....	89
6.7	Interoperabilität mit Java .....	90
7	Beispielapplikation .....	93
7.1	Architektur.....	94

7.2	Sicht des Benutzers .....	102
7.3	Sicht des Software Entwicklers.....	106
7.4	Sicht des Deployers .....	109
8	Auswertung und Erfahrungsberichte .....	112
8.1	Auswertung Technologiestudium .....	113
8.2	Persönliche Erfahrungsberichte .....	122
8.3	Schlusswort .....	128
9	Literaturverzeichnis.....	129

# 1 Abstract

---

Scala ist eine Programmiersprache, welche von der ETH Lausanne entwickelt wird. Aufgrund ihrer Kompatibilität mit Java und der Lauffähigkeit auf der Java Virtual Machine wird Scala als potenzieller Nachfolger von Java gehandelt. Das Lift Framework, welches auf Scala aufsetzt, wird in diesem Bericht auf die Praxistauglichkeit im Businessbereich untersucht. Da sich sowohl Scala als auch Lift in Entwicklung befinden und verfügbare Quellen im Vergleich zu anderen Technologien noch spärlich vorhanden sind, war es eine besondere Herausforderung die Praxistauglichkeit abschätzen zu können. Parallel zur Untersuchung der Sprachfeatures und neuartigen Ansätzen der Webprogrammierung wurde als Hilfestellung eine Beispielapplikation entwickelt. Mittels ausgesuchter, technischer Aspekte wie Benutzerverwaltung, Persistenzlayer, Validierung, GUI Widgets, Ajax & Comet und REST wurde die Eignung überprüft, indem diese Aspekte in einzelnen Feature Iteration theoretisch untersucht und anschliessend in die Beispielapplikation integriert wurden. Zusätzliche Best Practices und zahlreiche HowTos unterstützen die praktische Einarbeitung in das Lift Framework. Der vorliegende Bericht ist so strukturiert, dass sowohl technikinteressierte als auch langjährige Java Programmierer ihren Einstieg in die Scala/Lift Welt beschreiten können. Der Leser soll nach dem Studium dieses Berichts in der Lage sein, eine Technologieabschätzung zu tätigen und den Entscheid eines Einsatzes der Technologie Scala / Lift fällen zu können. Persönliche Erfahrungen, welche während der Arbeit von den Autoren gesammelt wurden, runden den Bericht ab.

## 2 Einleitung

---

Die folgenden Abschnitte wird der Zweck (Abschnitt 2.1) und der Aufbau (Abschnitt 2.2) des Dokuments erläutert. In Abschnitt 2.3 wird die optimale Handhabung des Dokuments durch den Leser anhand von Leserkreisen beschrieben.

### 2.1 Zweck des Dokuments

Dieses Dokument dient dem Know How Aufbau von Scala / Lift. Durch die detaillierte Auseinandersetzung mit verschiedenen Aspekten einer komplexeren Webapplikation erfüllt dieses Dokument die Anforderungen eines Proof of Concepts. Durch das Aufzeigen von verschiedenen HowTos und diversen Codebeispielen kann dazu der Reifegrad der Technologie eingeschätzt werden.

Leser aus verschiedensten Interessensgruppen sollen in der Lage sein, gezielt Informationen aus diesem Dokument zu erhalten, indem sie durch die hierfür aufbereitete Struktur durch das Dokument geführt werden.

### 2.2 Aufbau des Dokuments

Eine kurze Einführung in Scala / Lift und die ersten Schritte mit Lift befinden sich in den Kapiteln 3 und 4 dieses Dokumentes. Kapitel 5 listet die in Feature Iterationen behandelten technischen Aspekte auf. Im 6. Kapitel sind Lift Best Practices und HowTo's beschrieben. Die parallel zum Technologostudium entwickelte Beispielapplikation wird in Kapitel abgehandelt. Auswertungen der Arbeit und persönliche Erfahrungsberichte sind in Kapitel 8 zu finden.

### 2.3 Zielpersonen / Leserkreis

Das vorliegende Dokument wurde für drei unterschiedliche Leserkreise bzw. Zielpersonen aufbereitet und strukturiert. Nachfolgend werden diese Zielpersonen genauer beschrieben.

Die Zielperson A möchte sich einen Überblick über die Technologie Scala / Lift verschaffen und ist nicht an technischen Details interessiert. Für diese Person schlagen wir vor, nach Abschnitt 2.3.1 vorzugehen.

Die Zielperson B möchte im Vergleich zu Zielperson A mehr über Lift erfahren, indem sie detaillierte Einführungen und Codebeispiele erhält. Best Practices und Lift HowTos sind ebenfalls von grossem Interesse. Ein Vorgehen nach Abschnitt 2.3.2 ist für diese Person empfehlenswert.

Die letztere Zielperson C will sich intensiv ins Lift Webframework einarbeiten und somit tiefer in die Materie eindringen als Zielperson B. Die detaillierte Auseinandersetzung mit technischen Aspekten interessiert die Zielperson C ebenso

wie das Studieren von Erfahrungen. Zielperson C sei der Ablauf von Abschnitt 2.3.3 ans Herz gelegt.

### 2.3.1 Überblick Scala / Lift

Abschnitt 3.1 beschreibt, was Scala ist und zählt dazu einige Besonderheiten der Sprache auf. Grundkonzepte von Lift befinden sich im Abschnitt 4.1. In Abschnitt 7.2 wird die Beispielapplikation aus der Benutzersicht aufgezeigt. Ein Fazit der Arbeit befindet sich in Abschnitt 8.3.

### 2.3.2 Einführung in Lift

Kapitel 3 und 4 sollten vorerst eingehend studiert werden, bevor zu Best Practices und HowTos in Kapitel 6 übergegangen wird. Abschnitt 7.3 zeigt die Beispielapplikation aus der Sicht eines Softwareentwicklers.

### 2.3.3 Vertiefung einzelner Aspekte

Nebst einer eingehend Studie von der Kapitel 3 und 4 wird empfohlen, nahtlos auf Kapitel 5 weiterzugehen, welches technische Aspekte detailliert beschreibt. Kapitel 6 kann als allgemeines Lift Nachschlagewerk benutzt werden. Die Beispielapplikation in Kapitel 7 zeigt die Liftentwicklung aus verschiedenen Perspektiven, in welche sich der Leser versetzen kann. Die Auswertung des Technologiestudiums in Abschnitt 8 bringt hilfreiche Tipps für das eigene Studium mit ein.

## 3 Einführung in Scala

---

Scala ist eine funktionale und objektorientierte Programmiersprache, welche seit 2001 an der École polytechnique fédéral de Lausanne unter der Leitung von Martin Odersky entwickelt wird. Der erste Abschnitt dieses Kapitels bietet eine kurze Einführung in Scala und enthält zahlreiche Verweise auf externe Quellen. Im zweiten Abschnitt wird auf die Frage eingegangen, inwieweit sich ein Lift Entwickler in Scala einarbeiten sollte. Zu diesem Zweck fliessen die Erfahrungen der Autoren dieses Berichts mit ein.

### 3.1 Was ist Scala?

---

Der Name Scala leitet sich aus „scalable language“ ab und bezeichnet somit die hohe Skalierbarkeit der Sprache. Eine gute Einführung dazu liefert Martin Odersky gleich selber im Abschnitt „What makes Scala scalable?“ im Buch „Programming in Scala“ (1 S. 45).

Scala kann durch den mitgelieferten Compiler sowohl zu Java Byte Code als auch zu Common Intermediate Language kompiliert werden und ist somit auf der Java Virtual Machine von Sun bzw. der Common Language Runtime von Microsoft lauffähig. Im Falle der Java Virtual Machine ist Scala vollständig kompatibel zu Java. Näheres dazu findet man im Abschnitt „Why Scala?“ im Buch „Programming in Scala“ (1 S. 48).

Zweifellos wurde Scala entwickelt um den Möglichkeiten der heutigen Computergeneration gerecht zu werden. Der hohe Grad der Parallelität, welcher durch immer mehr CPU Kerne laufend ansteigen wird, verlangt nach ausgeklügelten Programmmechanismen. Mit der funktionalen Programmierung bietet hier Scala eine gute Antwort. Mehr dazu befindet sich im Buch „Programming in Scala“ (1 S. 46-48).

Die Programmiersprache Scala wartet zudem mit einigen Erneuerungen im Umgang mit immer wiederkehrenden Problemen auf. Ein Beispiel davon ist das in Java notwendige Testing gegen „null“. Ein sehr zu empfehlender Abschnitt über die „Best Practices“ von Scala bietet David Pollak in seinem Buch „Beginning Scala“ (2 S. 274-279).

Im Internet wartet Scala mit einer ansehnlichen Präsenz auf. Die neuste Scala Version ist auf der offiziellen Scala Webseite (3) verfügbar. Mailinglisten, Blogs über Scala, Wikis und Foren sind auf der Scala Website (4) übersichtlich aufgelistet. Der Scala Trac Server (5) zeigt aktuelle Bugs und Verbesserungswünsche auf.

### 3.2 Scala/Lift oder Lift/Scala?

---

Bei einem Hausbau tut man sich gut daran, zuerst ein stabiles Fundament zu errichten, bevor mit dem eigentlichen Hausbau begonnen wird. Ob dies im Falle Scala / Lift ebenfalls zutrifft, kann nicht vollends erörtert werden. Fakt ist, dass Lift

zahlreiche Features von Scala verwendet. Durch die Bytecodekompatibilität zu Java besteht allerdings die Gefahr, dass Scala in Java-Syntax geschrieben wird, was wiederum unnötig viel und aufgeblasener Code zur Folge hat. Als Java-Entwickler sei einem die Sammlung „Java to Scala“ (6) auf der Scala Website ans Herz gelegt. Es empfiehlt sich daher sich einen guten Überblick über Scala zu verschaffen, ehe die ersten Zeilen einer Lift Anwendung geschrieben werden. Tabelle 1 gibt einen Überblick über grundlegende Scala Features.

Feature	Begründung
<b>Scala Syntax</b>	Scala markiert zahlreiche Komponenten als optional, wie z.B. das return Statement oder das Semikolon, mit dem Ziel den Code lesbarer zu gestalten. Als Lift Entwickler sollte man sich an diese Richtlinien halten.
<b>Class, Object, Trait, Methoden</b>	Notwendig für das grundsätzliche Verständnis von Scala.
<b>Funktionale Programmierung</b>	Lift Methoden nutzen oft die Features der funktionalen Programmierung (z.B. bind()). Eine detaillierte Auseinandersetzung hilft, die Abläufe besser zu verstehen.
<b>Pattern Matching</b>	Ein sehr mächtiges Feature, welches in einer Liftapplikation immer wieder angetroffen wird. Eine sinnvolle Nutzung wird jedem ans Herz gelegt.
<b>Option Type</b>	Lift verwendet einen ähnlichen Ansatz mit dem Type „Box“. Ein Studium des „Option“ Types hilft daher im Verständnis des Konzeptes.

Tabelle 1: Scala, Wichtige Features

Erfahrungsgemäss ist ein vorangehendes Studium eines Scala Buches sehr ratsam. Als guter Einstieg und damit als Empfehlung geben wir hier das Buch von David Pollak „Beginning Scala“ (2) an. David Pollak ist ebenfalls der Vater des Lift Frameworks. Wer es detaillierter mag, ist mit dem Buch „Programming in Scala“ (1) sehr gut bedient.

Abschliessend sei an dieser Stelle gesagt, dass die Einstiegshürde zu Scala / Lift recht hoch angesetzt ist. Gewisse Kapitel müssen teils mehrmals studiert werden, um die zum Teil neuen Konzepte zu verstehen. Dies gilt übrigens auch für Lift Code, welcher zu Beginn ein wenig „magisch“ wirken kann. Ein Durchhaltevermögen zahlt sich aber vollends aus, weil die korrekte Anwendung der neuen Konzepte in dichterem Code resultiert.

## 4 Einführung in Lift

---

Lift ist ein modernes, in Scala geschriebenes, Web Framework. Bei der Entwicklung von Lift wurde versucht, die Vorteile und Stärken von bestehenden Web Frameworks mit neuen Ansätzen zu kombinieren, und so ein ausdruckstarkes und elegantes Web Framework zu kreieren. Lift legt besonderen Wert auf Aspekte wie Sicherheit, Wartbarkeit, Skalierbarkeit und Leistungsfähigkeit. Ein weiterer Schwerpunkt ist die hohe Produktivität bei der Entwicklung in Lift.

Da Lift in Scala geschrieben ist sind Lift Web Applikationen vollständig auf der JVM lauffähig. Scala als moderne, funktionale und objektorientierte Programmiersprache bietet mit seinen neuartigen Features eine optimale Grundlage für Lift.

### 4.1 Grundkonzepte

---

Dieser Abschnitt beschreibt einige der Grundkonzepte von Lift. Einige dieser Ansätze zeichnen Lift, im Gegensatz zu anderen Webframeworks, besonders aus.

- **HTTP Abstraktion**

Lift verfügt eine sehr starke Abstraktion des http Request und Response Lifecycle. Lift nimmt dem Entwickler diesbezüglich viel Arbeit ab. Der Entwickler muss sich keine grossen Gedanken über Request, Response, Gültigkeit und Zustand machen. Die Abstraktion ist auf einer viel höheren Ebene angesetzt, als das bei den meisten heute gebräuchlichen Frameworks üblich ist.

- **View First**

Lift Web Applikationen werden nach dem „View First“ Ansatz entwickelt. Dieses Design Pattern ist eine Alternative zum in heutigen Web Frameworks weit verbreiteten Model-View-Controller (MVC) Typ II (Model 2) Pattern. In Web Applikationen ist die View die erste Komponente welche bei einem Zugriff aufgerufen wird, daher der Name View First. In einem späteren Abschnitt dieses Kapitel wird dieses Pattern noch genauer erläutert.

- **Integration in Web Technologien**

Lift verfügt über eine sehr gute Integration von verschiedenen Web Technologien. So existiert ein Abstraktionslayer für JavaScript, so dass der Entwickler kein eigentliches JavaScript mehr schreiben muss. Weiter bietet Lift sehr ausgereifte und komfortable Unterstützung für Technologien wie Ajax, Comet, REST sowie Schnittstellen zu Diensten wie OpenID und Paypal um nur einige zu nennen.

- **Unterstützung für die Web Applikationsentwicklung**

Das Lift Framework bietet Bibliothek zur Unterstützung für die Entwicklung von Webapplikationen. Dazu gehört ein eigener OR Mapper sowie Unterstützung für typische Web Applikationsprobleme wie Authentifizierung, Internationalisierung, Logging und weitere.

Natürlich gibt es noch weitere Ansätze oder Konzepte in Lift welche von Bedeutung sind. Der Übersicht halber wurden hier nur Konzepte beschrieben, welche in der Beispielapplikation umgesetzt wurden.

## 4.2 View First im Detail

Das View First ist ein wichtiges Prinzip des Lift Web Framework. In diesem Abschnitt wird dieses detaillierter behandelt und einige Kernpunkte des Ansatzes hervorgehoben. Das Thema View First wird in der Vorabversion des Buches „Lift in Action“ (7 S. 9) ausführlich behandelt.

### 4.2.1 Einordnung

Das View First als Design Pattern kann als Alternative zum Model-View-Controller (MVC) angesehen werden. Anfragen einer Webapplikation kommen über die View. Diese steht an erster Stelle (View First). Man kann sich diesen Ansatz als modifizierter Model-View-Presenter (MVP) vorstellen.

### 4.2.2 View First im Modell

Bei der Betrachtung des View First Prinzip als Pattern kann das View-ViewModel-Model (V-VM-M) Modell verwendet werden. Die folgende Grafik aus dem Buch „Lift in Action“ (7 S. 10) stellt das View First Prinzip als V-VM-M Pattern dar. Die einzelnen Komponenten werden zusammen mit dem Ereignisfluss des Models dargestellt.

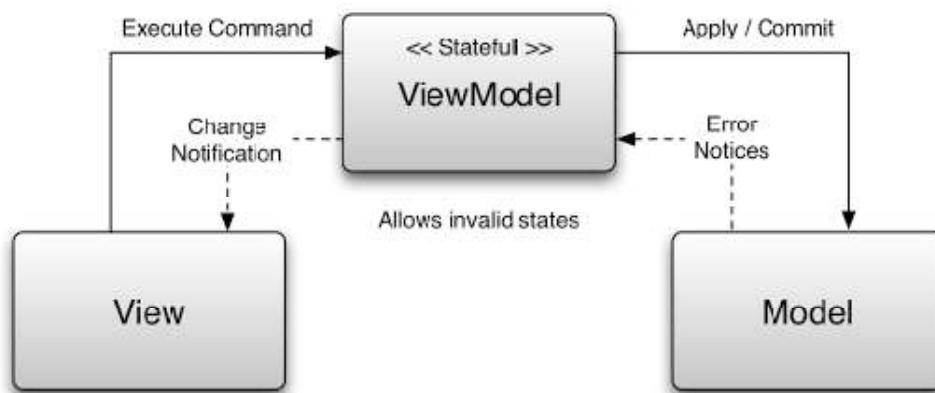


Abbildung 1: View First als View-ViewModel-Model visualisiert

Im Folgenden werden die einzelnen Komponenten des Modells mit ihrer jeweiligen Aufgabe beschrieben.

- **View**

Als View gelten die XML (XHTML) basierten Templates welche den Markup Code für das eigentliche Benutzerinterface der Web Applikation enthalten. Mit Lift ist es nicht möglich in diesen Dateien Programmlogik zu platzieren. Ebenfalls als View Komponenten gelten in Scala Klassen eingebundener XML Markup Code, welcher zur Laufzeit gerendert wird. In beiden Fällen zwingt Lift den Entwickler wohlgeformten XML Code zu schreiben.

- **ViewModel**

Das ViewModel wird im Lift Jargon als Snippet bezeichnet. Diese Scala Klassen sind für die dynamische Generierung von Inhalt und die Kommunikation zwischen View und Model verantwortlich. Snippets sind nicht als Controller im Sinne von MVC gedacht. Sie sollten keine Verantwortlichkeiten bezüglich Kontrollflusssteuerung oder ähnlich Controller Funktionen übernehmen. Aus einer View können mehrere Snippets aufgerufen werden. Jedes Snippet ist für eine Aufgabe zuständig.

- **Model**

Das Model ist eine Komponente welche durch verschiedene Instanzen repräsentiert werden könnte. Oft ist ein Model eine Scala Klasse aus dem Problem Domain Layer oder eine Entity Klasse aus dem Persistence Layer. Das Model ist für die Ausführung von Businesslogik gemäss den Anfragen des ViewModel zuständig.

Das View First Pattern wurde durch Lift eingeführt und kommt den Anforderungen und Anwendungen von modernen (interaktiv, single page) Web Applikationen nach. Bei traditionellen MVC Typ II Applikationen (z.B. Struts) gab es nur immer einen Controller, welche alle Aufgaben übernehmen musste.

Das View First Prinzip bietet somit einen neuartigen Ansatz für die Entwicklung von Web Applikationen. Wie dieses Prinzip in der Praxis umsetzt wird, erläutert der folgende Abschnitt anhand von exemplarischen Beispielen.

## 4.3 Erste Schritte mit Lift

Dieses Kapitel dient als eine kleine Einführung in die Entwicklung mit Lift. Als Anschauungsmaterial dient ein einfaches Beispiel einer Webapplikation. Dieses Beispiel wird nicht Schritt für Schritt, sondern nur überblickshalber durchgearbeitet. Das Kapitel soll als eine kleine Einführung, in wie die Entwicklung in Lift aussehen könnte, dienen.

Wenn man sich in eine neue Technologie einarbeitet, schaut man gerne Codebeispiele und kleinere Applikationen an. Dieses Kapitel soll auch als Grundlage für das bessere Verständnis von Codebeispielen und deren Hintergründe dienen.

### 4.3.1 Übersicht der Lift Komponenten

Lift Web Applikationen werden typischerweise aus einer Auswahl von verschiedenen Komponenten erstellt. Einige Begriffe oder Bezeichnungen von Komponenten unterscheiden sich von anderen Web Frameworks. Tabelle 2: Lift Komponenten listet verschiedene Komponenten und deren Rolle in einer Lift Applikation auf. Diese Tabelle soll das Verständnis der Struktur von Lift Applikationen fördern.

Komponenten Bezeichnung	Zweck, Rolle
<b>Template</b>	Ist ein XML File welches Lift spezifische Tags enthält. Es wird für das Rendering von Inhalten verwendet. Typischerweise sind Templates XHTML Seiten mit Lift Tags welche für das Einbinden von dynamischem Inhalt verwendet werden.
<b>View</b>	Wird für das Rendering von Inhalten über Scala Code verwendet. Es ist eine Funktion innerhalb einer Scala Klasse, welche ein XML Baum als Rückgabewert hat und über einen dispatch Mechanismus angesteuert wird. Views bieten die Möglichkeit Antworten auf Anfragen komplett in Scala Code zu schreiben.
<b>Snippet</b>	Sind Scala Klassen welche für die Bereitstellung von dynamischem Inhalt verwendet werden. Eine Snippet Klasse enthält eine oder mehrere Methoden welche ein XML Baum als Parameter entgegennehmen und auch wieder zurückgeben.
<b>Lift Tags</b>	Sind XML Tags welche mit <lift: ... /> beginnen. Sie werden im Markup Code verwendet (Templates oder Views) um diese mit dynamischen Inhalt anzureichern.

Tabelle 2: Lift Komponenten

### 4.3.2 Integration in Webcontainer

Lift Applikationen werden in einem Java Webcontainer (Tomcat, Jetty) betrieben. Das Lift Framework wird als Servlet-Filter Instanz in den Webcontainer eingebunden. Gegenüber der Integration als Servlet hat ein Servlet-Filter den Vorteil, dass Requests welche der Lift Servlet-Filter nicht behandelt, vom Container behandelt werden (z.B. statischer Inhalt). Die web.xml Datei des Containers muss entsprechend auf die LiftFilter Klasse konfiguriert werden. Das folgende Codelisting 1 wurde aus dem Definitive Guide to Lift (8) entnommen und zeigt eine für den Lift Filter konfigurierte web.xml.

---

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  ... DTD here ...
<web-app>
<filter>
  <filter-name>LiftFilter</filter-name>
  <display-name>Lift Filter</display-name>
  <description>The Filter that intercepts lift calls</description>
  <filter-class>net.liftweb.http.LiftFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>LiftFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>
```

---

Codelisting 1: web.xml für den Lift Filter konfiguriert

### 4.3.3 Bootstrapping mit Lift

Initialisierungs- und Konfigurierungseinstellungen einer Lift-Applikation werden in einer sogenannten Boot Klasse durchgeführt. Diese Boot Klasse muss von net.liftweb.http.Bootable ableiten. Standardmäßig (nach Projektvorlagen) ist dies die Klasse bootstrap.liftweb.Boot. Das folgende Codelisting 2 zeigt die Boot Klasse aus einer Standard Vorlage.

---

```
package bootstrap.liftweb
class Boot {
  def boot {
    // where to search snippet
    LiftRules.addToPackages("SimpleLift")

    // Build SiteMap
    val entries = Menu(Loc("Home", List("index"), "Home")) :: Nil
    LiftRules.setSiteMap(SiteMap(entries:_*))
  }
}
```

---

Codelisting 2: bootstrapping.liftweb.Boot Klasse

Je nach Web Applikation werden unterschiedliche Einstellungen in dieser Boot Klasse konfiguriert. Die meisten Einstellungen werden über das net.liftweb.http.LiftRules Objekt definiert. In der folgenden Liste werden einige Beispiele von Konfigurationsmöglichkeiten aufgezeigt.

- Wo Lift nach Snippets und Ressourcen suchen soll
- Auf welche Funktionalität gewisse URLs dispatched werden sollen
- Aufsetzen des Sitemap (Navigations- und Zugriffskontrolle von Lift)
- URL Rewritings
- Initialisierung von GUI Widgets
- Ressource Handling
- Einstellungen für Lokalisierung und Charakterencoding

Weiterführende Informationen zum Thema Bootstrapping können im „Definitive Guide to Lift“ (8 S. 26) entnommen werden.

### 4.3.4 Dynamischer Inhalt mit Snippets

Snippets (oder View-Models) werden in Lift zur Aufbereitung von dynamischem Inhalt verwendet. Technisch gesehen ist ein Snippet eine Funktion, welche einen XML Block entgegennimmt und auch wieder zurückgibt. Die Snippets werden über Lift Tags in den Markup Code eingebunden. Das folgende Beispiel ist ein einfaches Hello World mit der Ausgabe der aktuellen Zeit. Der Markup Code für das Template wird in Codelisting 3 dargestellt.

---

```
<lift:surround with="default" at="content">
  <h2>Welcome to your project!</h2>
  <p>
    <lift:helloWorld.howdy>
      <span>Welcome to SimpleLift at <b><:time/></b></span>
    </lift:helloWorld.howdy>
  </p>
</lift:surround>
```

---

Codelisting 3: Hello World Markup Code

Alle Tags welche mit `<lift:.../>` beginnen sind Lift Tags. Es gibt ein Set von vordefinierten Tags für verschiedene Zwecke. Diese sind jeweils auch als Snippets implementiert. Eigene Snippets werden auf die gleiche Weise eingebunden. In diesem Beispiel ist dies das HelloWorld Snippet. Das Tag `<lift:helloWorld.howdy/>` hat ein Aufruf der Methode `howdy` der Klasse `HelloWorld` zur Folge. Als Parameter wird der Inhalt des Tags übergeben. Codelisting 4 zeigt den dazugehörigen Snippet Code.

---

```
class HelloWorld {
  def howdy(in: NodeSeq): NodeSeq =
    Helpers.bind("b", in,
      "time" -> (new _root_.java.util.Date).toString)
}
```

---

Codelisting 4: Hello World Snippet Code

Die Klasse `HelloWorld` hat eine Methode `howdy`. Der Typ des Eingangsparameters `NodeSeq`, ist eine Klasse für die Repräsentation eines XML Baumes. Die Methode bindet das Tag `<b><:time/></b>` auf die String Repräsentation der aktuellen Zeit. Das Binding ist ein wichtiger Aspekt in Lift. Es ist die Grundlage für die Umsetzung des View First Patterns und die Bereitstellung von dynamischem Inhalt.

### 4.3.5 HTML Formular binden und auswerten

Dieser Abschnitt zeigt an einem einfachen Beispiel die Erstellung eines eigenen HTML Formulars mit anschliessender Auswertung der gesendeten Variablen in der zugehörigen Snippetklasse. Dieses Beispiel ist etwas fortgeschritten als das Hello World aus dem letzten Kapitel. Einige Punkte werden dabei etwas genauer erläutert.

## Formular

Codelisting 5 zeigt ein einfaches, vollständiges Formular welches in gültigem XHTML geschrieben ist.

---

```
<lift:Login.login form="POST">
  <f:username id="username_id"/>
  <f:password id="password_id"/>
  <f:submit/>
</lift:Login.login>
```

Codelisting 5: Markup Code für HTML Login Formular

Dies ist die typische Struktur von Markup Code für das Einbinden eines Lift Snippets. Das Tag `<lift:Login.login/>` verweist auf die Snippet methode login der Klasse Login. Das Attribut form wird von Lift ausgewertet und definiert dass automatisch ein `<form/>` Tag mit entsprechendem action Attribut generiert werden soll.

Die f-Tags sind an dieser Stelle nur Platzhalter. Durch was diese schlussentlich ersetzt werden bzw. wie diese verarbeitet werden wird in der Methode login der Snippetklasse Login definiert.

### *Snippet*

Das folgende Codelisting 6 zeigt die Snippetklasse Login, die bei einem Abschicken des Formulars die Variablen entgegen nimmt und zwischenspeichert.

---

```
import _root_.net.liftweb.util._
import Helpers._ // für bind()
import xml.NodeSeq
import net.liftweb.http._ // für SHtml

object userName extends RequestVar[String]("")
object password extends RequestVar[String]("")
object isLoggedIn extends SessionVar[Boolean](false)

class Login {
  def login(xhtml: NodeSeq): NodeSeq = {
    bind("f", xhtml,
      "username" -> SHtml.text("", userName.set(_)),
      "password" -> SHtml.text("", password.set(_)),
      "submit" -> SHtml.submit("Login", () => {
        isLoggedIn.set(true)
      })
    )
  }
}
```

Codelisting 6: Snippet Code für simples Login

Die Methode login wird vom Liftframework mit einem Objekt des Types NodeSeq aufgerufen und erwartet wiederum einen NodeSeq als Rückgabewert. Das als Parameter angegebene NodeSeq Objekt enthält den entsprechenden Inhalt des Tags als XML-Baum. Dieses Objekt wird nun anhand der bind – Methode geparsed. Die Elemente `<f:username />` und `<f:password />` werden durch HTML – Inputfelder ersetzt. Gleichzeitig wird angegeben, welche Funktion aufgerufen werden soll. Bei einem Klick auf den Button (`SHtml.submit()`) wird die Form ein weiteres Mal

gerendert und die Funktion isLoggedIn.set(true) ausgeführt. In Lift ist es möglich Scala Funktionen an Formular- Elemente oder Handler zu binden. Hier wir eine Scala Funktion an den Form Post Event gebunden.

Es wurde nicht auf alle Details des Beispielcodes eingegangen. Dieses Beispiel sollte nur die prinzipielle Funktionsweise von Lift und des View First Patterns aufzeigen.

Das in diesem Kapitel betrachtete Binding ist in Lift ein sehr mächtiges und auch sehr wichtiges Werkzeug. Weiterführende Informationen zum Thema binding können im „Definitive Guide to Lift“ (8 S. 35) oder auf dem Lift Wiki (9) gefunden werden.

## 4.4 Lift und andere Web Frameworks

---

Dieser Abschnitt stellt eine kleine Auswahl von anderen Scala Webframeworks vor. Auf eine kleine Einführung der Frameworks folgt jeweils ein Fazit in Form eines Vergleiches zu Lift. Das Fazit widerspiegelt dabei lediglich den ersten Eindruck des Frameworks und bezieht sich auf den deren aktuellen Entwicklungsstand (07.06.2010).

### 4.4.1 Step

Step ist ein kleines Webframework, welches von Sinatra, eine in Ruby geschriebene DSL für schnelle Webapplikationsentwicklung, inspiriert wurde. Bis auf die Syntaxunterschiede sehen die beiden Webframeworks codetechnisch ähnlich aus.

Step wird ebenfalls auf [github.com](#) entwickelt. Die Einstiegsseite (10) auf [github.com](#) liefert eine gute, nahezu vollständige Übersicht über die Features von Step. Der http Zyklus ist in den Beispielen gut zu erkennen.

Hervorzuheben ist die Integration von Scalate, einer in Scala geschriebenen Templating Engine, welches zum Ziel hat, eine ähnlich wichtige Rolle wie JSP in Java Webapplikationen einzunehmen.

#### Fazit

Im Vergleich zu Lift wartet Step nicht mit neuartigen Webansätzen auf. Step ist darauf spezialisiert, klein und einfach zu sein. Dabei bietet Step lediglich die Grundfunktionalitäten der Webentwicklung an. Ein Entwickler kann somit gleich mit der Implementierung einer einfachen Webapplikation beginnen. Ajax & Comet Support sucht man allerdings vergebens.

### 4.4.2 pinky

pinky bezeichnet sich als REST/MVC Framework (11), welches auf Guice aufsetzt. Guice wird in pinky vor allem für die aspektorientierte Programmierung und Dependency Injection verwendet. Nebst den neuen Möglichkeiten, welche vor allem durch Scala eröffnet werden, greift pinky auch gerne auf die altbewährten Filter und Servlets zurück, um die Lernkurve für Neueinsteiger tief zu halten.

## Fazit

Pinky bietet eine Mischung aus neu und alt. Erstaunlicherweise wartet pinky trotz zahlreicher Features nicht mit einer eigenen Templating Engine auf und überlässt es somit dem Entwickler ob und welche er verwenden will.

### 4.4.3 Sweet

Sweet steht für Simplicity, Works, Efficient, Extensible und Testable (12). Als Architektur wird ebenfalls das MVC Pattern eingesetzt. Sweet wrapped dabei die altbewährten javax.servlet Klassen in Scala Klassen um einfacher und in Scala Syntax auf Parameter und Attribute zugreifen zu können.

Als Templating Engine greift Sweet auf die Java Bibliothek FreeMarker zurück.

## Fazit

Mit Sweet erhält man eine altbewährte Technologie in neuer Syntax. Die zu schreibenden Controller beschränken sich darauf, Parameter entgegenzunehmen, auszuwerten um darauf eine View zurückzugeben.

## 4.5 Weiterführende Lift Ressourcen

---

Eine gute und umfangreiche Übersicht von Lift bietet das Einführungskapitel der Vorabversion von „Lift in Action“ (7 S. 5). Das offizielle Lift Wiki (9) bietet einige Artikel zu populären Themen und auch einiges an Beispielcode als Ausgangslage für die weitere Entwicklung. Wenn man sich einmal eine Übersicht gemacht und die Grundlagen erarbeitet hat, kann der „Definitive Guide to Lift“ (8) als ein gutes Nachschlagewerk verwendet werden. Eine Vielzahl von Informationen können auch im Archiv der Lift Group (13) gefunden werden. Dort wird man ebenfalls über das aktuelle Geschehen informiert und findet kompetente Hilfe bei Problemen.

## 5 Technische Aspekte

---

Um die Praxistauglichkeit und den Reifegrad des Lift Web Framework zu erforschen, sowie Erfahrungen mit der Entwicklung mit Lift zu sammeln, wurden im Vorfeld einige technische Aspekte definiert, welche den Kern des Technologiestudiums bilden.

Diese technischen Aspekte wurden anhand konkreter Praxisanforderungen zusammengestellt. Die Aspekte wurden in einem separatem Dokument (14), welches als eine Art Anforderungskatalog dient, beschrieben. Die Ergebnisse des Technologiestudiums gemäss den definierten Aspekten sind in diesem Kapitel festgehalten.

Für jeden technischen Aspekt gibt es einen entsprechenden Abschnitt in diesem Kapitel. Die Aspekte unterscheiden sich teilweise in ihrer Granularität. Einzelne Aspekte sind eher Allgemein gehalten und geben eine Übersicht über das entsprechende Thema. Andere Aspekte sind, aufgrund ihrer Natur, viel technischer und befassen sich tief mit Eigenschaften des Lift Web Framework.

Der erste Abschnitt 5.1 befasst sich mit der Infrastruktur rund um eine Lift Entwicklung und ist für jede Art von Lift Projekt gültig. Anschliessend wird in Abschnitt 5.2 das Thema Userverwaltung thematisiert. Dieses Kapitel geht schon sehr spezifisch auf konkreten Lift Programmcode ein, wie auch die darauffolgenden Kapitel. Abschnitt 5.3 befasst sich mit dem Aspekt der Persistenz. Ebenfalls mit einem Teilaспект der Persistenz, der Validierung, befasst sich der Abschnitt 5.4. Die folgenden Abschnitte sind eher Themen Richtung Interaktion mit dem Client. Abschnitt 5.5 dokumentiert die Verwendung und Erstellung von GUI Widgets. Die Möglichkeiten mit Lift interaktive Web Applikationen mit Ajax und Coment zu machen ist in Abschnitt 5.6 beschrieben. Der letzte Abschnitt 5.7 schlussendlich widmet sich der Entwicklung einer REST API für eine Web Applikation.

## 5.1 Infrastruktur

---

In diesem Abschnitt werden die Aspekte ausgeleuchtet, welche die Infrastruktur der Entwicklungs- und Laufzeitumgebung betreffen.

Dieser Abschnitt ist in die Unterabschnitte Scala / Lift, Maven und git unterteilt.

### 5.1.1 Scala / Lift

Dieser Abschnitt beschäftigt sich mit den Unterschieden der Scala Version 2.7.7, welche zu Beginn der Arbeit als stable deklariert war und zu Scala 2.8, welche sich im RC3 Stadium befindet. Anschliessend wird auf die Erneuerungen von Lift 1.1 zu 2.0 hingewiesen. Darauf wird das Zusammenspiel von Scala 2.7.7 / Lift 1.1 und Scala 2.8.0.RC3 / Lift 2.0-M5 untersucht. Zum Schluss wird die Entwicklungsumgebung IntelliJ IDEA in Verbindung mit dem Scala Plugin näher durchleuchtet.

#### Scala 2.7.7 / Scala 2.8

In diesem Kapitel werden die Scala Versionen 2.7.7 und 2.8.0.RC3 gegenüber gestellt. Es folgt die offizielle Ankündigung von Scala 2.8 und die wichtigsten Änderungen zu Scala 2.7.7.

##### *Ankündigung / Erneuerung*

Tabelle 3 zeigt die wichtigsten Neuerungen von Scala 2.8 (15).

Erneuerung	Beschreibung
<b>Named and Default Arguments</b>	Methodenargumente können per Name übergeben werden. Des Weiteren können Argumenten Standardwerte übergeben werden.
<b>Nested Annotations</b>	Annotations verschachteln. Dieses Feature ist für JPA bei einigen Domainentitätenbeziehungen ausschlaggebend.
<b>Redesigned Collections</b>	Die Klassenhierarchie wurde überarbeitet.
<b>Package Objects</b>	Packages per object importieren. Diese sind nötig, um z.B. die Abwärtskompatibilität zu den überarbeiteten Collectionbibliotheken zu erhalten.
<b>@specialized Annotation</b>	Annotation für primitive Datentypen um Boxing / Unboxing zu verhindern.
<b>Revamped REPL<sup>1</sup></b>	Neue Funktionen hinzugefügt. Befehlsvervollständigung integriert.
<b>Continuations</b>	Der Programmfluss kann beliebig gesteuert werden. Einer Methode kann mitgeteilt werden, wohin der Kontrollfluss nach Beendigung der Methode übergeben werden soll.
<b>Scala Swing Libraries</b>	Ausgebaut, verbessert und dokumentiert.

Tabelle 3: Scala, Erneuerungen

## Lift 1.1 (MS8) / Lift 2.0 (MS5)

In diesem Abschnitt sollen die Lift Versionen 1.1 Meilenstein 8 und 2.0 Meilenstein 5 gegenüber gestellt werden.

### Übersicht

Am 03.05.2010 ist der fünfte Meilenstein von Lift 2.0 erschienen. Diese Version kann von der Adresse vom GitHub Repository (16) bezogen werden. Als Stable Release werden Lift 1.1 und Lift 2.0 angeboten. Für Experimentierfreudige steht zurzeit (03.05.2010) Lift 2.0 im Meilenstein 5 zur Verfügung.

### Änderungen / Erneuerungen

Die Liste der Änderungen von Lift 1.1 zu Lift 2.0 enthält grösstenteils Verbesserungen. Eine abschliessende Auflistung würde den Rahmen dieses Abschnitts sprengen. Für die Vertiefung der technischen Aspekte (Kapitel 4) bzw. für die Erstellung der Beispielapplikation sind die wichtigsten kurz aufgeführt:

<sup>1</sup> Read-eval-print loop

Änderung / Erneuerung	Aspekt
Neue REST API	REST
Menu in DSL	Sitemap (Best Practice)
Autocomplete Widget	GUI Widget
Maven 2.2.1 oder höher (aber tiefer als 3.0)	Infrastruktur
Comet Updates bei langanhaltenden Pollingvorgängen gefixt.	Ajax & Comet

Die Liste ist unter folgendem Link verfügbar:

<http://github.com/dpp/liftweb/blob/master/framework/src/changes/changes.xml>

## Scala 2.7.7 / Lift 1.1

Lift 1.1 in Kombination mit Scala 2.7.7 ist zum aktuellen Zeitpunkt (12.03.2010) als stable markiert. Lift 1.1 setzt seit Meilenstein 7 auf Scala 2.7.7 auf. Neue Projekte sollten aufgrund der zahlreichen Features, die bei > Lift 1.1 hinzugekommen sind, nicht mehr auf Lift 1.1 aufsetzen.

## Scala 2.7.7 / Lift 2.0

Lift 2.0 nutzt ebenfalls Scala 2.7.7. Lift 1.1 Projekte können laut den Lift Entwicklern ohne grossen Aufwand auf Lift 2.0 portiert werden. Für ein neues Lift Projekt, welches im produktiven Umfeld arbeiten soll, wird deshalb diese Kombination angeraten.

## Scala 2.8.0.RC3 / Lift 2.0-M5

Die zurzeit aktuellste Variante ist Lift 2.0-M5 in Verbindung mit Scala 2.8.0.RC3. Es ist allerdings nur experimentierfreudigen Entwickler angeraten, ein Projekt auf dieser Basis aufzubauen, da sie nicht als stable deklariert ist und daher einige Bugs aufweisen kann.

## Fazit

Die Entscheidung, welche Versionskombination gewählt wird hängt stark von den geplanten Aspekten eines Projektes ab und in welchem Umfeld dieses verwendet wird. In unserem Falle ist die Verwendung von JPA ein wichtiger, wenn nicht der wichtigste Bestandteil der Beispielapplikation. Es ist daher durchaus schlüssig auf eine Liftversion zu setzen, welche sich auf > Scala 2.7.7 stützt, da erst ab dieser Version verschachtelte Annotationen möglich sind. Hinzu kommt, dass die Beispielapplikation nur zu Demo Zwecken erstellt wurde und nicht in einem produktiven Umfeld funktionieren muss. Der Entscheid lag in diesem Falle nahe auf

Lift 2.0-M5 / Scala 2.8.0.RC3 zu setzen und sich dabei nahe am Puls der Lift Community zu befinden.

## Entwicklungsumgebung

IntelliJ IDEA bietet ein Scala Plugin an, welches wie die Entwicklungsumgebung selbst von JetBrains Inc. entwickelt wird. Das Scala Plugin kann vom JetBrains Plugin Repository (17) bezogen werden oder direkt in der Entwicklungsumgebung mit dem Plugin Manager installiert werden.

Zurzeit bietet IntelliJ IDEA Scala in den Versionen 2.7.1 – 2.7.6 an. Die fehlenden Komponenten, wie scala-compiler.jar und scala-library.jar, können entweder manuell angegeben oder durch Maven automatisch heruntergeladen werden. Maven lädt sich hierbei die Dateien je nach Versionsangabe herunter.



### Hinweis

Nebst IntelliJ IDEA standen ebenfalls Eclipse und Netbeans als IDE zur Diskussion. Die Stabilität und Zuverlässigkeit zum Zeitpunkt dieser Arbeit sprachen für eine Verwendung der IntelliJ IDEA.

## *IntelliJ IDEA 9.0.1 CE mit Scala 2.8*

Da IntelliJ IDEA 9.0.2 Scala Projekte bis und mit Version 2.8.0.RC1 automatisch unterstützt, muss die Entwicklungsumgebung manuell konfiguriert werden, falls eine neuere Version wie Scala 2.8.0.RC3 gewünscht wird. Die auswählbaren Scala Versionen werden erst bei einem Update des IntelliJ Scala Plugins nachgetragen und haben dadurch einen fortwährenden Rückstand. Für die folgende Anleitung muss das Scala Plugin sowie auch die Scala Version 2.8.0.RC3 bereits installiert sein.

1. File → New Project... → Create project from scratch
2. Name: „Demo“, Select Type: Java Module → Next
3. Create Source Directory: Yes → Next
4. Desired technologies: Scala wählen, Pick files from disk: /lib/\*.jar Dateien von Scala 2.8.0.RC3 auswählen → Finish

Nach Erstellung des Projekts:

1. File → Project Structure → Facets → Scala → Scala (Demo)
2. Use Scala Compiler from specified jars: lib/scala-compiler.jar und lib/scala-library.jar von Scala 2.8 auswählen.



#### Template Analysis in XHTML Editor von IDEA

Der XHTML Editor von IntelliJ IDEA führt automatisch eine Codeanalyse durch und meldet Warnings und Errors. Die ist hilfreich um valides XHTML zu schreiben. Beim schreiben von Lift Templates kann es aber störend sein, da IntelliJ IDEA den Lift Namespace nicht auflösen kann. Somit werden alle Lift Tags als Fehler angezeigt. Um dies zu unterdrücken kann in Lift Templates die erste Zeile mit `<!--suppress ALL -->` markiert werden. Es werden keine Meldungen mehr angezeigt. Andererseits kann man auch über die Option „Configure Highlight Level“ Einstellen welche Meldungen dargestellt werden sollen.

## 5.1.2 Maven

Maven (18) ist das Standard Projekt Management Tool für Scala sowie auch für Lift Projekte. Es wird für das Erstellen, Builden, Testen und Verwalten verwendet.

### Installation

Maven, aktuell in der Version 2.2.1, wird von der Apache Foundation (19) entwickelt. Bei der Software handelt es sich um ein in Java entwickeltes Command Line Tool.

Bei der Installation sollte man beachten, dass die Umgebungsvariable m2 auf das Installationsverzeichnis von Maven zeigt. Weiter sollte der /bin Pfad des Installationsverzeichnis der PATH Umgebungsvariable hinzugefügt werden, damit Maven auch verzeichnisunabhängig ausgeführt werden kann.

Für IDEA IntelliJ und andere Entwicklungsumgebungen gibt es entsprechende Maven Plugins. Es ist jedoch äusserst praktisch, wenn für gewisse Aufgaben die Maven Konsole verwendet werden kann.

### Lokale / Globale Repositories

Ein Feature von Maven ist, dass es selbständig benötigte Referenzen aus definierten globalen Repositories herunterlädt und in einem lokalen Repository speichert.

Das lokale Repository ist standardmässig unter \$HOME/.m2/repository eingerichtet. Im Verzeichnis \$HOME/.m2 kann man eine Konfigurationsdatei mit Namen settings.xml platzieren. In dieser Konfigurationsdatei können zusätzliche externe Repositories definiert werden. Für Scala & Lift Projekte sollte eine Default settings.xml, mit der Konfiguration der offiziellen Scala/Lift Repositories, angelegt werden. Codelisting 7 zeigt den Inhalt dieser settings.xml Datei.

---

```

<settings>
  <pluginGroups>
    <pluginGroup>net.sf.alchim</pluginGroup>
  </pluginGroups>
  <profiles>
    <profile>
      <id>default</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <pluginRepositories>
        <pluginRepository>
          <id>scala-tools.org</id>
          <name>Scala Tools Maven2 Repository</name>
          <url>http://scala-tools.org/repo-releases</url>
        </pluginRepository>
      </pluginRepositories>
      <repositories>
        <repository>
          <id>scala-tools.org</id>
          <name>Scala Tools Maven2 Repository</name>
          <url>http://scala-tools.org/repo-releases</url>
        </repository>
        <repository>
          <id>scala-tools.org.snapshots</id>
          <name>Scala Tools Maven2 Repository</name>
          <url>http://scala-tools.org/repo-snapshots</url>
          <snapshots />
        </repository>
      </repositories>
    </profile>
  </profiles>
</settings>

```

---

Codelisting 7: Repositories, settings.xml

Mit dieser Konfiguration wird Maven alle benötigten Referenzen welche für die verschiedenen Archetypes (siehe nächstes Kapitel) auflösen können.

## Archetypes

In Maven gilt Convention over Configuration. Dieses Prinzip findet Anwendung in den Maven Archetypes. Einfach ausgedrückt sind diese Archetypes Vorlagen für neue Projekte. Sie geben meistens die Verzeichnisstruktur, sowie benötigte Bibliotheksreferenzen und Ähnliches vor. Einige Archetypes sind auch bereits mit Beispielcode versehen.

Die folgende Tabelle 4 gibt eine Übersicht der für die Scala und Lift zur Verfügung stehenden Archetypes.

Archetype Name	Beschreibung	Pfad im Repository
<code>scala-archetype-simple</code>	Ein einfaches Scala Projekt mit Beispielcode	/org/scala-tools/archetypes/
<code>lift-archetype-blank</code>	Ein leeres liftweb Projekt	/net/liftweb/
<code>lift-archetype-basic</code>	Ein einfaches liftweb Projekt mit Beispielcode	/net/liftweb/
<code>lift-archetype-jpa-basic</code>	Demo JPA Projekt für Lift	/net/liftweb/
<code>lift-archetype-jpa-blank</code>	Leeres JPA Projekt für Lift (Unterteilt zwei Projekte; Persistence und Web)	/net/liftweb/
<code>lift-archetype-jpa-blank-single</code>	Leeres JPA Projekt für Lift (einzelnes Projekt)	/net/liftweb/

Tabelle 4: Archetypes

Als Grundlage für die Beispielapplikation wurde der Archetype `lift-archetype-jpa-blank` gewählt. Die Abhängigkeiten dieses Archetypes wurden für die Verwendung mit JPA optimiert. Weiter ist ein Projekt auf diesem Archetyp basierend, standardmäßig in zwei Module aufgeteilt. Das eine Modul für den Persistenzlayer, das andere für die Webapplikation und die Problem Domain selbst.

Es gibt zwei wichtige Repositories, in welchen die Archetypes gefunden werden können. Die folgende Tabelle 5 beschreibt diese.

Repository URL	Beschreibung
<a href="http://scala-tools.org/repo-releases/">http://scala-tools.org/repo-releases/</a>	Im Release Repository befinden sich die als stable deklarierten Versionen. Diese sind fest veröffentlicht und sind keinen Änderungen mehr unterworfen.
<a href="http://scala-tools.org/repo-snapshots/">http://scala-tools.org/repo-snapshots/</a>	Im Snapshot Repository befinden sich die als unstable deklarierten Versionen. Es sind Snapshot vom aktuellen Entwicklungsstand und werden jede Nacht gemäss dem aktuellen Stand erstellt.

Tabelle 5: Archetypes, Repositories

Prinzipiell entscheidet man sich bei der Durchführung eines Projekts entweder für ein stable Release einer gewissen Technologie/Framework, oder aber für die Entwicklung gegen den Snapshot. Wenn man sich für die Entwicklung gegen einen stable Release entscheidet, hat man mehr Sicherheit gegenüber der Abhängigkeit. Der Snapshot ist immer der aktuelle Stand der Entwicklung. Wenn man auf diesem Aufbaut hat man den Vorteil, dass man Verbesserungen und neue Features direkt einsetzen kann. Gleichzeitig hat man aber auch das Risiko von nicht mehr vorhandener Funktionalität und neuen Fehlern.

Für die Entwicklung gegen aktuelle Snapshot Releases wird das Snapshot Repository verwendet. Für die Entwicklung gegen einen stable Release von Lift wird das Release Repository verwendet.

Damit Maven die Archetypes in einem Repository finden kann, werden diese durch ein 4-Tupel der Form {Name, GroupID, ArtifactID, Version} identifiziert. Diese Daten können in XML Files in den entsprechenden Ordner des gewünschten Repository

(Release, Snapshot) unter dem entsprechenden Pfad des Archetypes gefunden werden.

### *Verwendung mit IntelliJ IDEA*

Um ein neues Lift Webprojekt zu erstellen ist in IntelliJ IDEA wie folgt vorzugehen. Als Beispiel wird der Lift Archetype für JPA verwendet.

1. File → New Project... → Create project from scratch
2. Name: „LiftJPADemo“, Select Type: Maven Module → Next
3. Add Archetype

- a. Folgender Dialog wird angezeigt



Abbildung 2: Archetype hinzufügen in IntelliJ IDEA

- b. Daten gemäss Abbildung 2 ausfüllen (lift-archetype-jpa-blank für Lift2.0/Scala2.8Beta1 aus dem Snapshot Repository).
- c. Die gewünschte Identifikation liefern die maven-metadata.xml Dateien welche im gewünschten Repository unter dem Archetype Pfad zu finden sind (Im Beispiel: <http://scala-tools.org/repo-snapshots/net/liftweb/lift-archetype-jpa-blank-single/maven-metadata.xml>)

4. Neu hinzugefügter Archetype auswählen → Finish

Nach dem Abschliessen des Assistenten wird Maven mit den entsprechenden Parametern gestartet und ausgeführt. Die Maven Konsole wird für die Ausgabe angezeigt. Wenn diese BUILD SUCCESSFUL meldet, ist die Erstellung des Projekts abgeschlossen.

In diesem Fall wird ein Maven Projekt mit einem Master- und zwei Child Modulen erstellt. Im Maven Properties Window in IDEA kann man “Force Reimport All Maven Project” wählen, damit alle Maven Projekte erkannt werden (siehe Abbildung 3).

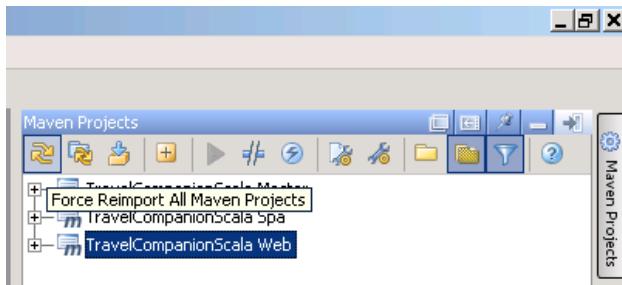


Abbildung 3: Maven Tab in IntelliJ IDEA

Das Scala Plugin von IDEA verfügt ebenfalls über Lift Support. Wenn das Kontextmenü des Web – Projekt aufgerufen wird, sollte ein Punkt „Run Project-web[jetty:run]“ auswählbar sein. Dieser Eintrag ruft das Maven Goal jetty:run auf, welches einen lokalen Jetty Webserver auf <http://localhost:9090> startet und das Projekt auf diesem deployed. Nach der Ausführung dieses Maven Goals kann das Projekt in einem Browser angeschaut werden.

Gemäss diesem Vorgehen können auch die anderen beschriebenen Archetypes für Scala & Lift verwendet werden.

### 5.1.3 Git

Git (20) ist ein verteiltes System für die Versionsverwaltung. Es wurde speziell für Geschwindigkeit optimiert. Viele Open Source Projektteams verwenden Git als Versionsverwaltungssystem. Das spezielle an Git ist, dass jedes Arbeitsverzeichnis ein vollständiges Repository mit Änderungsgeschichte etc. ist.

GitHub (21) bietet Hosting Service für Git Repositories an. Der Provider wird von vielen bekannten Open Source Projekten, darunter auch das Lift Webframework, genutzt.

Für die Entwicklung der Beispielapplikation wurde ebenfalls ein GitHub Repository (22) verwendet. Es ist empfehlenswert für Projekte basierend auf einer neuen Technologie, im selben Umfeld wie die eigentliche Entwicklung zu arbeiten, da diese mit Sicherheit die am besten unterstützte ist. Um unsere Arbeit der Lift Community nach Abschluss zur Verfügung zu stellen, ist GitHub als Plattform optimal. Aus diesen Gründen haben wir uns für die Arbeit mit Git auf GitHub entschieden.

### dpp Repository

Der gesamte Code rund um das Lift Webframework wird auf dem GitHub Repository von David Pollak (16) gehosted. Sämtlicher Source Code, sowie ein hilfreiches Wiki sind dort verfügbar. Das Wiki jedoch ist nicht mehr auf dem aktuellen Stand und wird auch nicht mehr aktualisiert. Es wurde ein neues, offizielles Lift Wiki (9) auf Assembla (Online Workspaces für Softwareentwicklung) erstellt.

## Installation

Git, aktuell in der Version 1.7.1, kann von der Projektwebseite (20) heruntergeladen werden. Git ist ein Command Line Tool. Für die verschiedenen IDEs gibt es entsprechende Plugins, welche Git als Source Code Management Tool einbinden.

Für Windows gibt es zwei verschiedene Alternativen. Einerseits kann Cygwin installiert werden, welches bereits mit Git ausgeliefert wird oder aber man installiert msysGit, welches eine GitBash für Windows mitliefert.

Will man aktiv auf GitHub arbeiten, so muss man zuerst ein Account anlegen und anschliessend ein SSH Public/Private Key Pair generieren. Der SSH Public Key muss auf GitHub unter Account Settings → SSH Public Keys gespeichert werden. Der Private Key wird benötigt um sich gegenüber GitHub zu authentifizieren. Für das gemeinsame Entwickeln muss das in GitHub gewünschte Konto als Collaborator hinzugefügt werden. Der entsprechende Vorgang für die Einrichtung der Public Keys wird in einem Artikel auf der GitHub Hilfeseite (23) beschrieben.

## Verwendung mit IntelliJ IDEA

Ein bestehendes Projekt in IDEA IntelliJ kann mittels Version Control → Git Init für eine Publikation auf einem GitHub Repository vorbereitet werden. Dabei wird ein lokales Git Repository erstellt. Anschliessend müssen die Files des Projektes dem Version Control System hinzufügt werden. Für diesen Zweck gibt es das Changes Fenster im unteren Bereich der IDE (siehe Abbildung 4).



Abbildung 4: Changes Tab in IntelliJ IDEA

Wenn alle Dateien dem VCS hinzugefügt wurden, kann über das Menü Git → Commit einen initial commit durchführt werden. Dabei wird automatisch ein master Branch erstellt.

---

### Warnung



Wenn man in IntelliJ IDEA eine neue Datei erstellt wird man gefragt ob diese dem VCS hinzugefügt werden soll. Metadateien der IDE (Projektfiles etc.) oder Binaries sollten nicht ins Repository gepusht werden.

---

Um das Repository auf den GitHub zu clonen, muss zuerst eine remote Referenz erstellt werden. Dazu wechselt man in der Git Bash auf das Projektverzeichnis. Der folgende Befehl definiert ein Bezeichner origin für das Remote Repository:

```
$ git remote add origin  
git@github.com:rmuri/TravelCompanionScala.git
```

Anschliessend kann das lokale Repository auf das GitHub Repository gepusht werden. In IntelliJ IDEA geschieht dies über den Menüpunkt Version Control → Git → Push Changes...

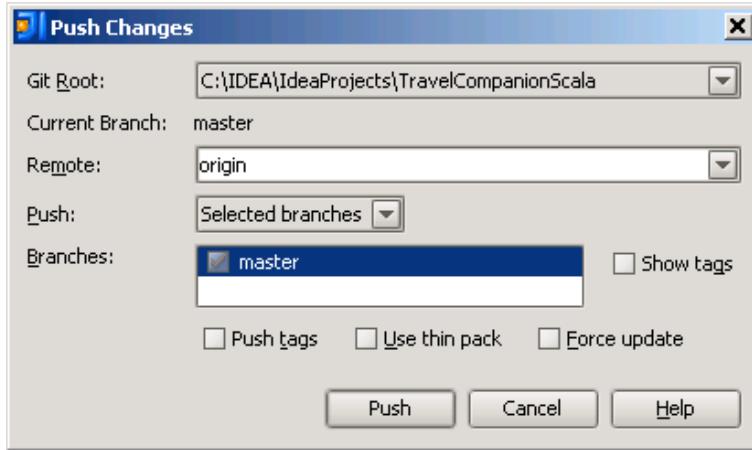


Abbildung 5: Git Push Dialog in IntelliJ IDEA

Abbildung 5 zeigt die Einstellungen. Damit wird das lokale Repository auf das globale GitHub Repository gepusht.

#### Best practice



Wichtig bei der Arbeit mit Git ist, dass beachtet wird, dass es ein verteiltes Versionsmanagement System ist. Konkret bedeutet dies, dass jeder Entwickler ein komplettes, lokales Repository hat. Änderungen müssen auf das globale Repository „gepusht“ werden. Von anderen Entwicklern gemachte Änderungen müssen vom globalen Repository „gepulkt“ werden.

## 5.2 Userverwaltung

---

In diesem Abschnitt werden die Möglichkeiten aufgezeigt, welches das Lift Framework bietet, eine Userverwaltung mit Authentifizierung und Zugriffskontrolle zu realisieren. Als Weiterführender Aspekt werden die Möglichkeiten zur Umsetzung einer Rollenbasierten Zugriffskontrolle betrachtet.

Die Realisierung einer Userverwaltung und Zugriffskontrolle mit Lift kann in zwei grosse Kategorien aufgeteilt werden.

Für die einfache, geradlinige Web Applikationsentwicklung bietet Lift viel vorgefertigten Code, welcher übernommen und je nach dem erweitert werden kann.

Gehen die Anforderungen für die Authentifizierung und Zugriffskontrolle über den Normalfall hinaus, muss diese jedoch selbst entwickelt werden. In den folgenden Abschnitten wird auch beschrieben wie solche Bedürfnisse mit der Unterstützung von Lift umgesetzt werden können.

### 5.2.1 Verwendung von ProtoUser

#### Übersicht Funktionsumfang

Für die einfache und schnelle Erstellung von Webapplikationen, bietet Lift viel Hilfe an. So auch in der Userverwaltung. Das Package `net.liftweb.mapper` enthält die beiden Traits `ProtoUser` und `MegaProtoUser`. Diese definieren einen einfachen User Account und können sehr einfach verwendet werden. Zur Verfügung stehen Felder, welche einen Benutzer im Web standardmäßig identifizieren kann wie Email, Name, Passwort etc. sowie folgende Funktionalitäten:

- User Registrierungsseite mit konfigurierbarer Email Validierung.
- Login Seite mit Authentifizierung.
- Eine „Passwort-vergessen“ Seite welche einen Reset per Email realisiert.
- Eine „User-bearbeiten“ Seite für das editieren der Benutzerdaten / Passwort.

Wie bereits erwähnt, befinden sich diese beiden Traits im Package `net.liftweb.mapper`. Die Traits sind so implementiert, dass Sie durch den Lift OR Mapper auf eine entsprechend konfigurierte Datenbank persistiert werden.

Diese gesamte beschriebene Funktionalität steht ohne grossen Implementations- bzw. Konfigurationsaufwand für eigene Webapplikationen zur Verfügung. Notwendig ist lediglich eine vorhandene Abhängigkeit zum Mapper Package.

**Warnung**

Verwendet man als Grundlage für sein Projekt ein JPA Archetype (wie z.B. bei der Beispielapplikation) ist standardmässig keine Abhängigkeit auf die Mapper Packages gesetzt. Das macht auch Sinn da ein durchmischen der OR Mapper Abhängigkeiten eine potentielle Gefahr für schlechtes Design darstellen würde.

Die Verwendung in eigenen Applikationen erfolgt nach folgendem Schema. Es wird eine eigene Userklasse erstellt welche von ProtoUser ableitet. Dort wird definiert über welche Felder der User verfügen soll und in welchen Bereichen sich das Verhalten vom implementierten Standartverhalten unterscheiden soll. Ein Blick in den Source Code der ProtoUser und MegaProtoUser in der API Dokumentation vom Lift Mapper (24) kann für das Verständnis sehr hilfreich sein. Die Verwendung der beiden Klassen wird im Lift Buch (8 S. 107) in einem kurzen Kapitel beschrieben. Tutorialmässige Beschreibungen zur Verwendung gibt es in der Lift Group (13) genügend.

## Schwachstellen / Limitationen

Auf den ersten Blick scheint die Verwendung von ProtoUser für die eigene Userverwaltung sehr attraktiv. Leider hat die Klasse auch einige erhebliche Nachteile, welche sie für etwas komplexere und umfangreichere Webapplikationen unpassend macht.

Die folgende Tabelle 6 fasst die gravierendsten Nachteile zusammen.

Problem	Beschreibung
<b>Abhängigkeit zu Mapper</b>	Die beiden Traits befinden sich im net.liftweb.mapper Package. Die Abhängigkeit zu Mapper Klassen ist sehr gross. Wenn man nicht den Lift OR Mapper für die Persistenz verwenden möchte, ist ein Grossteil des Codes unbrauchbar.
<b>Cohesion</b>	Die gesamte Funktionalität kommt beinahe in einer einzigen, riesigen Klasse daher. Diese wirkt zum Teil mehr als ein grosses Stück Beispielcode wie man etwas machen könnte als ein tatsächlicher Framework Bestandteil.
<b>Markup Code in Model</b>	Es ist verhältnismässig viel HTML Code im gesamten ProtoUser Trait vorhanden. Dies ist für Erweiterungen und Vereinheitlichung des Design unpraktisch.

Tabelle 6: ProtoUser Schwachstellen

## Fazit

Die beschriebenen Probleme sind auch oft angesprochene und kritisierte Punkte in der Lift Group (13). Nach Aussagen von Entwicklern wird die Klasse ProtoUser bald Ziel eines grundlegenden Refactorings. Dabei soll die Koppelung zu Mapper gelöst werden und die Klasse erweiterungsfähiger gemacht werden. Ohne diese Anpassungen ist die Klasse ProtoUser ein nettes und umfangreiches Codebeispiel für

die einfache Implementierung einer Userverwaltung, für den Einsatz in einem Enterprise System aber eher ungeeignet.

#### Best practice



Von den Lift Entwicklern wird vorgeschlagen, die benötigten Teile des ProtoUser Codes in eigene Userklassen zu kopieren und entsprechend anzupassen. Die ist zwar nicht unbedingt die gewünschte Framework Praxis, aber ein funktionierender Straight-Forward Ansatz.

Wenn man eine etwas komplexere Webapplikation bauen und nicht mit dem Lift OR Mapper auf den Persistence Layer zugreifen möchte, kann der ProtoUser höchstens als Inspiration für die eigene Implementierung der Userverwaltung dienen. Mehr bietet diese Klasse im Moment leider nicht.

In der Beispielapplikation haben wir uns für diesen Ansatz entschieden. Einige Teile des ProtoUser Code haben wir mit Anpassungen auf unsere Umgebung übernommen und weiterentwickelt.

Es ist vorstellbar, nach einem Grundlegenden Refactoring dieser Klasse, in einer kommenden Lift Version, mehr Unterstützung durch das Framework bezüglich Userverwaltung zu haben.

## 5.2.2 Lift Konzepte für Userverwaltung

### Übersicht

In diesem Abschnitt werden die verschiedenen Konzepte vorgestellt, welche das Lift Framework zur Verfügung stellt und für die Realisierung einer Userverwaltung verwendet werden können.

### Zustandsabstraktion

In Webapplikationen ist es durch die http Protokollfunktionsweise hilfreich zu wissen, welche Lebensdauer eine Variable besitzt. Lift verfügt über eine starke Abstraktion des http Protokolls. Trotzdem muss man sich in einigen Bereichen Gedanken über die Gültigkeit des Zustandes machen. Im Folgenden wird die Thematik des Zustandes in einer Lift Web Applikation dokumentiert.

#### *Lift ist Stateful per Default*

Praktisch die gesamte Funktionalität des Lift Framework ist Stateful. Eine Ausnahme bildet da das Stateless Dispatch (REST handling). Das gesamte Form handling ist Zustandsabhängig ausser man würde selbst HTML Formularfelder definieren und einen eigenen Handler definieren. Sobald eine Scala Funktion an einen Formular oder Link Handler gebunden wird, ist diese Funktion Zustandsabhängig. Gerade in der Zustandsabhängigkeit, welche die starke Abstraktion des http Protokolls erst erlaubt, liegt die Stärke von Lift.

## Zustand von Snippets

Standardmässig werden in Lift normale Snippets, welche den Snippet Zustand nicht sichern, verwendet. Das bedeutet dass Lift für jeden Request eine neue Instanz eines entsprechenden Snippets erstellt. Änderungen an Instanzvariablen etc. sind nach dem Request verloren. Das heisst aber nicht dass im Hintergrund für den Zustand des Snippets nicht festgehalten wird. Diese Snippets sind also nicht Stateless oder nur in bezug auf die Instanzvariablen. Es ist in diesen Snippets trotzdem möglich Scala Funktionen an Handler zu binden oder Ajax Funktionalität zu verwenden.

Weiter gibt es Stateful Snippets. Der Hauptunterschied zum Stateless Snippet ist, dass die gleiche Instanz des Snippets für jeden Page Request innerhalb einer Session verwendet wird. Eine Instanz eines Stateful Snippet ist also an eine Session gebunden. Stateful Snippets eignen sich z.B. für Formulare, welche über mehrere Seiten gehen. Codelisting 8 zeigt ein kleines Beispiel eines Stateful Snippet.

---

```
class StateExample extends StatefulSnippet {
    private var rememberMe = 0

    val dispatch: DispatchIt = {
        case "run" => run _
    }

    def run(html: NodeSeq): NodeSeq = {
        bind("count", html,
            "rememberMe" -> rememberMe,
            "submit" -> SHtml.submit("Increment",
                () => rememberMe += 1))
    }
}
```

---

Codelisting 8: Stateful Snippet

Dieses Snippet hat eine einzelne lokale Variable welche als eine Art Counter dient. Bei jedem Form Post Event wird sie inkrementiert. Ein Stateful Snippet hat eine Dispatch Methode welche bestimmt zu welcher Methode ein Snippet Aufruf weitergeleitet wird. Der Markup Code zu diesem einfachen Beispiel befindet sich im Codelisting 9.

---

```
<lift:StateExample.run form="POST">
    You were here <count:rememberMe/> times!
    <count:submit/>
</lift:StateExample.run>
```

---

Codelisting 9: Markup Code für Stateful Snippet Beispiel

Das Lift Snippet wird eingebunden. Dem Snippet wird gesagt, dass es auf den Form Post Event reagieren soll. Die beiden Lift Tags rememberMe und submit werden im Snippet entsprechend gebunden.

**Warnung**

Zu beachten ist dass das Stateful Snippet nicht für jeden beliebigen Aufruf den Zustand sichert. In diesem Beispiel sichert es den Zustand für den Form Post Event. Wenn der Zustand beim klick auf einen Link gesichert werden soll, muss der HTML Code für den Link über StatefulSnippet.link erstellt werden.

**Zustand von Variablen**

In Lift gibt es die Möglichkeit, die Gültigkeit von Varaiablen über Requests oder Sessions hinweg zu sichern. Zu diesem Zweck dienen die Klassen Request- & SessionVar welche im folgenden beschrieben werden.

Request- & SessionVars bieten typensicheren Zugriff auf die Daten welche sie speichern. Sie bieten die Möglichkeit ein default Wert festzulegen falls sie nicht initialisiert wurden. Ebenfalls kann mittels Lifecycle Callback auf die Zerstörung der Variable reagiert werden.

Tabelle 7 und Tabelle 8 geben eine Übersicht der beiden Klassen.

Eigenschaft	Beschreibung
<b>Lebenszeit</b>	Gültig für einen Request, d.h. ein Seitenaufruf
<b>Inhalt</b>	Beliebige Objekttypen können als RequestVar definiert werden.
<b>Verwendung</b>	RequestVars werden vor allem dazu verwendet, um den Zustand mit anderen Snippetklassen zu sharen. Sie werden daher meist als object implementiert.

Tabelle 7: RequestVar

Eigenschaft	Beschreibung
<b>Lebenszeit</b>	Gültig für eine Session. Eine Session wird über eine eindeutige Session Id identifiziert, welche der Client bei jedem Request mitschickt.
<b>Inhalt</b>	Beliebige Objekttypen können als SessionVar definiert werden.
<b>Verwendung</b>	SessionVars werden dazu verwendet, um sich den Zustand eines Benutzers zu merken. Mittels SessionVars ist es in einfachster Weise möglich, einen Benutzer wieder zu erkennen.

Tabelle 8: SessionVar

SessionVars und RequestVars sind für die Implementierung als Singleton Objects gedacht. So können sie von überall im Code erreicht werden.

```
// Erstellen einer typensicheren RequestVar
object exampleReqVar extends RequestVar[String]("default")

// Zugriff auf die RequestVar
exampleReqVar.is

// Ändern des Werts
exampleReqVar.set("newValue")
```

Codelisting 10: RequestVar Beispiel

Die Verwendung einer SessionVar funktioniert genau gleich. RequestVars und SessionVars sind eine gute Möglichkeit um Daten über mehrere Requests oder über

eine Gesamte Session weiterzugeben ohne sich mit http Parameter befassen zu müssen.

### *State Object*

Das State Object oder einfach S Object repräsentiert den Zustand des aktuellen Request einer Lift Applikation. Es wird verwendet um Informationen aus dem aktuellen Request zu lesen und ändern. Es erfüllt verschiedene Funktionen, darunter:

- Benachrichtigungen (Fehlermeldungen etc.)
- Cookie Management
- Localization/Internationalization
- Weiterleiten von Requests
- Zugriff auf http Parameter

Tabelle 9 zeigt die wichtigsten Merkmale des State Objects.

Eigenschaft	Beschreibung
<b>Lebenszeit</b>	Entspricht der Lebenszeit der Applikation
<b>Verwendung</b>	Mittels dem State Object (grosses S in Lift) kann die Applikationsumgebung abgefragt werden. So ist es möglich, Referer abzufragen, globale, applikationsweite Variablen auszulesen oder aber auch GET bzw. POST Parameter auszulesen. Hierbei gilt zu beachten, dass die Parameter mittels S.param() immer ein Objekt des Types Box[String] zurückgibt.

Tabelle 9: State Object

Codelisting 11 zeigt einige häufige Anwendungen des S Object.

---

```
// Weiterleiten eines Requests
S.redirectTo("/new/page")

// Zugriff auf einen lokalisierten String
S.?("myKey")

// Benachrichtigung und Fehlermeldung
S.notice("Eintrag hinzugefügt")
S.error("Hinzufügen fehlgeschlagen")

// Auslesen des http Parameter ,value'
S.attr("value").map(_.toString) openOr "defaultValue"
```

---

Codelisting 11: Verwendungen des S Object

Das S Object bietet noch einiges mehr an Funktionalität. Für einen kompletten Überblick kann die ScalaDoc im Lift Source Code (25) verwendet werden.

### Zugriffskontrolle mit SiteMap

SiteMap ist ein mächtiges und grundlegendes Konzept des Lift Webframeworks. Auf den ersten Blick stellt es ein einfaches Menü mit Navigationseintragen für eine

Webapplikation zur Verfügung. Es ist jedoch noch zu viel mehr fähig. Ein wichtiger Teil seiner Funktionalität ist die Zugriffskontrolle über mehrere Stufen. Somit ist das SiteMap ein wichtiger Pfeiler wenn es darum geht, eine Userverwaltung in einer Lift Webapplikation zu realisieren.

### *Grundlagen*

Die SiteMap ist eine globale Map, in welcher sämtliche gültigen Seiten bzw. Navigationspfade (URIs) eingetragen werden. Ist ein URL nicht in der SiteMap eingetragen, ist auch kein Zugriff möglich. Lift blockiert diesen, obwohl die Seite physikalisch vielleicht existieren mag. Das globale SiteMap besteht aus einem oder mehreren Menüs.

### *Menu*

Die Klasse net.liftweb.sitemap.Menu repräsentiert ein Menü. Ein Menüeintrag besteht aus einer Location (net.liftweb.loc), welche folgende Elemente enthält:

- Eindeutige Bezeichnung für die Location.
- Der effektive Link für die Location (URI, prefix matching möglich).
- Text für den Menüeintrag.
- LocParam Parameters: Optionales Set von Parametern, mit welchen das Verhalten des Menüeintrages kontrolliert werden kann.

In Lift 2.0 MS5 wurde eine einfache DSL (Domain Specific Language) für das Erstellen von Menüeinträgen implementiert. Geübte Programmierer können mit Hilfe von DSL's produktiver arbeiten.

---

#### **Hinweis**



Eine DSL ist eine ausgeklügelte und einfach zu verwendende Implementation einer spezifischen Problem Domain mit Hilfe von speziellen Sprachfeatures. In Lift werden für verschiedene kleine Bereiche solche DSL's verwendet. Sie haben zur Folge dass mit sehr wenig Code viel erreicht werden kann.

Menüs können auch verschachtelt werden, somit können hierarchisch strukturierte Menüs realisiert werden.

Folgendes Codelisting 12 zeigt die Verwendung von Locs und Menu.

---

```

val home = Menu(Loc("home", "index" :: Nil, "Hauptseite"))
val goodReference = Menu(Loc("reference",
ExtLink("http://www.google.ch"), "Google"))
val blog = Menu(Loc("blog", "blog" :: Nil, "Blog", AuthRequired))
// AuthRequired optionales LocParam
val help = Menu(Loc("helpHome", ("help" :: "" :: Nil) -> true,
"Help")) // Prefix Match Link
// Ab Lift2.0-M5 (Vereinfachte DSL)
Val items = Menu("Items") / "items" / "list" >> AuthRequired

```

---

Codelisting 12: Locs und Menu

Die globale SiteMap besteht aus den einzeln definierten Menüs. Das folgende Codelisting 13 zeigt das Vorgehen.

---

```

class Boot {
  def boot {
    LiftRules.setSiteMap(SiteMap(home, blog, help))
    // erste Möglichkeit
    val menus = home :: blog :: help :: Nil
    LiftRules.setSiteMap(SiteMap(menus: _*))
    // zweite Möglichkeit
  }
}

```

---

Codelisting 13: SiteMap

Der zweite wichtige Teil für das Erstellen von Menüs ist das <lift:Menu /> Tag. Dieses ist für das Rendering der Menüs in XHTML zuständig. Es verwendet ein eingebautes Snippet (net.liftweb.builtin.snippet.Menu) für das Handling von verschiedenen Rendering Funktionalitäten.

Das Codelisting 14 rendert das gesamte Menü.

---

```

<div class="menu">
  <lift:Menu.builder/>
</div>

```

---

Codelisting 14: Menu, Rendering

Es gibt auch verschiedene Möglichkeiten einzelne Elemente des Menüs mit benutzerdefinierten Attributen zu versehen.

Codelisting 15 rendert ein einzelnes Element des Menüs.

---

```

<lift:Menu.item name="index" a:class="homeLink">
  <b>Go Home</b>
</lift:Menu.item>

```

---

Codelisting 15: Menu, Rendering, Einzel

Im Beispiel sieht man auch gleich, wie ein Lift Tag (lift:Menu.item) mit einem HTML Attribut versehen werden kann. Hier sagt man dass das erstellte <a/> Tag mit dem Attribut class und dem entsprechenden Wert versehen werden soll.

Das globale Sitemap kann in verschiedene Menügruppen aufgeteilt werden. In den meisten Fällen möchte man ja nicht die gesamte Sitemap rendern. Für diesen Zweck wird der LocParam mit der Bezeichnung „LocGroup“ verwendet. Im folgenden Kapitel werden die LocParams noch detaillierter erläutert.

Codelisting 16 zeigt wie eine einzelne Menügruppe mittels einer LocGroup dargestellt werden kann.

---

```
<div class="site">
  <lift:Menu.group group="site" a:class="siteLink">
    <li>
      <menu:bind/>
    </li>
  </lift:Menu.group>
</div>
```

Codelisting 16: Menu, Group

### *LocParam*

Die optionalen LocParams sind ein essentieller Teil um das Verhalten von Locs steuern zu können. LocParams sind einfache Lift Klassen welche das AnyLocParam Trait implementieren. Es gibt verschiedene Arten von LocParams um verschiedene Eigenschaften wie Aussehen und Verhalten zu steuern. Einem Loc können mehrere LocParams mitgegeben werden.

Mit dem LocParam Hidden kann ein Menüeintrag versteckt werden. Der entsprechend parametrisierte Eintrag wird im Menü nicht angezeigt, Zugriff darauf ist aber trotzdem möglich. Im Codelisting 17 wird die Anwendung illustriert.

---

```
Loc("hideme", "hideme" :: Nil, "Hidden Link", Hidden)
```

Codelisting 17: Menu, Loc

Mit dem LocParam If können Bedingungen geprüft werden. Der LocParam nimmt eine Testfunktion ()=>Boolean sowie eine Funktion für den Fehlerfall ()=>LiftResponse entgegen. Wenn die Testfunktion true zurückgibt, dann wird die Seite normal angezeigt, ansonsten wird die Fehlfunktion ausgeführt. Das folgende Codelisting 18 zeigt eine einfache Anwendung.

---

```
val loggedIn = If(() => User.loggedIn_?, () =>
  RedirectResponse("/login"))
val profileMenu = Menu(Loc("Profile", "profile" :: Nil, "Profil",
  loggedIn))
```

Codelisting 18: LocParam

Die Funktion loggedIn wird als LocParam angegeben. Sie delegiert die Verarbeitung an die Methode loggedIn\_? der Klasse User. Anstelle dieser Delegation könnte natürlich eine beliebige Überprüfung stehen. Im Fehlerfall leitet die Applikation den Benutzer auf die login Seite weiter.

Weiter gibt es den Unless LocParam welcher eigentlich das Gegenteil des If LocParam ist. Die Seite wird angezeigt falls die Testfunktion false zurückgibt. Mit zusammengehängten If und Unless LocParams können beliebige Bedingungen überprüft werden. Die Zugriffskontrolle auf Page Level kann so relativ bequem realisiert werden. Für Locations bei denen die Fehlfunktion keinen gültigen Wert zurückliefert werden die entsprechenden Einträge im Menü ausgebendet.

Es gibt weitere LocParams mit deren Hilfe Verhalten oder Aussehen von Seiten von Code Logik abhängig gemacht werden können. Diese können (in gewissen Fällen) ebenfalls für das Steuern des Zugriffs auf gewisse Funktionalitäten der Webapplikation verwendet werden. So könnten zum Beispiel auf einer View Links zum Entfernen von Objekten bei einem nicht authentifizierten User nicht angezeigt werden (anderes Aussehen) oder der Inhalt einer Seite durch ein anderes Snippet (anderes Verhalten) gerendert werden.

Die folgende Aufzählung bietet eine Übersicht der LocParam Klassen in Lift:

- net.liftweb.sitemap.Loc.Template

Der Template LocParam nimmt eine Funktion () => NodeSeq entgegen. Die Funktion kann abhängig von gewisser Logik unterschiedliche Inhalte in Form von NodeSeq zurückgeben.

---

```
val templ = Template({
  () => if (User.loggedIn_?) {
    <b>Eingeloggt</b>
  } else {
    <b>Nicht eingeloggt</b>
  }
})
```

Codelisting 19: LocParam Template

- net.liftweb.sitemap.Loc.Snippet

Über den LocParam Snippet kann gesteuert werden, welches Snippet bzw. welche Funktion für das tatsächliche Rendering verwendet werden soll. Das nachfolgende Beispiel zeigt die Verwendung. Enthält die Location auf welcher der LocParam ausgeführt wird über das <lift:mySnippet/> Snippet, wird dieses auf die Methode welche aus der Logik resultiert gebunden.

---

```
val snippet = Snippet("mySnippet",
  if (User.loggedIn_?) {
    Entries.list -
  } else {
    Utils.welcome -
  })
```

Codelisting 20: LocParam Snippet

- net.liftweb.sitemap.Loc.DispatchLocSnippet

Der LocParam DispatchLocSnippet gibt die Möglichkeit mehrere Snippet Mappings für eine einzelne Location zu definieren. Mit diesem LocParam welche Funktionen für welche Snippets verwendet werden. Im Beispiel wird für die <lift:entries/> und <lift:add/> Snippets die Dispatch Funktion definiert.

---

```
val dsnippet = new DispatchLocSnippets {
    def dispatch = {
        case "entries" => Entries.list _
        case "add" => Entries.newEntry _
    }
}
```

---

Codelisting 21: LocParam DispatchLocSnippet

- net.liftweb.sitemap.Loc LocGroup

Eine Location wird über einen oder mehreren String Namen zu einer LocGroup zugeteilt.

---

```
Val siteMenu = Menu(Loc( ..., LocGroup("admin", "site")))
```

---

Codelisting 22: LocParam LocGroup

Das Sitemap ist ein wichtiger Pfeiler in der Zugriffskontrolle von Lift. Die LocParams werden für erweitertes Verhalten von Locations verwendet und somit ein sehr wichtiger Bestanteil der Zugriffskontrolle.

## Simple rollenbasierte Zugriffskontrolle

Bei der rollenbasierten Zugriffskontrolle wird einem User eine oder mehrere Rollen zugewiesen welche seine effektiven Berechtigungen steuern. Ein solches System ist sehr dynamisch da User und Rollen und die Zuordnungen optimalerweise in einer Datenbank gespeichert werden.

Dabei muss zwischen Authentifizierung und Autorisierung unterschieden werden. Authentifizierung beschreibt den Vorgang die Identität des Users zu verifizieren. Die Autorisierung bestimmt ob ein authentifizierter User aufgrund seiner Berechtigung (Rolle) eine Funktion ausführen oder auf eine Ressource zugreifen darf. Dieser Abschnitt legt den Fokus auf das Autorisierungsproblem.

### *Lift Unterstützung mit HTTP Authentication*

Lift bringt von Haus aus Funktionalität zur Verwendung von http Authentication mit. Dazu gehört auch die Definition und Überprüfung von Rollen. Die Verwendung von http Authentication ist in der Praxis nicht sehr verbreitet da die Möglichkeiten ziemlich begrenzt sind.

Die implementation einer Hierarchischen Struktur mit Rollen (RBAC, Role Based Access Control) welche von Lift zur Verfügung gestellt wird, beschränkt sich leider nur auf http Authentication und kann nicht mit einer eigenen Authentication Implementation verwendet werden. Aus diesem Grund wird hier nicht detaillierter darauf eingegangen.

Im Lift Buch (8 S. 149) wird in einem separaten Kapitel etwas ausführlicher auf diese Möglichkeit eingegangen.

## *Realisierung einer Authorisierung mit Sitemap und LocParam*

Wie bereits erwähnt ist das Sitemap das Herzstück der Zugriffskontrolle mit Lift. Somit wird auch die Umsetzung eines RBAC System mit Hilfe des Sitemaps realisiert.

Wir gehen von einem einfachen Beispiel aus, in welchem ein User eine Liste von Rollen besitzt. Um den Zugriff zu einer Location zu gewähren muss überprüft werden, ob der User eine entsprechende Rolle eigen hat. Dies kann man über einen If LocParam realisieren.

Der folgende Codelisting 23 zeigt dies Beispielhaft.

---

```
val EntryModification = If(
  () => {
    (UserManagement.currentUser.roles.exists(_ == "mod"))
  },
  () => RedirectWithState("/accessrestricted", RedirectState(() =>
  S.error("access denied")))
)
```

Codelisting 23: If LocParam zur Überprüfung der Rolle

Dieser LocParam überprüft ob die Liste roles des aktuellen Users über eine Rolle mit der Bezeichnung „mod“ verfügt. Falls diese Funktion true zurück liefert wird der Zugriff gewährt. Andernfalls wird der User auf den Pfad /accessrestricted weitergeleitet.

Eine Location kann ganz einfach mit diesem LocParam abgesichert werden, das folgende Codelisting 24 zeigt wie.

---

```
Menu(
  Loc(
    "blog_remove",
    "blog" :: "remove" :: Nil,
    "Remove Entry"
    LoggedIn, EntryModification, LocGroup("blog")
  )
)
```

Codelisting 24: Location mit LocParam sichern

Im Beispiel wird die Location mit dem LocParam EntryModification gesichert. Zusätzlich wird sie auch noch mit LoggedIn und LocGroup gesichert welches weitere LocParams repräsentieren.

## *Fazit*

Der rudimentäre Ansatz aus dem Beispiel aus dem letzten Abschnitt zeigt auf, dass mit Hilfe des Sitemap von Lift die Umsetzung eines RBAC prinzipiell möglich ist.

In der Beispielapplikation wurde der hier beschriebene Ansatz verfolgt und eine einfache Rollenbasierte Zugriffskontrolle umgesetzt.

Vom Framework her selbst wird bei der Verwendung von http Authentication entsprechende Funktionalität zur Verfügung gestellt, welche es erlaubt eine

Rollenbasierte Zugriffskontrolle zu implementieren. Wenn man auf eigene Authentifizierungslogik zurückgreifen möchte, können diese Klassen aber nicht verwendet werden. Das Sitemap stellt die notwendige Funktionalität zur Verfügung, eine Rollenbasierte Zugriffskontrolle mit eigenen Ansätzen zu implementieren.

## 5.3 Persistenz

---

Komplexere Webapplikationen verwenden für die Business-Logik eine objektorientierte Problemdomain. Diese muss oft mit einer relationalen Datenbank abgeglichen werden. Dafür werden sogenannte Object Relational Database Mapper, kurz OR-Mapper verwendet. In Java hat sich das Java Persistence API (JPA) als De-facto-Standard etabliert. Scala/Lift stützt sich auf bewährte Konzepte aus der Java-Welt. Dazu gehört auch JPA.

In diesem Kapitel wird die Verwendung von JPA als Persistenzlayer untersucht. Hierzu wird kurz auf die Projektstruktur des Lift Projektes eingegangen. Auf die Anbindung des JPA Providers an die Datenbank und die Annotation der Scala Klassen wird im Unterkapitel 5.3.2 genauer eingegangen.

### 5.3.1 Projektstruktur

Die Liftcommunity stellt für JPA Projekte zwei strukturell unterschiedliche Archetypes zur Verfügung.

- lift-archetype-jpa-blank
- lift-archetype-jpa-single

Ein Lift Beispielprojekt steht unter dem Namen lift-archetype-jpa-basic zur Verfügung. Kapitel 0.0.0 zeigt die Anwendung eines Archetypes mit IntelliJ.

Der Unterschied zwischen den Archetypes liegt in der Projektstruktur. Der Archetype lift-archetype-jpa-blank enthält die Maven Module SPA und WEB, wobei das Modul SPA ausschliesslich für den Persistenzlayer verwendet wird. Das WEB Modul besitzt in diesem Projekt eine Abhängigkeit auf das SPA Modul, um auf den Domainlayer zugreifen zu können.

Der Archetype lift-archetype-jpa-single besitzt lediglich das WEB Modul. Das SPA Modul ist in diesem Falle in das WEB Modul integriert.

Während sich der Archetype lift-archetype-jpa-single für neue, kleinere Projekte eignet, ist bei grösseren, ständig erweiterbaren oder bereits bestehenden Projekten übersichtshalber der lift-archetype-jpa-blank vorzuziehen. Um sowohl dem Projekt als auch diesem Bericht eine sauberere Trennung zu ermöglichen, wird im Folgenden auf die Struktur des lift-archetype-jpa-blank eingegangen. Dieser Archetype wurde ebenfalls für die Beispielapplikation verwendet.

### 5.3.2 spa Modul

Dieses Kapitel zeigt die Aufgabenbereiche des SPA Moduls auf.

## Scala Entitäten

Die JPA Entitäten sind zwar als Scala Klassen implementiert, haben allerdings gegenüber Java annotierten Klassen in keinen Vorteil, da die Scala Klassen im Java Stil geschrieben sind. Die Anzahl der Codezeilen ist nahezu identisch.

Die Codelisting 25 zeigt die Scala Entität Tour aus der Beispielapplikation.

---

```

import javax.persistence._
import _root_.java.util._

@Entity
@Table(name = "tours")
class Tour {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    var id: Long = _

    @Column(name = "description")
    var description: String = ""

    @ManyToOne
    var owner: Member = null

    @OneToMany(mappedBy = "tour", cascade = Array(CascadeType.REMOVE),
    targetEntity = classOf[Stage])
    var stages: List[Stage] = new ArrayList[Stage]()
}

}

```

---

Codelisting 25: Scala Entität

Die Unterschiede zu einer JPA Entität in Java fallen auf den ersten Blick in Scala 2.8 Beta1 minimal aus. Die nachfolgende Tabelle 10 zeigt dennoch einige Besonderheiten auf.

Typ / Parameter	Bemerkung
@OneToMany	Für @OneToMany Beziehungen sind die Collection Klassen List, Map oder Set des Packages java.util. zu verwenden sind. Dies begründet sich in der Funktionsweise von JPA, welche auf Bytecode Ebene lediglich diese Collection Klassen erlaubt. Durch diese Einschränkung müssen die java Collection Klassen in der Applikation manuell in scala Collection Klassen konvertiert werden um die Features nutzen zu können. Soll die Konvertierung implizit geschehen, reicht es aus folgenden Import anzugeben: import scala.collection.JavaConversions._
cascade	Cascaden Argument müssen als Scala Array angegeben werden. Diese Eigenschaft ist von Scala vorgegeben.
targetEntity	Vor Scala 2.8 Beta1 war dieses Argument zwingend nötig um der Laufzeitumgebung ein korrektes Casting zu ermöglichen. In der Beispielapplikation wurde es verständnishalber hinzugefügt.

Tabelle 10: Scala Entität, Unterschiede

**Warnung**

Die Syntax der Annotationen bezieht sich auf Scala 2.8. In vorhergehenden Versionen kann die Syntax abweichen. Auch sind verschachtelte Annotationen erst seit Scala 2.8 möglich. Eine oft anzutreffende, verschachtelte Annotation ist eine ManyToMany-Beziehung mit anschliessender (optionaler) JoinTable Annotation:

```
@ManyToMany(cascade = Array(CascadeType.ALL))
@JoinTable(name = "member_roles", joinColumns = Array(new
JoinColumn(name = "member", referencedColumnName = "id")),
inverseJoinColumns = Array(new JoinColumn(name = "roles",
referencedColumnName = "id")))
```

## JPA Provider

Der lift-archetype-jpa-blank Archetype kommt standardmässig mit Hibernate als JPA Provider daher. Der folgende Abschnitt zeigt den Wechsel von Hibernate zu EclipseLink schrittweise auf.

1. Änderung der Maven Konfigurationsdatei pom.xml im SPA Modul:

Die Abhängigkeit auf Hibernate muss durch die folgenden Abhängigkeiten ersetzt werden. Die zu verwendende Version soll nach eigenem Ermessen angepasst werden. Das folgende Codelisting 26 zeigt die Einbindung der EclipseLink und javax.persistence Bibliotheken.

---

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>2.1.0-SNAPSHOT</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>javax.persistence</groupId>
  <artifactId>persistence-api</artifactId>
  <version>1.0</version>
</dependency>
```

---

Codelisting 26: JPA Provider, pom.xml

2. Änderung der Konfigurationsdatei persistence.xml im Verzeichnis spa/src/main/resources/META-INF:

Codelisting 27 zeigt die persistence.xml mit EclipseLink als JPA Provider.

---

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="jpaweb" transaction-type="RESOURCE_LOCAL">

        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

        <exclude-unlisted-classes>false</exclude-unlisted-classes>

        <properties>
            <!-- Kapitel 5.2.3 -->
        </properties>

    </persistence-unit>
</persistence>
```

---

Codelisting 27: JPA Provider, EclipseLink

Im Gegensatz zu Hibernate müssen die zu verwaltenden Entitäten bei EclipseLink nochmals explizit angegeben werden. Um nicht jede Klasse mit `<class>MeinPackage.model.MeineKlasse</class>` angeben zu müssen, führt die Angabe von `<exclude-unlisted-classes>false</exclude-unlisted-classes>` dazu, dass nicht aufgelistete Entitäten dennoch verwaltet werden.

## Datenbank

Der lift-archetype-jpa-blank Archetype verwendet standardmäßig Derby als Datenbank. Der folgende Abschnitt zeigt schrittweise den Wechsel von Derby zu H2 auf.

### 1. Änderung der Maven Konfiguration pom.xml im SPA Modul

---

```

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.2.133</version>
</dependency>
```

---

Codelisting 28: Datenbank, pom.xml

Durch die Einbindung dieser Abhängigkeit kann der H2 JDBC Treiber von EclipseLink verwendet werden.

### 2. Änderung der Konfigurationsdatei persistence.xml im Verzeichnis spa/src/main/resources/META-INF:

---

```

<properties>
    <property name="eclipselink.target-database"
value="org.eclipse.persistence.platform.database.H2Platform"/>
    <property name="javax.persistence.jdbc.password" value="" />
    <property name="javax.persistence.jdbc.user" value="sa" />
    <property name="javax.persistence.jdbc.driver"
value="org.h2.Driver" />
    <property name="javax.persistence.jdbc.url"
value="jdbc:h2:file:~/TravelCompanion" />
    <property name="eclipselink.ddl-generation" value="create-
tables" />
    <property name="eclipselink.ddl-generation.output-mode"
value="database" />
</properties>
```

---

Codelisting 29: Datenbank, persistence.xml

Die Properties in Codelisting 29 sind grösstenteils selbsterklärend. Als Datenbankadresse wurde eine Datei im Benutzerverzeichnis namens TravelCompanion.h2 gewählt. Beim Start der Applikation versucht EclipseLink die Tabellen neu zu erstellen. Gelingt dies nicht, da die Tabellen bereits existieren, werden die bereits vorhandenen Tabellen ausgewählt. Eine abschliessende Liste aller Schemaoptionen befindet sich hier:

[http://wiki.eclipse.org/Using\\_EclipseLink\\_JPA\\_Extensions\\_%28ELUG%29#Using\\_EclipseLink\\_JPA\\_Extensions\\_for\\_Schema\\_Generation](http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Extensions_for_Schema_Generation)

---

#### Hinweis



Als javax.persistence.jdbc.url Property wurde jdbc:h2:file gewählt um die Applikation portabler und einfacher zu halten. Mit der Angabe von jdbc:h2:tcp anstatt file wäre eine h2 Serverinstallation nötig und damit auch in Testfällen umständlichere Umgebung geschaffen.,

### 5.3.3 Web Modul

Der Archetype lift-archetype-jpa-blank beinhaltet sowohl die nötige Abhängigkeit zum SPA Modul im pom.xml als auch das Model Object.

Das Codelisting 30 zeigt die Abhängigkeit im pom.xml im WEB Modul.

---

```

<dependency>
    <groupId>MeinProjekt</groupId>
    <artifactId>MeinProjekt-spa</artifactId>
    <version>1.0</version>
</dependency>
```

---

Codelisting 30: Web Modul, pom.xml

Das Model Object wird hierbei an die Persistence Unit gebunden und stellt gleichzeitig den EntityManager dar. Die Best Practices um mit den Domain Klassen umzugehen und damit möglichen detached Entityexceptions aus dem Weg zu gehen sind dem Kapitel 6 zu entnehmen.

## 5.4 Validierung

Das Thema dieses Kapitels ist die Validierung von Domain Entitäten. Als Domain Entitäten gelten dabei die JPA Klassen aus der Problem Domain, welche mittels Objektrelationalem Mapping auf die Datenbank bzw. den Persistenz Layer abgebildet werden. Die Validierung umfasst das Überprüfen einer konkreten Instanz auf gültige Werte in allen Feldern.

### 5.4.1 Realisierungsebene und Koppelung

Die Validierung kann auf verschiedenen Ebenen umgesetzt werden. Sie kann auf der GUI Ebene, z.B. direkt an Formulare gebunden, stattfinden oder auch erst auf der Persistenzebene, wo sie vor dem Speichern einer Entität in die Datenbank durchgeführt wird. Des Weiteren kann die Validierungslogik direkt an eine Domain Entität gebunden werden oder als „freie“ Funktion an passender Stelle, z.B. bei der Auswertung eines Eingabeformulars, im Code vorhanden sein.

#### Best practice



Wenn eine Webapplikation Ihre Dienste über verschiedene Kanäle anbietet, macht es Sinn die Validierung auf der Persistenzebene durchzuführen. So kann der Zugriff auf die Applikation z.B. über ein Browserinterface oder durch eine Webservice API erfolgen. Um duplicated Code zu vermeiden, ist es in diesem Falle ein guter Ansatz, die Validierung auf der Persistenzebene anzusetzen.

Für den Entscheid wie eine Validierung realisiert wird, sollten die Aspekte der Kopplung und Wiederverwendbarkeit als wichtige Faktoren berücksichtigt werden.

### 5.4.2 Technische Grundlagen

Für die Umsetzung einer Validierungslogik stehen verschiedene Möglichkeiten zur Verfügung. Dieser Abschnitt beschreibt die Realisierung einer möglichen Umsetzung mittels Sprachmitteln von Scala und einem Validierungsframework.

#### Einfache Validierung

Validierung besteht immer aus Überprüfungen von bestimmten Bedingungen. Ein üblicher Ansatz in Java wäre das Verwenden von diversen If - Abfragen für das Überprüfen von Bedingungen. Dies führt zu unschönem und aufgeblasenem Code.

Mit den funktionalen Aspekten von Scala können einfache und schöne Konstrukte für das Überprüfen von Bedingungen realisiert werden. Das Codelisting 31 zeigt ein entsprechendes Beispiel.

---

```
// Utility methods for processing a submitted form
def is_valid_Entry_(toCheck: Entry): Boolean =
  List(
    (if (toCheck.name.length == 0)
     {S.error("You must provide a name"); false} else true),
    (if (toCheck.content == null)
     {S.error("You must provide content"); false} else true)
  ).forall(_ == true)
```

---

Codelisting 31: Einfache Validierungslogik

Das Prinzip ist einfach. Wir erstellen eine Liste mit den Bedingungen und überprüfen, ob alle Bedingungen true zurückliefern. Es wird die error Methode des S (Session) Objekt verwendet, um allfällige Fehler zu protokollieren und dem Benutzer zurückzumelden.

In diesem Beispiel wurde die Logik als freistehende Methode implementiert. Eine weitere, denkbare Möglichkeit wäre die Methode direkt in der JPA Entity Klasse zu implementieren, so dass die aktuelle Instanz die Validierung in einer eigenen Methode zur Verfügung stellt. In diesem Falle müsste die Methode allerdings technologieunabhängiger gestaltet werden, sprich auf die Verwendung des S Objekts zu verzichten.

## JPA 2.0 Bean Validation

Die Java Persistence API JPA (26) Spezifikation enthält ab Version 2.0 Unterstützung für Validierung gemäss des Java Specification Request JSR 303 (27). Die Referenzimplementation dieser Spezifikation ist Hibernate Validator 4.x (28). JSR 303 definiert ein Metadaten Modell und eine API für JavaBean Validierung. Als Metadaten können, wie in JPA üblich, Annotationen sowie XML Deskriptoren verwendet werden.

Hibernate Validator ist ein mächtiges Validierungsframework. Constraints für Felder von Entity Klassen werden mittels Annotationen definiert. Über die API des Frameworks können Entitäten validiert werden. Häufig verwendete und für die meisten Fälle wohl ausreichende Constraints sind im Framework bereits implementiert. Es werden auch entsprechende Schnittstellen für die Implementation eigener Constraints zur Verfügung gestellt. Weitere Informationen und nicht behandelte Aspekte von Hibernate Validator können in dessen Dokumentation (29) nachgeschlagen werden.

## Verwendung von Hibernate Validator

Für die Verwendung von Hibernate Validator in eigenen (Maven basierten) Projekten sind folgende Schritte zu befolgen.

### 1. JBoss Maven Repository konfigurieren

Das JBoss Maven Repository, in dem sich der Sourcecode und die Dokumentation von Hibernate Validator befinden, muss im Maven Setup definiert werden. Dies kann durch das Hinzufügen des Repository zur lokalen

Repositorykonfiguration unter \$MAVEN\_HOME/settings.xml oder im pom File des entsprechenden Projekts erreicht werden. Das folgende Codelisting 32 zeigt dies.

---

```
<repositories>
  <repository>
    <id>jboss</id>
    <url>http://repository.jboss.com/maven2</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

---

Codelisting 32: Maven Konfiguration für JBoss Repository

## 2. Abhängigkeit im Projekt

Im pom File des entsprechenden Projekts muss ein Verweis auf das Hibernate Validator Framework gesetzt werden, damit die Klassen der API auch gefunden werden kann. Codelisting 33 beschreibt die Abhängigkeit.

---

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.0.2.GA</version>
</dependency>
```

---

Codelisting 33: Maven Hibernate Validator Dependency

---

### Bug



Das Scala-Tools Repository und das JBoss Repository gemeinsam verwendet, ergibt ein Versionskonflikt mit der Abhängigkeit von javax.mail. Die Bibliothek net.liftweb.util besitzt eine Abhängigkeit zu javax.mail ohne dabei eine Version anzugeben, was dazu führt, dass immer die aktuellste verwendet wird. Das JBoss Repository gibt vor, die Version 1.4.2 von javax.mail zu hosten, welche Maven aber nicht finden kann. Dies resultiert in einer nicht aufgelösten Abhängigkeit. Wenn in eigenen Projekten ein Verweis zu net.liftweb.util bzw. javax.mail und den entsprechenden Konflikt auftritt, sollte im eigenen pom ein Verweis auf javax.mail mit der Version 1.4.1 gesetzt werden. Dadurch wird die Verwendung dieser Version erzwungen, was das Problem löst.

## 3. Verwendung von JPA 2.0

Vorerst sollte sichergestellt werden, dass die JPA 2.0 API verwendet wird. Wenn als JPA Provider EclipseLink verwendet wird, ist zum aktuellen Zeitpunkt standardmäßig die API von JPA 1.0 aktiv. Dieses Problem kann gelöst werden, indem die Abhängigkeit von JPA (javax.persistence) explizit auf Version 2 gesetzt wird. Im Codelisting 34 wird die Abhängigkeit beschrieben.

---

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>javax.persistence</artifactId>
  <version>2.0.0</version>
</dependency>
```

---

Codelisting 34: Maven JPA 2.0 Dependency

Wenn diese Schritte soweit befolgt wurden, kann nun mit der Verwendung von Hibernate Validator begonnen werden.

## Constraint Annotationen in JPA Entity Klassen

Codelisting 35 zeigt beispielhaft, wie eine JPA Entity Klasse mit Validation Annotationen aussehen könnte.

---

```
import javax.persistence._
import javax.validation.constraints._
import org.hibernate.validator.constraints._

class Comment {
  var id: Long = _

  @NotEmpty
  var content: String = ""

  @NotNull
  var member: Member = null

  @NotNull
  var dateCreated: Date = null
}
```

---

Codelisting 35: JPA Entity Klasse mit Constraint Annotationen

Tabelle 11 listet die verwendeten Imports mit einer kurzen Beschreibung auf.

Package Import	Beschreibung
<code>javax.persistence</code>	API der JPA Spezifikation (nur Interfaces).
<code>javax.validation.constraints</code>	API der JSR 303 Spezifikation (nur Interfaces).
<code>org.hibernate.validator.constraints</code>	Referenzimplementation von Hibernate für die JSR 303 Spezifikation (konkrete Implementation).

Tabelle 11: Packages für JSR 303 Bean Validation

Die Annotationen `@NotEmpty` und `@NotNull` sind Constraint Annotationen. Sie verlangen, dass das entsprechend annotiert Feld die Bedingung, welche hinter der Annotation steht, zum Zeitpunkt der Persistierung einhält.

Hibernate Validator bringt nebst den vom Standard vorgeschriebenen Constraints noch diverse andere mit. Eine vollständige Liste der verfügbaren Constraints ist in der Dokumentation (29) zu finden.

**Hinweis**

Ist der Persistenz Layer in einer Applikation als separates Modul realisiert (Wie dies im Abschnitt Persistenz mit dem spa Modul beschrieben ist), darf nicht vergessen werden, dass bei Änderungen an z.B. Domain Entitäten (neue Annotationen etc.) das Modul neu zu builden ist (maven install).

## Validieren von Domain Entitäten

Die letzte offene Frage bezüglich der Validation ist, wie diese jetzt im eigenen Code verwendet werden kann. Die eigentliche Validierung von annotierten Domain Entitäten geschieht über die API des Hibernate Validator. Codelisting 36 zeigt eine beispielhafte Anwendung.

```
import javax.validation.{Validation, Validator}

object validator {
  def get: Validator =
    Validation.buildDefaultValidatorFactory.getValidator

  def is_valid_entity_(toCheck: Object): Boolean = {
    val validationResult = validator.get.validate(toCheck)
    validationResult.foreach(
      (e) => S.error(e.getPropertyPath + " " + e.getMessage)
    )
    validationResult.isEmpty
  }
}
```

Codelisting 36: Validierung von Domain Entities mit Hibernate Validator

Dieser Codeausschnitt stellt die Schnittstelle zwischen der eigenen Programmlogik, und der Hibernate Validator API dar. Es wird ein Singleton Object validator definiert, über welches Entitäten validiert werden können.

Dazu stellt es die Methode `is_valid_entity_?` zur Verfügung, welche ein Boolean zurückliefert. Über die API der JPA 2.0 Spezifikation kann per Factory eine Validator Instanz geholt werden. Die Methode `validate` des Validator nimmt ein Objekt entgegen und validiert dieses. Die Methode gibt zudem ein `java.util.Set` mit dem Ergebnis zurück. Im Beispielcode wird durch das Set mit dem Ergebnis iteriert, und allfällige Fehler der `S.error` Methode übergeben. Wenn das Set leer ist, war die Validierung erfolgreich.

Wenn man im eigenen Code nun eine Instanz einer Domain Entität validieren möchte, übergibt man sie einfach der Methode `is_valid_entity_?` und wertet das Ergebnis entsprechend der eigenen Verwendung aus.

Das validieren ist einfach und komfortabel. Das soeben erläuterte Beispiel zeigt nur eine ganz einfache Anwendung. Das Framework bietet noch viele weitere Funktionen für den erweiterten Bedarf. Diese Funktionalität kann in der Dokumentation (29) nachgeschlagen werden.

## Fazit

Für die Beispielapplikation wurde Hibernate Validator verwendet. Als Teil der JPA 2.0 Spezifikation ist es eine optimale Ergänzung und passt sich problemlos in die Konzepte einer Applikation mit JPA als Persistenz Layer ein.

## 5.5 GUI Widgets

---

Der erste Teil dieses Kapitels beschäftigt sich mit der Einbindung und Verwendung von lift-widgets. Im zweiten Teil wird die Erstellung eines eigenen Widgets schrittweise aufgezeigt.

### 5.5.1 Verwendung von Lift Widgets

Lift besitzt bereits von Haus aus eine Sammlung von Widgets. Diese wird laufend erweitert, weshalb wir hier auf eine vollständige Aufzählung verzichten und stattdessen auf den [github.com](http://github.com/dpp/liftweb/tree/master/framework/lift-modules/lift-widgets/src/main/scala/net/liftweb/widgets/) Ordner des Liftprojekts verweisen:

<http://github.com/dpp/liftweb/tree/master/framework/lift-modules/lift-widgets/src/main/scala/net/liftweb/widgets/>

Im folgenden Abschnitt wird die Verwendung des Tablesorter Widgets aufgezeigt, welches auch in die Beispielapplikation eingebaut wurde. Es sei bemerkt, dass es sich beim Tablesorter Widget um ein einfach verwendbares Widget handelt und sich dieser Vorgang daher nur exemplarisch auf die anderen Widgets anwenden lässt. Das Ziel ist es, die grundlegenden Abläufe aufzuzeigen und sich nicht detailliert mit dem Widget auseinander zu setzen.

#### Tablesorther Widget

Ein von Maven neu generierter Lift Archetype besitzt in der Regel keine Abhängigkeiten zum Modul Lift Widgets, weshalb dieses manuell ins pom.xml im WEB Modul eingefügt werden muss.

---

```
<dependencies>
...
<dependency>
    <groupId>net.liftweb</groupId>
    <artifactId>lift-widgets</artifactId>
    <version>2.0-scala280-SNAPSHOT</version>
</dependency>
...
</dependencies>
```

---

Codelisting 37: Widgets, pom.xml

Mit dieser in Codelisting 37 aufgezeigten zusätzlichen Abhängigkeit können nun alle zur Verfügung stehenden Lift Widgets verwendet werden.

Um das Tablesorter Widget nutzen zu können, ist die in Codelisting 38 gezeigte Zeile in die Boot Klasse einzufügen.

---

**TableSorter.init**

---

Codelisting 38: Tablesorter, Boot.scala

**Hinweis**

Ressourcen nicht gefunden?  
 Die init Funktion teilt Lift mit, in welchem Ordner nach zusätzlichen Ressourcen für die Verwendung des Widgets gesucht werden darf. Es handelt sich demnach um eine Freigabe der eigenen Ressourcen für die Lift Applikation. Die zunächst verwirrende Fehlermeldung, welche beim Unterlassen des init Aufrufs auftritt, macht in diesem Falle durchaus Sinn, ist aber auf den ersten Blick nicht nachvollziehbar, weshalb wir hier darauf nochmals ausdrücklich hinweisen möchten.

Das Tablesorter Widget ist nun bereit, verwendet zu werden. Das folgende Codelisting 39 zeigt eine XHTML Datei, welche eine einfache Tabelle beinhaltet. Das Augenmerk soll hierbei auf die erste Zeile, die id und class Angabe der Tabelle gerichtet werden.

```
<lift:TableSorterExample for="listtours"/>
<table class="tablesorter" id="listtours">
<thead>
  <tr>
    <th>Name</th>
    <th>Description</th>
  </tr>
</thead>

<tbody>
  <tr>
    <td>Scala</td>
    <td>Der Java Nachfolger</td>
  </tr>
  <tr>
    <td>Lift</td>
    <td>Webframework für Scala</td>
  </tr>
</tbody>
</table>
```

Codelisting 39: Tablesorter, tabelle.html

Im Anschluss muss nun ein Snippet mit dem Namen TableSorterExample.scala erstellt werden, um die Funktionalität des Tablesorters zu initialisieren.

```
class TableSorterExample {
  def render(xhtml: NodeSeq): NodeSeq = {
    val which = S.attr("for").map(_.toString) openOr ""
    TableSorter("#" + which)
  }
}
```

Codelisting 40: Tablesorter, TableSorterExample.scala

Die Methode render wird von Lift standardmäßig aufgerufen, falls in der XHTML Datei keine Methodennamen angegeben ist. Zunächst wird das Attribut for ausgelesen, welches in diesem Falle gleich lautet wie das id Attribut der Tabelle. Diese id wird nun dem eigentlichen Tablesorter Widget übergeben, welches daraufhin einen NodeSeq zurückliefert.

## 5.5.2 Eigenes Lift Widget

In Kapitel 5.5.1 wurde die Einbindung und Verwendung eines Lift Widgets aufgezeigt. Dieses Kapitel zeigt nun die Erstellung eines eigenen Widgets. In unserem Falle erstellen wir das für die Beispielapplikation verwendete Gauge Widget, welches ein Tachometer darstellt. Diesem Tachometer kann eine Zahl zwischen 0 und 100 mitgegeben werden.

### Gauge Widget

Bei einem Widget handelt es sich um ein eigenes von der Lift Applikation unabhängiges Package, welches grösstenteils über Ressourcen in Form von Javascript und Stylesheets verfügt. Die programmtechnische Aufgabe des Widgets liegt einzig darin, die übergebenen Werte für die Verwendung in Javascript aufzubereiten.

---

```
object Gauge {
    // Konstruktor, welchem ein Wert zwischen 0 und 100 mitgegeben
    // werden muss
    def apply(value: Int) = renderOnLoad(value)

    // Die Methode welche von der Boot Klasse aufgerufen wird.
    def init() {
        ResourceServer.allow({
            case "gauge" :: tail => true
        })
    }

    // Methode, welche ein NodeSeq zurückgibt
    def renderOnLoad(value: Int) = {
        val onLoad = "jQuery(document).ready(function() { var g = new
Gauge(); g.initialize(" + value + ", 'gauge'); });"
        <head>
            <script type="text/javascript" src={"/" +
LiftRules.resourceServerPath + "/gauge/gauge.js"}></script>
            <script type="text/javascript" charset="utf-8">
                {onLoad}
            </script>
        </head>
        <canvas id="gauge" width="269" height="269"/>
    }
}
```

---

Codelisting 41: Gauge Widget, Gauge.object

Codelisting 41 zeigt das vollständige Gauge Object, welches von einem Snippet in Form von Gauge(0...100) aufgerufen werden kann.

Die init() – Methode teilt Lift mit, das Package web/src/main/resources/toserve/gauge/ für die Durchsuchung nach Ressourcen freizugeben. Dieses Package enthält in unserem Falle die in Tabelle 12 aufgelisteten Bilder und Javascript Dateien.

Datei	Beschreibung
<a href="#">gauge.png</a>	Hintergrundbild des Tachometers
<a href="#">arrow.png</a>	Zeiger des Tachometers
<a href="#">gauge.js</a>	Javascript Datei, welche die in Javascript geschreienen Gauge Klasse beinhaltet.

Tabelle 12: Gauge, Resources

Die renderOnLoad() – Methode generiert einen NodeSeq und gibt diesen zurück. Hierfür verwendet es das von Lift zur Verfügung gestellte Feature „Head merging“. Im Script Bereich wird die Javascript Klasse Gauge instanziert und mit dem mitgegeben Wert initialisiert. Damit diese Funktion auch automatisch und erst dann ausgeführt wird, wenn der DOM (Domain Object Model) des Clienten vollständig erstellt ist, wird diese Anweisung in den jQuery OnLoad Wrapper eingefügt.

Auf die Auflistung des Javascripts Codes der Gauge Klasse wird hier verzichtet, da es keine Abhängigkeiten zu Lift aufweist. Kurzum bedient sich die Gauge Klasse dem HTML5 Canvas Element um den Tachometer zu zeichnen.

### 5.5.3 Fazit

Widgets sind in Lift schnell eingefügt bzw. erstellt. Auch bei kleinen Widgets übersteigt der Javascript Anteil schnell den zu implementierenden Anteil in Lift. Demnach sind für die Erstellung eines Widgets eher Javascript / HTML als Scala / Lift Spezialisten gefragt. Die fast perfekte Trennung zwischen Javascript und Lift macht es auch für einen Nicht-Lift Programmierer möglich, ein Widget für Lift zu erstellen.

## 5.6 Ajax & Comet

---

Der erste Teil dieses Kapitels beschäftigt sich mit den von Lift zur Verfügung gestellten Ajax Funktionen. Im zweiten Teil wird Comet näher betrachtet.

### 5.6.1 Ajax

Lift bietet eine umfangreiche Unterstützung für das Erstellen von Rich Client Web Applikationen mit Hilfe von Ajax. Dazu bietet Lift sog. „Ajax Generatoren“ für das Erstellen von Ajax Komponenten, sowie einen Abstraktionslayer für die Generierung von clientseitigem JavaScript. Ajax Anfragen werden aus durch Lift zur Verfügung gestelltem JavaScript realisiert. Diese Bibliotheken sind ebenfalls über ein Abstraktionslayer konfigurierbar. Benötigte JavaScript Bibliotheken werden automatisch in Templates eingebunden, wenn deren Funktionen im Code dahinter verwendet werden. Standardmäßig arbeitet Lift mit JQuery (30). Durch den Abstraktionslayer ist aber die Verwendung anderer JavaScript Bibliotheken möglich. Die Ajax Unterstützung in Lift ist sehr flexibel und ermöglicht es dem Entwickler Rich Client Web Applikationen zu erstellen, ohne direkten JavaScript Code schreiben zu müssen.

#### Ajax Generatoren

Über das SHtml Objekt wird eine Vielzahl von Ajax Generatoren zur Verfügung gestellt. Ajax Generatoren liefern HTML Elemente wie Links oder Buttons welche mit Ajax Funktionalität angereichert werden können. Dies geschieht über Callback Methoden, welche ein Objekt vom Typ JsCmd zurückgeben. Diese Methoden können an ein HTML Element gehängt werden. Beim Ausführen können Sie ein JsCmd Objekt zurückgeben, welches für das clientseitige Update verwendet werden kann. Das folgende Codelisting 42 zeigt die Verwendung eines Ajax Generator und das Anfügen einer Callback Methode für ein Client Update.

---

```
def myFunc(html: NodeSeq): NodeSeq = {
    def ajaxTest(msg: String): JsCmd = {
        println("server side processing...")
        JsRaw("alert('" + msg + "')").cmd
    }
    bind("ajax", html,
        "button" -> SHtml.ajaxButton("Ajax test", ajaxTest("hello world"))
    )
}
```

---

Codelisting 42: Verwendung von Ajax Generator mit Callback Methode

In diesem Beispiel erzeugt die Methode SHtml.ajaxButton einen HTML Button. Die Funktion ajaxTest wird als Callback Methode mitgegeben. Beim Klicken des Buttons wird diese ausgeführt. In dieser Funktion kann die serverseitige Logik implementiert werden. Als Rückgabewert wird ein JsCmd erwartet. Ein JsCmd ist eine Abstraktion von JavaScript Aktionen. Der zurückgegebene JsCmd wird anschliessend clientseitig ausgeführt. Es kann auch ein leeres JsCmd Objekt (Noop) zurückgegeben werden falls

keine clientseitige Aktualisierung notwendig ist. Das JsRaw Objekt wird verwendet um direkt JavaScript Code auszuführen. Das Beispiel bindet das <ajax:button/> Tag einer HTML Seite mit einem Button welcher beim Klicken eine Dialogbox mit „hello world“ als Inhalt anzeigt.

Wie bereits erwähnt, stellt das SHtml Objekt Ajax Generatoren für die unterschiedlichen HTML Elemente zur Verfügung. Für eine komplette Liste der Generatoren kann die API Dokumentation (25) verwendet werden.

## Ajax Formulare

Wenn HTML Formulare über Ajax verarbeitet werden sollen, müssen einige Dinge beachtet werden.

- **Das Formular muss als Ajax Formular definiert werden**

Der Markup Code aus Templates welche ein Formular repräsentieren, muss als Ajax Form definiert werden. Ein solches Formular soll nicht per gewöhnlichem GET oder POST Request, sonder über Ajax Calls weiterverarbeitet werden. Ein NodeSeq welcher ein Formular repräsentiert kann über die Methode SHtml.ajaxForm als Ajax Formular markiert werden. Es können Callack Methoden für Pre- & Post Form Submission mitgegeben werden.

- **Ajax Submit Button**

Für ein Ajax Formular wird kein normaler Submit Button, sondern eine Ajax Submit Button verwendet. Ein solcher wird über SHtml.ajaxSubmit erstellt. Dieser kann mit einer Callback Methode für den Submit Event versehen werden.

- **Verarbeitungslogik und clientseitige Aktualisierung**

Dem Submit Button wird eine Funktion angegeben. Diese übernimmt die serverseitige Verarbeitung des Formulars und gibt ein JsCmd für die clientseitige Aktualisierung zurück.

- **Weitere Ajax Formular Elemente**

Typischerweise werden in einer dynamischen Ajax Form noch weitere HTML Komponenten mit Ajax angereichert. Dazu gehören Textfelder, Checkboxen, Drop-Down Menüs etc. Diese können über das SHtml Objekt generiert und ebenfalls mit Callback Logik ausgerüstet werden.

Das Codelisting 43 zeigt Markup Code für ein einfaches Ajax Formular. Es wird die render Methode des SimpleAjaxForm Snippet aufgerufen. Für gewöhnliche Formulare würde das Snippet Tag mit form="POST" attributieren. Dies ist hier nicht notwendig da Lift im Snippet Code erfährt, wie das Formular behandelt werden soll.

---

```
<lift:surround with="default" at="content">
  <div id="yourName">Please enter your name:</div>
  <lift:SimpleAjaxForm>
    <ajax:name/>
    <ajax:submit/>
  </lift:SimpleAjaxForm>
</lift:surround>
```

Codelisting 43: Markup Code für simples Ajax Formular

Der entsprechende Snippet Code für dieses Formular folgt in Codelisting 44.

---

```
def render(html: NodeSeq): NodeSeq = {
  def setName(name: String): JsCmd = {
    println("name was set to:" + name)
    val msg: NodeSeq = <span>your name is:
      <b>
        {name}
      </b>
    </span>
    JqSetHtml("yourName", msg)
  }
  val name = ""
  bind("ajax", SHtml.ajaxForm(html),
    "name" -> SHtml.text(name, name = _),
    "submit" -> SHtml.ajaxSubmit("Whats my name?", setName(name)))
}
```

Codelisting 44: Snippet Code für simples Ajax Formular

Am Beispiel werden die vorher beschriebenen Punkte umgesetzt. Die `setName` Funktion gibt für die clientseitige Aktualisierung ein `JqSetHtml` Objekt zurück. Durch dieses wird das `<div>` mit der angegebenen id mit dem Inhalt von `msg` überschrieben. Dem User wird ein direktes Feedback über seine Eingabe gegeben.

## JavaScript Unterstützung

Die JavaScript Abstraktion in Lift ermöglicht es dem Entwickler JavaScript Funktionalität in Scala zu implementieren. Somit wird das Separieren von Darstellung und Logik unterstützt. Lift stellt eine Abstraktionsebene für die JavaScript Bibliotheken jQuery (30), YUI (31) und ExtCore (32) zur Verfügung.

Lift stellt eine Klassenhierarchie für JavaScript Ausdrücke zur Verfügung. Der bereits häufig erwähnte `JsCmd` Trait repräsentiert einen JavaScript Befehl. Es gibt auch noch ein `JsExp` Trait welcher implizit in ein `JsCmd` konvertiert wird. Für die normale Entwicklung ist der Unterschied zwischen `JsCmd` und `JsExp` deshalb nicht von Bedeutung.

Um einen Überblick der JavaScript Unterstützung in Lift zu bekommen, konsultiert man am besten die Objekte `net.liftweb.http.js.JsCmds` und `JE` der Lift API (25). Diese beiden Objekte stellen einen grossen Teil an Funktionalität zur Verfügung.

Im folgenden Codelisting 45 wurde das Beispiel aus Codelisting 44 um eine zusätzliche JavaScript Funktion erweitert welche eine einfache client-seitige Validierung durchführt.

---

```

def render(html: NodeSeq): NodeSeq = {
  def setName(name: String): JsCmd = {
    println("name was set to:" + name)
    val msg: NodeSeq = <span>your name is:
      <b>
        {name}
      </b>
    </span>
    JqSetHtml("yourName", msg)
  }
  def validation(name: String) = {
    JsIf(name == "") {
      Alert("Please enter your name")
    }
  }
  val name = ""
  bind("ajax", SHtml.ajaxForm(html),
    "name" -> SHtml.ajaxText(name, validation),
    "submit" -> SHtml.ajaxSubmit("Whats my name?", setName(name)))
  )
}

```

---

Codelisting 45: Snippet mit JavaScript Funktionalität

Das SHtml.ajaxForm Textfeld ruft die angegebene Callback Methode beim Verlassen auf und übergibt ihr den eingegebenen String. Diese gibt ein JsCmd zurück. Die Validierung geschieht in diesem Beispiel über JavaScript Clientseitig. Falls nichts eingegeben wurde wird eine Dialogbox mit einer entsprechenden Meldung angezeigt.

Wie bereits erwähnt, verwendet Lift standardmäßig jQuery als erweiterte JavaScript Bibliothek. Eine detaillierte Beschreibung wie von jQuery nach YUI gewechselt werden kann, befindet sich in Kapitel 8 des Lift Buches (8). Die Anbindung einer externen JavaScript Bibliothek geschieht über das JSArtifacts Trait, welches eine konkrete Implementation für eine spezifische Bibliothek erfordert. Die Unterstützung von jQuery in Lift geht aber noch über das JSArtifacts Trait hinaus. Die beiden Singleton Objects JqJE und JqJsCmds stellen eine Vielzahl von Funktionen zur Verfügung. Für eine komplette Übersicht ist die API Dokumentation (25) der entsprechenden Objects zu konsultieren.

Im Codelisting 46 wird eine Anwendung der DisplayMessage Funktion, aus der jQuery Bibliothek beispielhaft aufgezeigt. Das Snippet AjaxExample implementiert dabei die Methode render.

---

```

def render(html: NodeSeq): NodeSeq = {
  def txtmsg(msg: String) =
    SHtml.ajaxText("", 
      v => JqJsCmds.DisplayMessage("msgdiv",
        "You entered " + msg + " in the text box",
        4 seconds, 1 second))
  bind("ajax", html,
    "text" -> txtmsg,
    "msgdiv" -> <div id="msgdiv"></div>)
}

```

---

Codelisting 46: Verwendung von jQuery Funktionalität

Der entsprechende Markup Code für das Template wird im folgenden Codelisting 47 gezeigt.

```
<lift:surround with="default" at="content">
  <lift:AjaxExample>
    <p>Please enter some text:</p>
    <ajax:text/>
    <ajax:msgdiv/>
  </lift:AjaxExample>
</lift:surround>
```

Codelisting 47: Markup Code für jQuery Snippet

Der Code zeigt ein Textfeld an, welches seinen Inhalt beim Verlassen in ein angegebenes div projiziert. Die Nachricht bzw. das div wird für 4 Sekunden angezeigt und dann während einer Sekunde fliessend ausgeblendet.

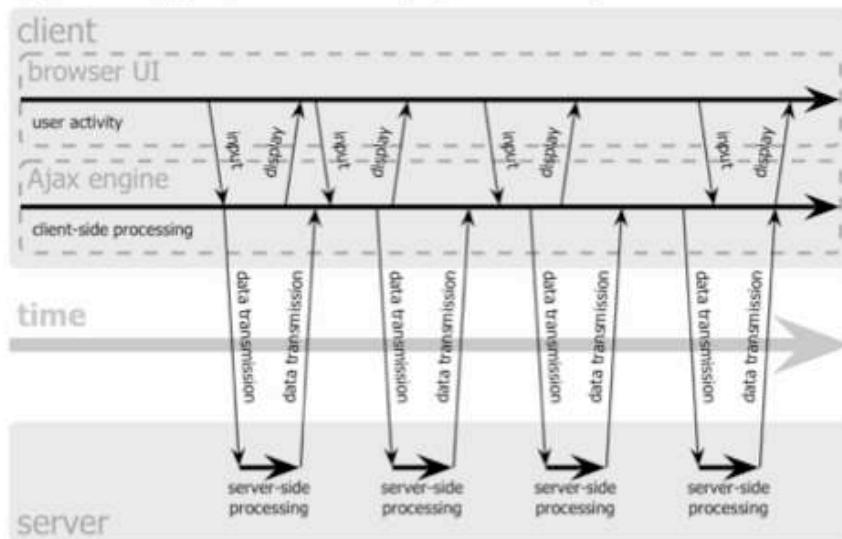
## 5.6.2 Comet

Während sich Ajax für clientseitig Aufrufe eignet, ist die Technologie für serverseitig ausgelöste Events kaum bzw. nur bedingt brauchbar. Durch endloses Polling in zeitlich kurzen Abständen sind zwar ähnliche Effekte wie bei Comet erreichbar, diese haben allerdings den Nachteil, unnötig Ressourcen zu verschwenden – dies sowohl auf der Client- als auch auf der Serverseite.

Comet macht sich die Funktion zu Nutze, Anfragen eines Clients nicht schnellstmöglich zu beantworten, sondern die Antwort so lange hinauszuzögern, bis auch eine Antwort vorliegt. Dadurch können neue Ereignisse sofort zum Client gepusht werden, welcher nach Erhalt gleich wieder eine Comet Verbindung zum Server aufbaut und auf eine erneute Antwort wartet.

Abbildung 6 zeigt die unterschiedliche Funktionsweise von Ajax und Comet anhand eines Diagrammes (33).

### Ajax web application model (asynchronous)



### Comet web application model

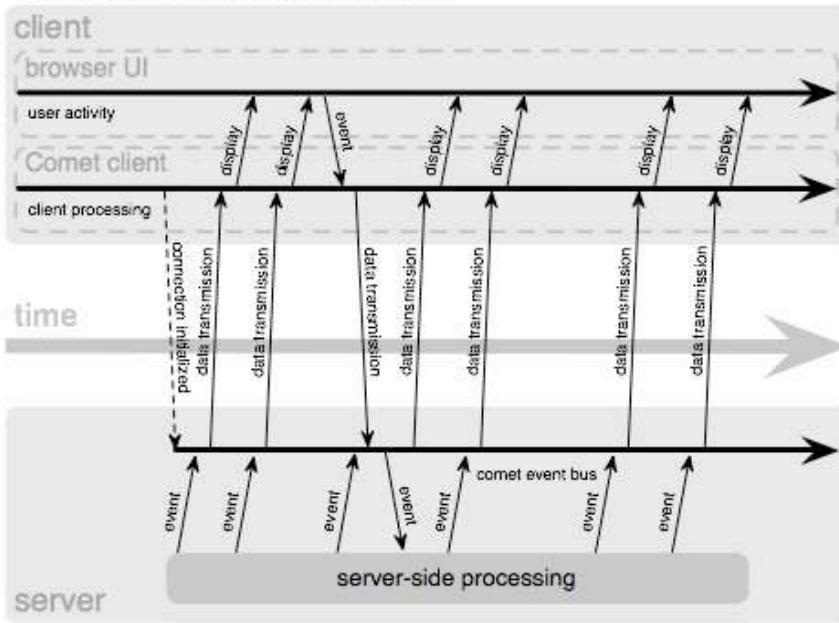


Abbildung 6: Ajax vs. Comet

Im folgenden Abschnitt wird Comet schrittweise implementiert. Dazu wird die Implementierung der Beispielapplikation verwendet und anschaulich erklärt. Abbildung 7 zeigt die Übersicht einer beispielhaften Abfolge von Events.

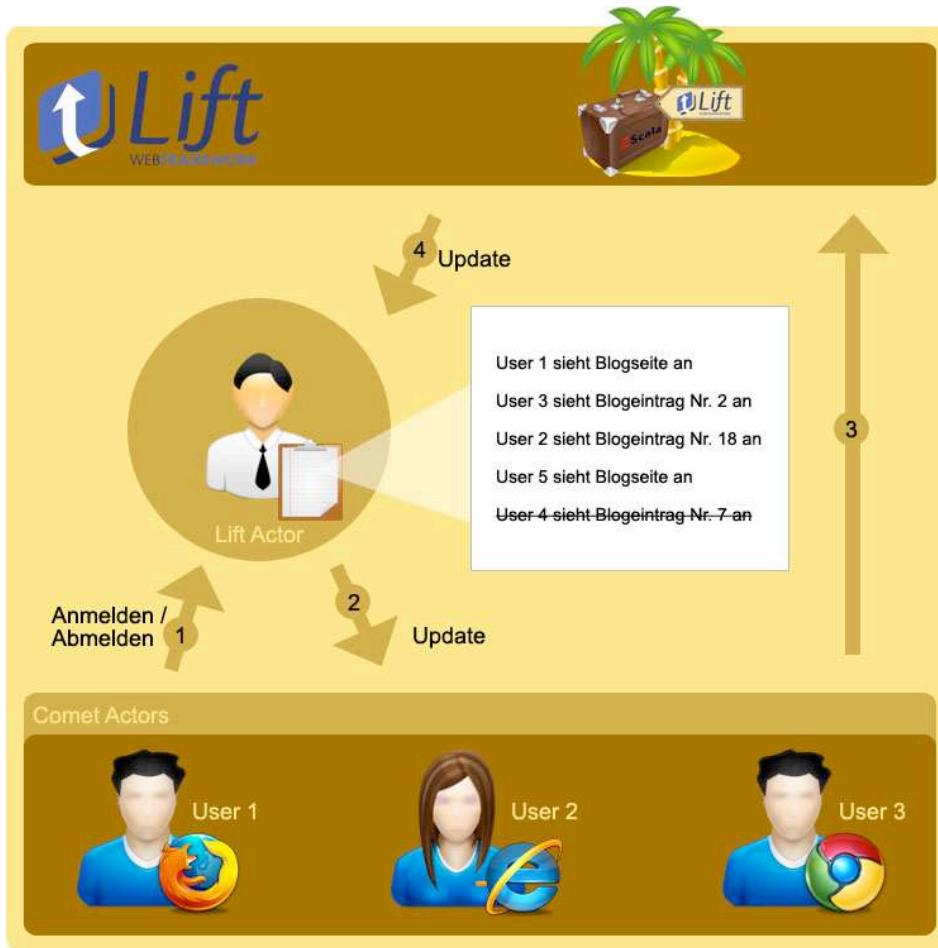


Abbildung 7: Comet, Übersicht

Die Akteure in unserem Comet Beispiel sind einerseits die Comet Actors, welchen je ein Benutzer in einer Lift Session gegenübersteht und andererseits der Lift Actor und die Beispielapplikation als Website, welche durch das Palmen Logo visualisiert ist. Selbstverständlich sind alle Akteure zur Laufzeit Objekte in der Beispielapplikation. Die Aufteilung soll lediglich zu einem besseren Verständnis führen.

Die nummerierten Pfeile in Abbildung 7 zeigen einen grundlegenden Ablauf in aufsteigender Reihenfolge. Vorerst melden sich Comet Actors beim Lift Actor durch den Besuch einer Website an (1). Der LiftActor händigt darauf ein erstes Update aus, welches für eine erste Darstellung der Website benötigt wird (2). Durch Interaktionen der CometActors mit der Website, wie z.B. ein Hinzufügen eines Blogeintrages (3), meldet die Webapplikation eine Änderung der Blogseite an den Lift Actor weiter (4). Der Lift Actor weiss nun genau, welchen Comet Actors er dieses Update auszustellen hat, da eine aktuelle Liste aller aktiven Comet Actors besitzt (2).

Der Ablauf kann auch als Kreislauf betrachtet werden (3 -> 4 -> 2) welcher einen Ein- und Ausgangspunkt besitzt (1).

Der soeben geschilderte Ablauf wird nun schrittweise durch Codelistings aufgezeigt.

Codelisting 48 zeigt den kleinstmöglichen Ausschnitt einer Rendering Seite, welche dazu führt, dass eine Instanz der ExampleCometActor Klasse erstellt wird.

---

```
<lift:comet type="ExampleCometActor">
</lift:comet>
```

---

Codelisting 48: Comet, Rendering

Codelisting 49 zeigt eine stark gekürzte Fassung der in der Beispielapplikation vorhandenen DynamicBlogViews Klasse, welche wir hier einfacheheitshalber ExampleCometActor nennen. Comet Klassen müssen sich hierbei im comet Package einer Lift Applikation befinden.

---

```
class ExampleCometActor extends CometActor {
    var blog: List[BlogEntry] = Nil
    var comments: List[Comment] = Nil

    def render = {
        // Markup Code der sich zwischen den <lift:comet> Tags befindet,
        // kann in dieser Methode durch Verwendung von defaultXml gebunden
        // werden.
    }

    // Diese Methode wird gleich nach der Instanzierung aufgerufen
    override def localSetup {
        (ExampleLiftActor.list !? AddBlogWatcher(this)) match {
            case BlogUpdate(entries) => this.blog = entries
        }
    }

    // Diese Methode wird aufgerufen, sobald der Comet Actor für längere
    // Zeit inaktiv ist
    override def localShutdown {
        ExampleLiftActor.list ! RemoveBlogWatcher(this)
    }

    // Diese Methode stellt den Eingangskanal dar, durch welchen der
    // Lift Actor den Comet Actor bei Neuigkeiten informieren kann
    override def lowPriority: PartialFunction[Any, Unit] = {
        case BlogUpdate(entries: List[BlogEntry]) => this.blog = entries;
        reRender(false);
        case CommentUpdate(entries: List[Comment]) => {
            this.comments = entries;
            partialUpdate(JqSetHtml("blog_comments", renderComments))
        }
    }
}
```

---

Codelisting 49: Comet, ExampleCometActor.scala

Bei der Instanziierung einer Cometklasse wird zuerst die localSetup Methode aufgerufen. Diese meldet sich sogleich beim Lift Actor an und wartet auf ein erstes Update. Dies ist in unserem Falle wichtig, da beim späteren Aufruf der render Methode die Blogeinträge bereits vorhanden sein müssen.

Wie bereits erwähnt erfolgt nun der Aufruf der render Methode. Im Gegensatz zu Snippet Methoden besitzt die render Methode keinen NodeSeq Eingangsparameter, was für einen Einsatz von bind() aber zwingend benötigt wird. Um die mächtige bind() Funktion dennoch einzusetzen, speichert die Comet Actor Klasse den Markup Code, welcher sich zwischen den <lift:comet> Tags befindet, in ein privates Feld,

welches durch den Aufruf der öffentlichen Methode defaultXml ausgelesen werden kann.

Folgt ein Aufruf des Lift Actors wird automatisch die Methode lowPriority angesprungen. Nebst low gibt es noch die Prioritätsstufen middle und high. Zeitkritische Updates sollten in die high-, weniger zeitkritische Updates in die middle bzw. lowPriority Methode gesetzt werden.

Updates können auf zwei grundsätzlich verschiedene Ansätze behandelt werden. Zum einen kann die render Methode der Klasse erneut aufgerufen werden, was durch den Aufruf von reRender geschieht. Zum anderen können durch die Verwendung von partialUpdate gezielt Javascript Befehle ausgeführt werden. Je nach Anwendungsfall empfiehlt sich die eine oder andere Variante. In unserem Beispiel wird die reRender Methode für die Blögeinträge und die partialUpdate Methode für die Kommentare eines Blögeintrages verwendet.

Codelisting 50 zeigt einen vereinfachten Ausschnitt der Lift Actor Klasse in der Beispieldapplikation. Das Handling mehrerer Aktionen wie Ändern oder Löschen von Blögeinträgen wurde entfernt. Auch ist das komplette Kommentarhandling übersichtshalber entfernt worden. Die Klasse wurde deswegen von BlogCache in ExampleLiftActor umgetauft. Es ist üblich, dass Lift Actor Klassen in das controller Package einer Lift Applikation gesetzt werden.

---

```

class ExampleLiftActor extends LiftActor {

    private var cache: List[BlogEntry] = List()
    private var sessions: List[SimpleActor[Any]] = List()

    def getEntries(): List[BlogEntry] = DB.getBlogEntries
    def getComments(blogEntry: BlogEntry): List[Comment] =
        DB.getCommentFrom(blogEntry)

    protected def messageHandler =
    {
        case AddBlogWatcher(me) =>
            val blog = getEntries
            reply(BlogUpdate(blog))
            cache = cache :: blog
            sessions = sessions :: List(me)
        case AddEntry(e) =>
            cache = cache :: List(e)
            sessions.foreach(_ ! BlogUpdate(cache))
        case _ =>
    }
}

case class AddBlogWatcher(me: SimpleActor[Any])
case class AddEntry(entry: BlogEntry)

case class BlogUpdate(xs: List[BlogEntry])
case class CommentUpdate(xs: List[Comment])

object ExampleLiftActor {
    lazy val list = new ExampleLiftActor
}

```

---

Codelisting 50: Comet, ExampleLiftActor.scala

Wie bereits in Codelisting 49 aufgezeigt, registriert sich die Comet Actor Klasse gleich nach Instanziierung beim LiftActor. In der messageHandler Methode ist ersichtlich, dass der CometActor sowohl ein BlogUpdate mit einer aktuellen Liste aller Blogeinträge erhält, sowie auch vom LiftActor als aktiver Comet Actor gespeichert wird.

Wird dem Lift Actor nun signalisiert, dass ein neuer Blogeintrag erstellt wurde, wird dieser zur Liste hinzugefügt und darauf alle aktiven Comet Actors mit dieser neuen Liste versorgt. Die Singalisation erfolgt hierbei über das Scala Compagnon Objekt (object ExampleLiftActor). Dadurch ist es der gesamten Applikation jederzeit möglich auf den Lift Actor über Neuigkeiten zu informieren.

---

```
ExampleLiftActor.list ! AddEntry(newEntry)
```

---

Codelisting 51: Comet, Signalisierung des Lift Actors

---

#### Best practice



Lift sieht vor, dass für jeden Client genau ein Comet Actor existiert, welcher sich bei einem oder auch mehreren Lift Actors für Events an-/ abmeldet. Grund hierfür ist, dass ein Comet Actor von Lift pro Session und nicht pro Request erstellt wird.

## 5.7 REST

Dieses Kapitel beschäftigt sich mit REST (Representational State Transfer) und den Möglichkeiten von Lift, eine REST Schnittstelle zu implementieren. Nach einer kurzen Einführung in den REST Architekturstil wird im nächsten Abschnitt eine Implementation exemplarisch aufgezeigt.

### 5.7.1 Einführung

Der REST Architekturstil stammt aus der Dissertation von Roy Fielding, welche im Jahr 2000 erschien. Der Grundgedanke war ein Service zu entwerfen, der sich den HTTP-Request-Methoden bedient um aus einem eigentlichen Transportprotokoll ein Applikationsprotokoll zu formen. Dabei soll jede Ressource eine eindeutige URL besitzen. Aktionen auf diesen Ressourcen sollen durch die HTTP-Request-Methoden identifiziert werden. Die Ressource und die darauf auszuführende Aktion sind in diesem Verfahren sauber voneinander getrennt.

HTTP-Request-Methode	Bedeutung	Body nötig?	Identifikation der Ressource nötig?
GET	Lesen einer Ressource	Nein	Ja
POST	Erstellen einer Ressource	Ja	Nein
PUT	Ändern einer Ressource	Ja	Ja
DELETE	Löschen einer Ressource	Nein	Ja

Tabelle 13: Übersicht REST Operationen

Bis auf POST benötigt jede Request-Methode eine eindeutige URL. GET und DELETE benötigen im Gegensatz zu den Methoden POST und PUT keinen Body im HTTP-Request. Die URL in Verbindung mit der HTTP-Request-Methode reicht in diesem Falle aus.

Als Body im HTTP-Request wird häufig XML verwendet. Es aber auch denkbar eine andere Auszeichnungssprache wie z.B. JSON zu verwenden.

Viele RESTful programmierten Anwendungen bieten ihre Dienste sowohl in XML als auch in JSON an. Das Format der Serverantwort kann dabei vom Clienten gesteuert werden. Während einige REST - Dienste durch einen zusätzlichen URL Parameter die Formatentscheidung treffen, parsen andere den Accept Header des Clientrequests nach dem auszuliefernden Format. Oft sind bei einem REST - Service auch beide Varianten anzutreffen, wobei dann der URL Parameter eine höhere Priorität als der Accept Header besitzt.

## 5.7.2 REST Praxisbeispiel

Lift besass schon immer die Möglichkeit, den REST Service einfach und schnell einer Webapplikation hinzuzufügen. Seit dem 28.04.2010 ist dies durch eine erneuerte REST API noch einmal einfacher geworden. Die folgenden Schritte zeigen den neuen Weg REST zu implementieren.

### 1. Erstellung des REST Objects

Es empfiehlt sich, das in Codelisting 52 gezeigte Object in ein separates Package zu speichern. Ein oft gewählter Packagename ist hierfür „api“. Die Klasse RestHelper stellt die Basisfunktionalität zur Verfügung, die verschiedenen eingehenden Requests abzuhandeln. Sie enthält Typen für das Matching und ist für das Extrahieren des Request – Bodys und die Konvertierung des Rückgabewerts zuständig.

---

```
object RestAPI extends RestHelper {

    // implizite Konvertierungsmethode //
    implicit def cvt: JxCvtPF[Object] = {
        case (JsonSelect, o: MyObject, _) => o.toJson
        case (XmlSelect, o: MyObject, _) => o.toXml
    }

    serve { /* URL Matching */ }
    serveJx { /* URL Matching mit impliziter Kovertierung */ }
}
```

---

Codelisting 52: REST, RestAPI Object

Eine Erläuterung des Codes folgt in einem späteren Abschnitt, in welchem die Funktionsweise der Helper-Methoden serve und serveJx detailliert vorgestellt werden.

### 2. RestAPI in Boot.scala hinzufügen

---

```
LiftRules.dispatch.append(RestAPI)
```

---

Codelisting 53: REST, Boot.scala

Durch diese Schritte ist die Umgebung für einen REST - Service bereits geschaffen und lauffähig. Da der REST – Service bisher keine Anfragen entgegennimmt wird in den folgenden Ausschnitten nun die mögliche URL Matchingszenarien aufgezeigt und die Verwendung von serve / serveJx ausgeleuchtet.

## Helper-Methode serve

Die Helper – Methoden serve und serveJx bilden das Bindeglied zwischen eingehenden Requests und der Antwort aus der Businesslogik. Eingehende Requests werden gematcht und darauf eine Antwort des Typs LiftResponse zurückgegeben.

---

```

def createBlogEntry(xml: Node) = {
  val e = BlogEntry.fromXML(xml).save()
  e.toXML
}

serve {
  case "api" :: "blog" :: Nil XmlPost xml => _ =>
  createBlogEntry(xml)
}

```

---

Codelisting 54: REST, RestAPI, serve

Codelisting 54 zeigt einen möglichen Ansatz, wie ein POST – Request auf die URL /api/blog/ mit XML im Request Body verarbeitet werden kann. Die mitgelieferten XML Daten werden extrahiert und der Methode createBlogEntry übergeben, welche aus den XML Daten ein Objekt erzeugt, speichert und es in XML Form zurückgibt. Der Rückgabewert der createBlogEntry Methode wird dabei als Antwort auf den Client Request angezeigt. Wichtig ist, dass dieser Rückgabewert ein Objekt des Typs LiftResponse zurückgibt oder ein Objekt, welches von Lift automatisch in LiftResponse konvertiert werden kann. In diesem Falle wurde mit XML angefragt, weshalb die Antwort ebenfalls in XML erfolgt.

Nebst XmlPost implementiert die Klasse RestHelper folgende RESTlichen Methoden: XmlGet, XmlPut und XmlDelete. Falls JSON anstatt XML entgegengenommen werden will, existieren auch die Methoden JsonGet, JsonPost, JsonPut und JsonDelete. Da die Methoden ebenfalls in die Matchingkriterien mit einfließen, ist es kein Problem diese parallel anzubieten, da Lift automatisch detektiert, ob es sich bei den mitgelieferten Daten um XML oder JSON handelt. Codelisting 55 macht dies deutlich.

---

```

serve {
  case "api" :: "blog" :: Nil XmlPost xml => _ =>
  createBlogEntry(xml)
  case "api" :: "blog" :: Nil JsonPost json => _ =>
  createBlogEntryJson(json)
}

```

---

Codelisting 55: REST, RestAPI, serve, XML vs. JSON

## Helper-Methode serveJx

Falls der REST Service sowohl XML als auch JSON unterstützen soll, empfiehlt sich die Verwendung von serveJx, welche im Unterschied zu serve die Unterscheidung von XML zu JSON in die in Codelisting 52 gezeigte Methode auslagert und diese per implizite Konvertierung aufruft.

---

```

serveJx {
  case Get("api" :: "blog" :: entry :: Nil, _) =>
  Full(MyObject(entry))
}

```

---

Codelisting 56: REST, RestAPI, serveJx

Das Statement Full(MyObject(entry)) in Codelisting 56 bewirkt nun, dass das MyObject der Methode cvt, welche in Codelisting 52 ersichtlich ist, übergeben wird. In dieser wird nun mittels case Matching überprüft, ob der Aufruf per XML oder JSON stattgefunden hat und das übergebene Objekt überhaupt behandelt werden soll. Die

Anzahl Codezeilen können durch serveJx beinahe halbiert werden da nun für XML und JSON lediglich die Operationen GET, POST, PUT und DELETE definiert werden müssen.

## Prüfung von Parametern mit Scala Extractors

In den vorangegangen Abschnitten sind wir davon ausgegangen, dass die URL korrekt aufgerufen wird und immer ein Rückgabewert vorliegt. Natürlich ist dem nicht immer so, weshalb zusätzliche Prüfungen nötig sind, z.B. ob eine Ressource überhaupt existiert.

### Hinweis



Als Extractor werden Objekte bezeichnet, welche eine Methode in der Form def unapply(str: String) : Option[MyObject] besitzen. Die Methoden apply (injection method) und unapply() (extraction method) sind Gegenpaare, heben sich bei verschachteltem Gebrauch also auf. Da sowohl apply() als auch unapply() bei Objektklammerangaben automatisch aufgerufen werden könnten, ist der Kontext das entscheidende Kriterium, welche Methode schlussendlich ausgeführt wird. Im Falle von case Matching wird automatisch unapply() aufgerufen und gegen Some bzw. None geprüft.

Ein Ansatz ist, diese Überprüfung in die aufgerufene Methode zu setzen und die Entscheidung dort zu treffen. Lift bietet vorgefertigte Extractor-Methoden an, mit welchen bereits im Matching geprüft werden kann. Codelisting 57 veranschaulicht dies, indem die URL nun zwingend eine Zahl, die nach Long konvertiert werden kann, beinhalten muss. Ist dem nicht so, wird der Matchingvorgang abgebrochen und ein HTTP-Response-Code 404 zurückgegeben.

```
serveJx {
    case Get("api" :: "blog" :: AsLong(entry) :: Nil, _) =>
    Full(MyObject(entry))
}
```

Codelisting 57: REST, Lift Extractor

Durch die Erstellung eines eigenen Extractors kann die Validierung noch feiner erfolgen. Codelisting 58 zeigt ein Extractor aus der Beispielapplikation, welcher überprüft, ob die Ressource BlogEntry existiert.

```
object AsBlogEntry {
    def unapply(in: String): Option[BlogEntry] = {
        Model.find(classOf[BlogEntry], toLong(in))
    }
}

serveJx {
    case Get("api" :: "blog" :: AsBlogEntry(entry) :: Nil, _) =>
    Full(MyObject(entry))
}
```

Codelisting 58: REST, eigener Lift Extractor

Die Prüfung erfolgt hierbei auf den Option Rückgabewert des Extractors. Falls die Option kein Ergebnis enthält, wird das Matching abgebrochen und die HTTP Anfrage in einem HTTP-Response-Code 404 enden.

## Prüfung von Abhängigkeiten zwischen den Parametern mit Scala Guards

Mit Extractors haben wir eine elegante Möglichkeit kennengelernt, Eingabeparameter in der URL bereits im Matching zu prüfen. Was ist aber nun, falls die Eingabeparameter nicht einzeln, sondern in Abhängigkeit zueinander geprüft werden sollen? Hier kommen die Guards ins Spiel.

```
serveJx {  
    case Get("api" :: "blog" :: AsBlogEntry(entry) :: "comment" ::  
            AsComment(comment) :: Nil, _) if entry.comments.contains(comment) =>  
        Full(comment)  
}
```

Codelisting 59: REST, Guard

Codelisting 59 zeigt ein Codeausschnitt, mit welchem geprüft wird, ob sowohl der Blogeintrag als auch der Kommentar existiert. Mit dem zusätzlichen Guard „if entry.comments.contains(comment)“ kann nun geprüft werden, ob der Kommentar auch wirklich zum Blogeintrag gehört. Die Prüfung findet dabei nicht im sondern gleich nach dem Matching statt. Dies ist nötig da nur nach dem Matching auf beide Parameter zugegriffen werden kann. Trotz diesem Umstand ist das Verhalten der Lift Applikation dasselbe, wie bei einem fehlgeschlagenen Matching: Schlägt der Guard fehl, wird dem Client ein 404 HTTP-Response-Code zurückgegeben.

## 6 Lift Best Practices und HowTos

---

In diesem Kapitel werden einige Best Practices für Lift beschrieben welche für die Entwicklung mit Lift hilfreich sind. Lift verfolgt einige spezielle Konzepte welche für das produktive Entwickeln von Webapplikationen hilfreich sind. Damit diese Konzepte auch entsprechend verwendet werden können, ist es hilfreich, gewisse Probleme mit den beschriebenen Best Practices zu lösen.

Weiter werden in diesem Kapitel einige übliche Probleme bzw. Aufgaben, welche bei der Entwicklung einer Webapplikation auftreten, in Form von kleinen HowTos beschrieben. Diese HowTos dienen als Anleitung, wie ein gewisses Problem in Lift gelöst wird. Die Auswahl der HowTo Themen setzte sich aus Anwendungen der Beispielapplikation zusammen. So sind die Lösungen von technischen Problemen, welche nicht spezifische in den technischen Aspekten dokumentiert wurden, in dieser Sektion zu finden.

Das Kapitel dient dem Zweck, den Entwickler bei der Verwendung von Lift zu unterstützen. Die dokumentierten Informationen sind hilfreich, wenn man mit der Entwicklung mit Lift beginnen möchte. Weiter helfen sie beim Aufbau des Grundverständnisses von Lift.

## 6.1 Wichtige Klassen im Überblick

Als Einstieg in dieses Kapitel listen wir hier einige, häufig verwendete Lift Klassen / Objects auf.

### 6.1.1 S Object

Das S Object beinhaltet Methoden um mit eingehenden Requests arbeiten zu können. Des weiteteren bietet sie Zugriff auf die Lift Session.

Methode	Verwendungszweck	Beispiel
S.?	Auslesen von eigens definierten resources Bundles für Internationalisierung.	S.?("key")
S.??	Auslesen von Lift Internationalisierungs Bundles	S.??("key")
S.attr	Zugriff auf Attribute in Template Lift Tags	S.attr("attribute")
S.findCookie	Cookie finden	S.findCookie("name")
S.addCookie	Cookie zur Response hinzufügen	S.addCookie(cookie)
S.param	HTTP POST bzw. GET Parameter auslesen	S.param("id")
S.session	Gibt die Liftsession in einer Box zurück. Hilfreich um alle Comet Actors zu finden.	
S.addAround	Nimmt Funktionen entgegen, welche vor bzw. nach einem Request ausgeführt werden sollen.	

Tabelle 14: S Object

### 6.1.2 SHtml Object

Das SHtml Object beinhaltet alle HTML Komponenten um ein Formular erstellen zu können. Dabei unterscheidet SHtml zwischen normalen HTML Elementen und solchen mit einer Ajax Anbindung. Tabelle 15 zeigt die meistverwendeten HTML Formular Elemente. Beinahe alle besitzen eine Ajax Variante, welche gut am ajax Präfix im Methodennamen erkennbar sind.

Methode	Verwendungszweck
<b>SHtml.link</b>	Generierung eines Links in Form <a href="link">name</a>
<b>SHtml.checkbox</b>	Erstellung einer Checkbox <input type="checkbox" name="name" /> Beschreibung
<b>SHtml.area</b>	Erstellung einer Textarea <textarea></textarea>
<b>SHtml.hidden</b>	Erstellung eines Hidden Feldes <input type="hidden" name="name" value="x" />
<b>SHtml.text</b>	Erstellung eines Textfeldes <input type="text" name="name" />
<b>SHtml.password</b>	Erstellung eines Passwortfeldes <input type="password" name="name" />
<b>SHtml.select</b>	Erstellung einer Auswahl <select name="name"></select>
<b>SHtml.radio</b>	Erstellung einer Radio Auswahl <input type="radio" name="name" value="value"> Beschreibung
<b>SHtml.submit</b>	Erstellung eines submit Buttons <input type="submit" name="name">

Tabelle 15: SHtml Object

## 6.2 Objektübergabe zwischen Requests

### 6.2.1 Problemstellung

Ein klassisches Problem bei der Entwicklung von Webapplikationen ist die Verwaltung von Domain Entitäten, mittels der sogenannten CRUD Operationen. Meistens hat man eine Menge von Entitäten, welche erstellt (Create), gelesen (Read), bearbeitet (Update) und gelöscht (Delete) werden können. Die einzelnen Operationen werden typischerweise auf unterschiedlichen HTML Seiten implementiert. Das Problem ist die Übergabe der Referenz auf die entsprechende Domain Entität von Seite zu Seite.

### 6.2.2 Snippet und Requestvar

Die Bearbeitung der Domain Entitäten erfolgt in Snippet Methoden, welche in den HTML Seiten mittels Lift Tags entsprechend eingebunden werden. Für jeden Aufruf eines Snippets wird dieses neu initialisiert, ausser man verwendet Stateful Snippets. Man hat die Möglichkeit sog. RequestVars zu definieren. Diese sind für einen ganzen HTTP-Request lang gültig. Somit sind beim Wiedereintritt in ein Snippet die Werte immer noch vorhanden.

Das folgende Codelisting 60 illustriert eine mögliche Lösung für das Bearbeiten einer Domain Entität.

---

```
// Globale (für inter-Snippet Parameterübergabe) RequestVar mit Domain
// Entität
object blogEntryVar extends RequestVar[BlogEntry](new BlogEntry())

def editBlogEntry(entry: BlogEntry) {
    // innere Funktion für Bearbeitung
    def doEdit() = {
        // Objektreferenz wieder holen
        val editedEntry = blogEntryVar.is
        // weiteres handling mit editedEntry z.B. DB einfügen etc.
    }
    // Referenz auf den aktuellen BlogEntry
    val cE = blogEntryVar.is

    // Binding des Formular
    bind("entry", html,
        // Der aktuelle Inhalt des Entry wird in das Textfeld gebunden
        // Beim Submit werden die neuen Werte in den Entry geschrieben
        "title" -> SHtml.text(cE.title, currentEntry = _),
        "content" -> SHtml.textarea(cE.content, cE = _),
        // Beim Submit wird das übergebene Function Object zu einem
        // Closure welches die freie Value cE „einfängt“ und bindet.
        "submit" -> SHtml.submit("Save", () => {blogEntryVar(cE); doEdit}))
}

```

---

Codelisting 60: Domain Entität bearbeiten

Dieses Beispiel zeigt wie das beschriebene Verfahren Umgesetzt werden kann. Natürlich gibt es noch weitere Möglichkeiten dasselbe Resultat zu erzielen. Die Methode mit den RequestVar hat sich aber etabliert und kann vielfältig eingesetzt werden.

## 6.2.3 Parameterübergabe zwischen Snippets

Im obigen Codebeispiel wurde die RequestVar zur Sicherung der Entität global (d.h. ausserhalb der Klasse) deklariert. Dies ist sinnvoll da so der Zugriff auch aus anderen Snippets heraus erlaubt ist. Dies ist dann notwendig wenn die Entitäten irgendeine Fremdschlüsselbeziehung aufweisen. Dabei ist zu beachten das bei dem Auslösen der entsprechenden Aktionen (erstellen, bearbeiten, löschen, ...) vorher die RequestVar für die gewünschte Entität auch gesetzt wird.

Das folgende Codelisting 61 veranschaulicht dies.

---

```
// Globale (für inter-Snippet Parameterübergabe) RequestVar mit
// Domain Entität
object blogEntryVar extends RequestVar[BlogEntry](new BlogEntry())

class BlogSnippet {
  def showBlogEntry(html: NodeSeq): NodeSeq = {

    // erstellen eines Closures welches bei wiedereintritt in
    // das Snippet immer noch gültig ist
    val currentEntry = blogEntryVar.is
    val currentComment = commentVar.is

    // Binding des Formular
    bind("entry", html,
      "title" -> Text(currentEntry.title),
      // Textfeld für Kommentar, wird direkt an Entität
      // gebunden
      "comment" -> SHtml.textarea(c.content, c.content = _)
      // Beim Submit wird aus dem Function Object ein Closure welches
      // auf die freien Variablen currentEntry und currentComment
      // Zugriff hat.
      "addComment" -> SHtml.link("comment/new ", () => {
        blogEntryVar(currentEntry);
        commentVar(currentComment);
      }, "Add Comment"))
  }

  object commentVar extends RequestVar[Comment](new Comment())
}

class CommentSnippet {
  def addComment(html: NodeSeq): NodeSeq = {
    val currentEntry = BlogEntryVar.is
    val newComment = CommentVar.is
    currentEntry.Comments.add(newComment)
    newComment.blogEntry = currentEntry
    // Binding für HTML Formular etc...
  }
}
```

---

Codelisting 61: Parameterübergabe zwischen Snippets mit RequestVars

Das Beispiel zeigt, wie die global deklarierten RequestVar die Übergabe von Parametern zwischen Snippets einfach gestalten.

**Warnung**

Snippets welche von Parameter abhängen sollten auch ein Verhalten für nicht vorhandene Parameter definieren. Ansonsten können sie zu unerwünschtem Fehlverhalten führen. Im Codebeispiel geschieht dies über die Initialisierung der RequestVars mit leeren Entitäten bei der Erstellung.

Der Ausschnitt zeigt die beiden Snippets BlogSnippet und CommentSnippet mit je einer Methode. Wenn einem Blog Eintrag ein Kommentar hinzugefügt werden möchte, muss die Methode addComment des CommentSnippet aufgerufen werden. Der Methode muss der aktuelle Blog Eintrag übergeben werden, damit diese die Verbindung zwischen dem Kommentar und dem Blog Eintrag korrekt setzen kann.

Abbildung 8: Ablauf der Parameterübergabe illustriert die einzelnen Schritte der Parameterübergabe grafisch.

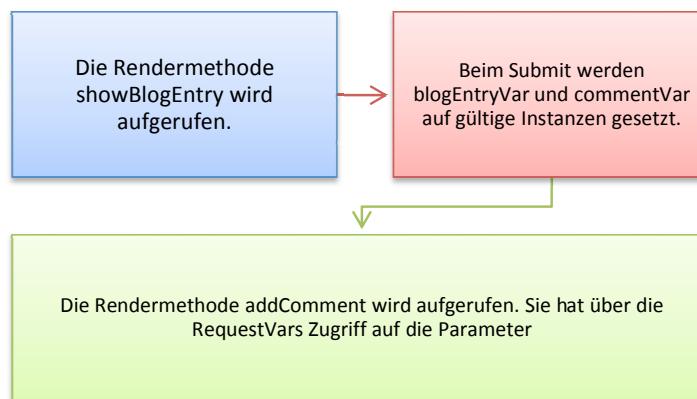


Abbildung 8: Ablauf der Parameterübergabe

## 6.3 Fortgeschrittenes Binding

### 6.3.1 Problemstellung

Für gewisse Problemstellungen ist die gewöhnliche Verwendung des bind() Befehl nicht ausreichend um das gewünschte Resultat zu erzielen. Beispielweise wenn ein Attribut einer Entität eine Liste ist welche man ausgeben möchte (Ein Blog Eintrag hat eine Menge von Kommentaren). Ein weiterer Anwendungsfall ist, wenn man nicht einfach ein Lift Tag mit einem gewissen Wert ersetzt haben möchte, sondern eine Abhängigkeit des Wertes zum umgebenden Markup Code besteht. Ein solcher Fall wird im folgenden etwas genauer betrachtet.

### 6.3.2 Verschachteltes Binding

Im folgenden Beispiel wird ein Lift Tag im Markup entweder durch einen Lösch-Link ersetzt, sofern der aktuelle Benutzer über ausreichend Rechte verfügt, oder aber es wird nichts angezeigt. Die Herausforderung dabei ist, dass die umschliessenden <tr> und <td> Tags nicht angezeigt werden sollen wenn der User nicht authentifiziert ist.

Das folgende Codelisting 62 zeigt den entsprechende Markup Code im Template.

---

```
<table>
  <lift:BlogSnippet:showComments>
    <comment:options>
      <option:remove>
        <tr class="editieren">
          <td>[
            <link:remove/>
          ]
        </td>
      </option:remove>
    </comment:options>
  </lift:BlogSnippet:showComments>
</table>
```

Codelisting 62: Verschachteltes Binding, Markup Code

Der eigentliche Link für den Löschvorgang wird durch das <link:remove/> Element gebunden.

Anschliessend wird durch das <option:remove/> ein weiterer Behälter definiert, welcher das <link:remove/> Tag einschliesst. Dieser Behälter wird wiederum durch das <comment:options/> Tag eingeschlossen. Durch diesen Übergeordneten Behälter hat man die Möglichkeit weitere Tags auf der Ebene des <option:remove/> Tags zu definieren (z.B. <option:edit/>). Mittels den erweiterten Möglichkeiten des Lift Binding kann so eine Struktur, wie im folgenden Abschnitt gezeigt wird, sinnvoll verwendet werden.

Der Code des zugehörigen Snippets BlogSnippet ist im Codelisting 63: Fortgeschrittenes Binding Code dokumentiert.

---

```

def showComments(html: NodeSeq): NodeSeq = {
    comments.flatMap(comment =>
        bind("comment", html,
            "options" -> {
                if (hasPermission)
                    bind("link", chooseTemplate("option", "remove", html),
                        "remove" -> SHtml.link("remove", () => {
                            }), Text("Remove"))
                else
                    NodeSeq.Empty
            }))
}

```

---

Codelisting 63: Fortgeschrittenes Binding Code

Im Zentrum dieses Beispiels steht die chooseTemplate Methode. Diese Methode extrahiert aus einer XML Struktur in Form einer NodeSeq den Inhalt eines Tags. Das gewünschte Tag kann mittels Prefix und Tag identifiziert werden. Das folgende Codelisting 64 dient zur Veranschaulichung der Funktionalität.

---

```

val textXml = <root>
    <bla1>...</bla1>
    <my:content>
        <div>Dieses div wird extrahiert</div>
    </my:content>
    <bla2>...</bla2>
</root>

println(chooseTemplate("my", "content", textXml))
// Ausgabe auf Konsole:
// <div>Dieses div wird extrahiert</div>

```

---

Codelisting 64: Erklärung der chooseTemplate Funktion

Im Weiteren wird auf die Details des Codelisting 63 eingegangen.

Das Tag options wird mit einer inneren Funktion gebunden, diese muss ein NodeSeq Objekt zurückgeben. Die Funktion prüft ob die Berechtigung vorhanden ist. Wenn nicht gib Sie einfach ein leeres NodeSeq Objekt zurück und das gesamte <comment:options> </comment:options> Tag wird durch nichts ersetzt, also entfernt. Damit ist oben erwähnte Anforderung erreicht, d.h. die umschliessenden <tr> und <td> Tags werden nicht angezeigt, wenn der User nicht authentifiziert ist

Falls der User authentifiziert ist, wird ein weiteres Binding durchgeführt. Dazu wird die beschriebene Lift Funktion chooseTemplate verwendet. Im Codebeispiel wird das Element <option:list></option:list> aus dem übergebenen XML Baum extrahiert und das Binding nur in diesem Element durchgeführt. Diesen zweiten bind Aufruf könnte man beispielsweise auch in einer Schlaufe plazieren um eine Liste von Elementen auszugeben.

**Hinweis**

Die Lift Funktion chooseTemplate() der Klasse BindHelpers wird gewöhnlich für andere (Templating) Zwecke eingesetzt. Das Beispiel in diesem Abschnitt zeigte wie Sie für die Lösung einer bestimmten Anwendung eingesetzt wurde.

### 6.3.3 Fazit

Das binding in Lift ist ein mächtiges Werkzeug und man tut gut daran sich etwas detaillierter darin einzuarbeiten. Meistens ist die Lösung für ein bestimmtes Problem ziemlich einfach realisierbar, wenn man erstmal mit den Details vertraut ist. Das Binding ist ein wichter Teil des Lift View First Prinzip. Durch konsequentes Anwenden erreicht man, dass Markup Code im eigentlichen Programmcode so weit wie möglich verschwindet. Die Auswahl an weiterführende Dokumentation zu diesem Thema ist leider noch ziemlich beschränkt. Ein Artikel (34) auf dem Lift Wiki erklärt weitere Hintergründe.

## 6.4 URL Rewriting

---

Unter URL Rewriting verstehen wir hier das Auswerten einer URL und das entsprechende Zuordnen einer Funktionalität auf diese URL. In der JSP Umgebung beschreibt der Begriff URL Rewriting auch das Verfahren die Session ID in jeder URL als Parameter anzuhängen um nicht von Cookies abhängig zu sein. Dabei werden aus der URL gewisse Parameter extrahiert. Diese Parameter sind Teil der URL und werden nicht etwa als http Parameter übergeben. So können lesbare URLs auf entsprechenden Inhalt gemappt werden. Dies erleichtert Funktionen wie Lesezeichen und auch die Indexierung durch eine Suchmaschine wird dadurch gefördert.

In Lift gibt es verschiedene Möglichkeiten URL Rewriting durchzuführen. Diese Möglichkeiten werden hier im Detail erläutert.

### 6.4.1 Rewriting mit Pattern Matching

Scala verfügt über eine sehr mächtige Pattern Matching. In Lift wird das Pattern Matching beim Parsen von URLs von grosser Bedeutung. Typisches URL Rewriting besteht aus Parsen einer URL und entsprechender Funktionszuordnung. So ist dies auch in Lift implementiert.

Einfaches URL Rewriting geschieht über das LiftRules Object. Das LiftRules Object ist für fast alle Applikationsspezifische Konfiguration verantwortlich. Lift unterscheidet zwischen StatefulRewrite und StatelessRewrite, die Unterschiede werden in Tabelle 16 aufgeführt.

Rewrite Typ	Beschreibung
<b>StatelessRewrite</b>	Dieser Rewrite wird sehr früh im http Request Cycle ausgeführt. Zu diesem Zeitpunkt sind noch keine Zustandsinformationen verfügbar. Dieser Rewrite sollte für Stateless Dispatch verwendet werden.
<b>StatfulRewrite</b>	Dieser Rewrite wird im Scope des S Zustandes ausgeführt. Der Zustand des S Objects sowie SessionVars und andere Sessionabhängige Daten sind vorhanden.

Tabelle 16: Vergleich zwischen Statefull- und less Rewrite

Konfigurationen über das LiftRules Objekt werden in der Boot Klasse vorgenommen. Im Codelisting 65 wird ein entsprechendes Beispiel gezeigt.

```
LiftRules.statefulRewrite.prepend(NamedPF("EntityRewrite") {
  case RewriteRequest(
    ParsePath("entity" :: "view" :: AsInt(id) :: Nil, _, _, _), _, _) =>
  {
    entityVar.set(findEntityById(id)) // Entity mit Id finden
    RewriteResponse("entity" :: "view" :: Nil)
  }
})
```

Codelisting 65: URL Rewriting in Boot Klasse

In diesem Beispiel wird ein Request auf die URL /entity/view/<id> auf die URL /entity/view weitergeleitet. Zusätzlich wird die <id> ausgewertet. In diesem Beispiel wird ein Extractor AsInt verwendet der an der entsprechenden Position ein Int Wert aus der URL extrahiert. Dieser ist anschliessend lokal unter dem Bezeichner id verfügbar. Anschliessend erfolgt die Auswertung dieses Argumentes. Über eine Funktion findEntityById wird eine Entity mit der übergebenen id gefunden und schliesslich der entityVar übergeben. Dies ist ein typisches Beispiel für das URL Rewriting. Aus einer URL wird ein Teil geparsst und in einer anderen Form als Parameter der Applikation zur Verfügung gestellt. In diesem Beispiel ist entityVar eine RequestVar die anschliessend im Scope der Seite /entity/view vorhanden ist.

Das Beispiel verwendet LiftRules.statefulRewrite. Die Verwendung von LiftRules.statlessRewrite ist identisch. Das Rewriting wird über sogenannte Named Partial Functions realisiert. Ein guter Artikel über Partial Functions findet sich hier (35).

#### Hinweis



Das URL Rewriting ist in anderen Web Frameworks ein absolutes Best Practice und weit verbreitet. In Lift jedoch ist dies nicht unbedingt zutreffend. Über das URL Rewriting muss die Web Applikation jeweils selber auf einen gültigen Zustand gesetzt werden. Wenn man sich auf das sehr konfortable Lift Funktionsbinding stützt (was auch Sinn macht) gerät man je nach dem mit dem URL Rewriting in Konflikt (Verlust von Zustandsinformationen). Dieser Problematik sollte man beim Design der Aufrufstruktur von Inhalten bewusst sein.

Detailliertere Informationen zu dieser Art von URL Rewriting können im entsprechenden Artikel auf dem Lift Wiki (9) entnommen werden.

## 6.4.2 Benutzerdefiniertes Dispatching

Lift bietet auch die Möglichkeit ein Benutzerdefiniertes Dispatching zu realisieren. Dies wird verwendet um selber dynamisch eine http Response für einen bestimmten Request zu generieren. Ein möglicher Anwendungsfall für diese Methode ist das bereitstellen von Bildern aus einer Datenbank. Diese Bilder existieren nicht als Files auf dem Webserver, sollen aber trotzdem über eine URL aufgerufen werden können.

Für diesen Zweck können über das LiftRules Object sogenannte Dispatch Partial Functions definiert werden. Diese enthalten erweiterende Dispatch Logik. Das Beispiel aus Codelisting 66 realisiert den vorher beschriebenen Anwendungsfall. Ein Aufruf auf den Pfad /image/<platzhalter> soll das entsprechende Bild aus der Datenbank holen und dem Client zurückschicken.

---

```

object ImageLogic {
  object TestImage {
    def unapply(in: String): Option[Picture] =
      // Finde Bild mit id und gebe es zurück
      Model.find(classOf[Picture], in.toLong)
  }

  def matcher: LiftRules.DispatchPF = {
    case req@Req("image" :: TestImage(img) :: Nil, _, GetRequest) =>
      () => serve(img, req, false)
  }

  def serve(img: Picture, req: Req, t: Boolean): Box[LiftResponse] = {
    // Generiere http Response mit Bild
    Full(InMemoryResponse(
      img.image,
      List("Last-Modified" -> toInternetDate(Helpers.millis),
           "Content-Type" -> img.imageType,
           "Content-Length" -> img.image.length.toString),
      Nil, 200))
  }
}

```

---

Codelisting 66: Benutzerdefinierte Dispatch Logik

Die Methode matcher ist für den Dispatch Vorgang zuständig. Ein Request auf z.B. /image/5 wird über den TestImage Extractor ausgewertet. Wenn dieser zur angegebenen Id 5 ein entsprechendes Image in der Datenbank findet ist es ein gültiger request und die Methode serve wird aufgerufen. Diese generiert eine http Response mit dem Bild als Inhalt.

Über die Klasse LiftResponse bietet Lift die Möglichkeit selbst http Responses zu generieren. Die Klasse InMemoryResponse ist eine Subklasse von LiftResponse. Für eine komplette Übersicht dieser Funktionalität kann die Lift Source Code Dokumentation der Klasse LiftResponse (25) zu Rate genommen werden.

Natürlich muss die Dispatch Funktion (im Beispiel ImageLogic.matcher) auf dem LiftRules Object konfiguriert werden. Dies geschieht wie gewohnt in der Boot Klasse.

---

```
LiftRules.dispatch.append(ImageLogic.matcher)
```

---

Codelisting 67: Benutzerdefinierter Dispatcher konfigurieren

## 6.4.3 Fazit

Lift bietet saubere Lösungen für das Realisieren von URL Rewritings. Ebenfalls wird ein komfortables Interface zur Verfügung gestellt, welches es erlaubt benutzerdefinierte dispatch Logik in den Request/Response Lifecycle des Frameworks einzuflechten. Die Möglichkeiten über die abstrahierten http Response Klasse die Antworten zu kontrollieren sind sehr mächtig und komfortabel zu verwenden.

## 6.5 Internationalisierung

---

Die Internationalisierung von Webapplikationen umfasst nebst verschiedenen Sprachen auch Datum, Währungen und andere kulturell verschiedene Aspekte einer Region (bzw. Locale). Dieser Abschnitt betrachtet zwei Möglichkeiten von Lift bezüglich der Unterstützung von verschiedenen Sprachen.

### 6.5.1 Property bundles

In den resources Ordner werden Dateien angelegt, welche key – value Paare enthalten. Diese Methode wurde in der Beispielapplikation angewendet.

---

```
tour.editTour>Edit Tour
tour.add/Create new tour
tour.yours=Your tours
tour.yours.desc=The following list contains all tours created by us
```

Codelisting 68: Internationalisierung, Tour\_en.properties

Codelisting 68 zeigt einen Ausschnitt der Datei Tour\_en.properties. Für die Beispielapplikation existiert immer auch eine Datei in Deutsch. In diesem Falle hätte das Bundle in Deutsch einen Dateinamen der Form Tour\_de.properties.

Die abgelegten Bundles müssen nun der Liftapplikation bekannt gemacht werden. Dazu ist das Codelisting 69 in der Boot.scala Klasse nötig. Da es sich dabei um eine Liste handelt, können neue Bundles leicht hinzugefügt werden.

---

```
LiftRules.resourceNames = "Tour" :: Nil
```

Codelisting 69: Internationalisierung, Boot.scala

### Zugriff über S Objekt

Für den Zugriff aus dem Scala Code kann das S Objekt mittels ? Methode verwendet werden.

---

```
S.?("tour.editTour")
```

Codelisting 70: Internationalisierung, S Objekt

Lift verfügt bereits über ein vordefiniertes Property Bundle namens lift-core\_\*\_.properties. Der Zugriff darauf verhält sich nahezu identisch wie in Codelisting 70 gezeigt. Anstatt eines Fragezeichens werden allerdings zwei (??) verwendet.

### Zugriff aus XHTML

Da in den Templates von Lift kein Markup Code erlaubt ist, kann das S Object nicht verwendet werden. Lift hält dafür ein Tag namens lift:loc bereit um in XHTML Dateien dennoch Gebrauch von den Property Bundles machen zu können.

---

```
<lift:loc locid="tour.editTour">Fallback value</lift:loc>
```

---

Codelisting 71: Internationalisierung, XHTML

Mit Codelisting 71 wird dasselbe wie mit Codelisting 70 erreicht. Codelisting 71 enthält zudem einen Standardwert, welcher ausgegeben wird, falls der angegebene Key in den Bundles nicht gefunden werden konnte.

## 6.5.2 Benennung der Templates

Als zweite Möglichkeit der Internationalisierung bietet sich die Benennung der Templates an:

index\_en.html  
index\_en\_GB.html  
index\_de.html

Lift entscheidet nun aufgrund der gesetzten Locale, welche der Templates gerendert wird.

## 6.5.3 Lokalisierung

Welches Bundle schlussendlich ausgewählt wird, wird aufgrund der zurückgegebenen Locale entschieden. Die Locale wird standardmäßig von Lift detektiert. Dazu durchsucht Lift den mitgesendeten Request Header des Clienten und versucht daraus die Sprache festzustellen. Gelingt dies nicht, wird die Sprache des Server Containers verwendet.

Will die Sprache manuell gesetzt werden, muss die LiftRules.localCalculator Methode durch eine Eigene ersetzt werden.

```

def localeCalculator(request: Box[HTTPRequest]): Locale = {
    object sessionLanguage extends SessionVar[Locale](LiftRules.defaultLocaleCalculator(request))
    request.flatMap(r => {
        def localeCookie(in: String): HTTPCookie =
            HTTPCookie("language", Full(in),
                       Empty, Full("/"), Full(2629743), Empty, Empty)
        def localeFromString(in: String): Locale = {
            val x = in.split("_").toList;
            sessionLanguage(new Locale(x.head, x.last))
            sessionLanguage.is
        }
        def calcLocale: Box[Locale] =
            S.findCookie("language").map(
                _.value.map(localeFromString)
            ).openOr(Full(sessionLanguage.is))
        S.param("locale") match {
            case Full(null) => calcLocale
            case f@Full(selectedLocale) =>
                S.addCookie(localeCookie(selectedLocale))
                Helpers.tryo(localeFromString(selectedLocale))
            case _ => calcLocale
        }
    }).openOr(sessionLanguage.is)
}

LiftRules.localeCalculator = localeCalculator -

```

#### Codelisting 72: Internationalisierung, localeCalculator

Codelisting 72 zeigt eine mögliche localeCalculator Methode in der Boot.scala Klasse. Die Methode wird dabei bei jeder Lokalisierungsauflösung durch S.?() oder <lift:loc> angesprungen. Der Ablauf ist wie folgt:

Zunächst wird nach einem Parameter namens „locale“ gesucht. Falls dieser existiert wird ein Cookie namens „language“ angelegt und dem Benutzer übergeben. Gleichzeitig wird die SessionVariable auf die neue Locale gesetzt und zurückgegeben.

Falls der „locale“ Parameter nicht existiert, wird nach einem Cookie namens „language“ gesucht. Falls dieses ebenfalls nicht existiert wird die Locale der SessionVariable zurückgegeben.

Um die Sprache dauerhaft umzustellen, genügt nun ein Link in Form von:

<http://www.meinseite.ch/?locale=en>

#### Hinweis



Bei der SessionVariable „sessionLanguage“ handelt es sich um einen Workaround. Komponenten, welche sich im Session Scope befinden (z.B. Comet Actors) und dennoch auf die localeCalculator Methode zugreifen, bringen keinen gültigen HTTPRequest mit, was die Detektierung eines Cookies verunmöglicht. Deshalb wird parallel zum Cookie die SessionVariable mitgeführt, welche von Session Komponenten ausgelesen werden können.

## 6.6 Verwendung von Properties

Werte, welche häufig ändern, können mit Property Dateien bequem ausgelagert werden. Lift bietet dazu im lift.util Package die Klasse Props an mit welcher die Werte aus den Properties-Dateien gelesen werden können. Damit die Klasse die Properties-Datei auch findet, muss sich diese im Ordner „resources“ der Webapplikation befinden.

---

```
version=0.9
lastUpdate=27.05.2010
authors=Ralf Muri, Daniel Hobi
```

---

Codelisting 73: Properties, default.props

Codelisting 73 zeigt den Inhalt der default.props Datei, wie sie für die Beispielapplikation verwendet wird. Der Inhalt ist nach dem Prinzip key=value gegliedert.

Damit die Properties auch in Templates verwendet werden können, empfiehlt es sich, eine Snippetklasse zu erstellen.

---

```
class Props {
    def render(html: NodeSeq): NodeSeq = {
        S.attr._("key").map(_.text) match {
            case Some(key) => Text(Props.get(key, "not found"))
            case _ => NodeSeq.Empty
        }
    }
}
```

---

Codelisting 74: Properties, Props.scala

Die Funktionsweise der Klasse Props (Codelisting 74) ist simpel. Die render – Methode erwartet ein Attribut namens „key“, welches einen Wert beinhaltet. Falls kein key Attribut gefunden wird, wird ein leeres NodeSeq zurückgegeben. Falls das key Attribut gefunden wurde, aber in der default.props Datei nicht definiert ist, wird ein „not found“ zurückgegeben.

Codelisting 75 zeigt die Verwendung der Snippetklasse Props in einem Template. In diesem Beispiel wird das Tag durch den Wert „Ralf Muri, Daniel Hobi“ ersetzt.

---

```
<lift:Props key="authors" />
```

---

Codelisting 75: Properties, Verwendung in Template

## 6.7 Interoperabilität mit Java

### 6.7.1 Problemstellung

Scala Entwickler kommen häufig aus dem Java Umfeld auf Grund der nahen Verwandschaft. Bei der Entwicklung einer Webapplikation mit Lift gibt es viele Aspekte welche mit der (richtigen) Verwendung des Frameworks zusammenhängen. Ein anderer Teil der Funktionalität beschränkt sich auf die Problem Domain der Applikation oder das Lösen von allgemeinen Problemen. Für solche Zwecke gibt es von der Java Seite her viel Unterstützung. Dieses Kapitel betrachtet die Interoperabilität mit Java im Kontext von Lift.

### 6.7.2 Interoperabilität mit Java

Die Bytecode-Kompatibilität von Scala zu Java ermöglicht hier eine optimale Zusammenarbeit um die Produktivität zu maximieren. Scala Klassen werden direkt in Java-Bytecode kompiliert und sind somit firstclass-children der JVM (Ein anderer Ansatz wäre das kompilieren in Java).

Diese Kompatibilität erlaubt es Scala Klassen von Java Klassen abzuleiten und umgekehrt. Obwohl die Scala Traits als Sprachelemente mächtiger sind als Java Interfaces (Traits erlauben Definitionen von Methoden, also nicht komplett abstrakt) werden diese auf Bytecode Level in Interfaces umgewandelt. Für eine detailliertere Einführung mit direktem Vergleich mit Scala/Java Code kann der informative Artikel (36) von Daniel Spiewak zur Hand genommen werden.

Mit dieser vollständigen Interoperabilität könnte man Scala, wie es von Entwicklern zum Teil gemacht wird, einfach als eine Java Library bezeichnen.

### 6.7.3 Implizite Konvertierung für Java Collections

Ein sehr wichtiges und häufig verwendetes Feature ist die Verwendung von Java Collections in Scala. Die Konvertierung ist praktisch und notwendig, da Java basierte Libraries mit Java Collections arbeiten und Scala basierte (z.B. Lift) mit Scala Collections. Weiter sind die Scala Collections einiges mächtiger als diejenigen von Java. Es gibt zwei verschiedene Möglichkeiten für die Konversion von Java Collections zu Scala Collections.

Die erste Möglichkeit ist in Scala enthalten. Seit Scala 2.8 gibt es Unterstützung für das automatische (implizite) Konvertieren von Java Collections in Java Collection und umgekehrt mittels der Klasse `scala.collection.JavaConversion`. Einen Überblick liefert die Source Code Dokumentation (37). Da es sich um implizite Konvertierungen handelt ist das Einbinden in den Scala Code denkbar einfach. Das Codelisting 76 zeigt wie die Konvertierungen eingebunden werden.

---

```
import scala.collection.JavaConversions._
```

---

Codelisting 76: Implizite Java Collection Conversions und Anwendung

Nach diesem Import Statement kann die automatische Konvertierung verwendet werden. Der Screenshot in Abbildung 9 zeigt, dass die funktionalen Methoden der Scala List nun auch auf der Java List vorhanden ist.

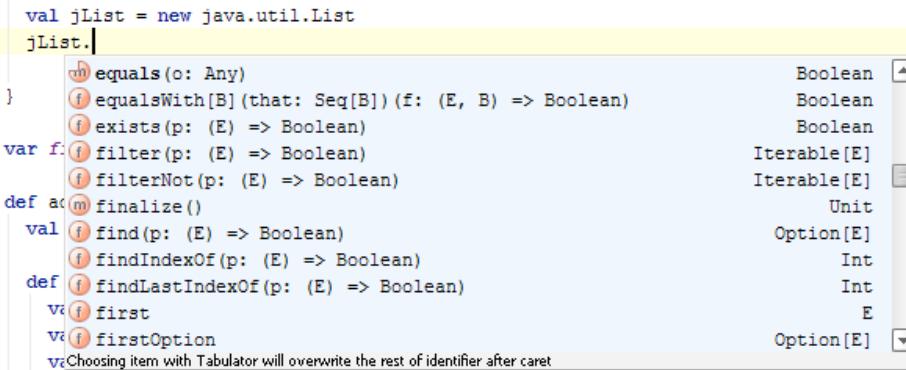


Abbildung 9: Funktionale Methoden durch implizite Konvertierung

Eine zweite Möglichkeit bietet die von Jorge Ortiz entwickelte Library „scalaj-collection“. Die Autoren dieser Library behaupten die implizite Konvertierung des ersten Ansatzes sei irreführend und fehleranfällig. Eine Gegenüberstellung der beiden Ansätze kann aus einem Artikel (38) von Daniel Spiewak entnommen werden. Die scalaj-collection Library stellt explizite Konvertierungsmethoden zur Verfügung.

Der Source Code dieser Library wird über ein GitHub Repository zur Verfügung gestellt. Die gepackte Bibliothek kann über das Scala-Tools Repository(39) bezogen werden. Wenn die Library in einem eigenen Projekt verwendet werden möchte, kann diese einfach über eine Maven Abhängigkeit hinzugefügt werden. Der entsprechende Code wird in Codelisting 77 gezeigt.

---

```

<dependency>
  <groupId>org.scalaj</groupId>
  <artifactId>scalaj-collection_${scala.version}</artifactId>
  <version>1.0.Beta2</version>
</dependency>
  
```

Codelisting 77: Maven Abhängigkeit für scalaj-collection Library

Die scalaj-collection Library erweitert die Java/Scala Collections über eine implizite Konvertierung mit der Methode asScala respektive asJava. Die Verwendung wird in folgendem Codelisting 78 demonstriert.

---

```

import scalaj.collection.Imports._

List(1, 2, 3).asJava
// returns java.util.List[java.lang.Integer]

Map(1 -> "a", 2 -> "b", 3 -> "c").asJava
// returns java.util.Map[java.lang.Integer, java.lang.String]

Set(1, 2, 3).asJava
// returns java.util.Set[java.lang.Integer]
  
```

Codelisting 78: Verwendung von scalaj-collections Conversions

## 6.7.4 Fazit

Die vollständige Kompatibilität zu Java ist ein mächtiges Feature von Scala. Sie ermöglicht die Verwendung von unzähligen Java Bibliotheken welche für alle möglichen Problemstellungen existieren. Sie ermöglicht auch die Zusammenarbeit von Java- und Scala-Entwicklern an einem Projekt. Ebenfalls erleichtert Sie den Umstieg zu Scala bzw. die Erweiterung der Java Welt um Scala.

In der Beispielapplikation wurden verschiedene male auf die implizite Konvertierung von Java Collections zu Scala Collections zurückgegriffen. Da man mit JPA nur Java Collections annotiert werden können, wurden in den Domain Entity Klassen Java Collections verwendet. Bei der späteren Verwendung der Entity Klassen wurden diese jeweils zu Scala Collections konvertiert um von dem erweiterten Funktionsumfang profitieren zu können. Ebenfalls wurden z.B. beim handling von Images (Grösse ändern) auf die Funktionalität der Java2D Libary zurückgegriffen.

## 7 Beispielapplikation

---

Parallel zum Technologiestudium wurde eine Beispielapplikation entwickelt, welche die gelernte Theorie in die Praxis umsetzt. Die TravelCompanion getaufte Applikation besitzt alle der abgehandelten Aspekte des 4. Kapitels und war Inspirationsquelle für Kapitel 6, in dem auf Best Practices und Lift Patterns eingegangen wird.

Es sei an dieser Stelle erwähnt, dass es sich bei TravelCompanion nicht um eine von Grund auf neue Applikation handelt. Sie wurde aus der Diplomarbeit „Grails – Studie und Webapplikation Reiseplaner“ (40) entnommen. Sowohl die grafische Oberfläche als auch das gesamte Domainmodell wurde 1:1 übernommen um daraus einen Technologievergleich ziehen zu können.

Die Beispielapplikation TravelCompanion dient als Testobjekt für das Studium von Scala mit dem Lift Web Framework. Im Rahmen dieser Arbeit wurde nicht geplant die Applikation produktionsreif zu entwickeln. Dies gilt sowohl für den eigentlichen Source Code, als auch für die Dokumentation.

Eine Applikation, welche für die Produktion entwickelt wird, hat andere Anforderungen an die Dokumentation, als eine Prototypenhafte Beispielapplikation wie TravelCompanion dies ist. Dieses Kapitel repräsentiert die gesamte Architektur- & Benutzer Dokumentation von TravelCompanion, unterteilt in verschiedene Abschnitte. Der Umfang der Dokumentation der Softwarearchitektur wurde in einem für das Projekt zweckmässigen Rahmen gehalten.

Im Abschnitt 7.1 wird auf die Architektur der Web Applikation eingegangen. Es wird eine grobe Übersicht der Applikation gegeben. Weiter werden einige technische Aspekte welche Einfluss auf die Applikation haben genauer beschrieben.

In den folgenden Abschnitten 7.2 bis 7.4 wird die Applikation aus drei verschiedenen Perspektiven beschrieben. Die Beschreibung der Applikation aus Sicht des Benutzers (Abschnitt 7.2) kann als eine User Interface und eine Art Benutzerdokumentation verstanden werden. Die Sicht des Software Entwicklers (Abschnitt 7.3) beschreibt die Projektstruktur, wie sie für den Entwickler relevant ist. Zum Schluss folgt die Sicht des Deployers (Abschnitt 7.4) welcher die Applikation im laufenden Zustand pflegen muss.

## 7.1 Architektur

### 7.1.1 Vorgabe

Die Architekturvorgaben für die Beispielapplikation TravelCompanion wurden im Verlauf der Bachelorarbeit „GRails und das Springframework“ (40) erarbeitet. Für die Web Applikation TravelCompanionScala wurden dieselben Rahmenbedingungen wie für die GRails Studie, sofern anwendbar, verwendet. Da es sich bei der Applikation lediglich um eine Beispielapplikation, quasi ein grosses Stück Beispielcode, handelt, sind die Architektonischen Richtilien nicht von so grosser Bedeutung. Entsprechend wurde bei der gesamten Projektdurchführung dieser Aspekt auch nicht besonders verfolgt, sondern der Fokus mehr auf den eigentlichen Kern des Projekts, das Technologiestudium, gesetzt.

TravelCompanionScala ist ein Nachbau der Web Applikation TravelCompanion aus der GRails Studie. Das Verhalten sowie die Bedienbarkeit der beiden Webapplikation unterscheiden sich nur minimal. Die Architektur der Web Applikation unterscheidet sich, bis auf technologisch bedingte Aspekte, nicht. Die beschriebenen Aspekte sind teilweise sehr technologiespezifisch, d.h. es werden Scala/Lift spezifische Begriffe oder Aussagen verwendet bzw. gemacht.

In den folgenden Unterabschnitten wird eine grobe Übersicht der Architektur von TravelCompanionScala gegeben. Weiter werden einige technologiespezifische Aspekte hervorgehoben. Für weitere Details zur Architektur kann die GRails Studie (40) als Referenz genommen werden.

### 7.1.2 Domainmodell

Die folgende Abbildung 10 zeigt das Domainmodell von TravelCompanion.

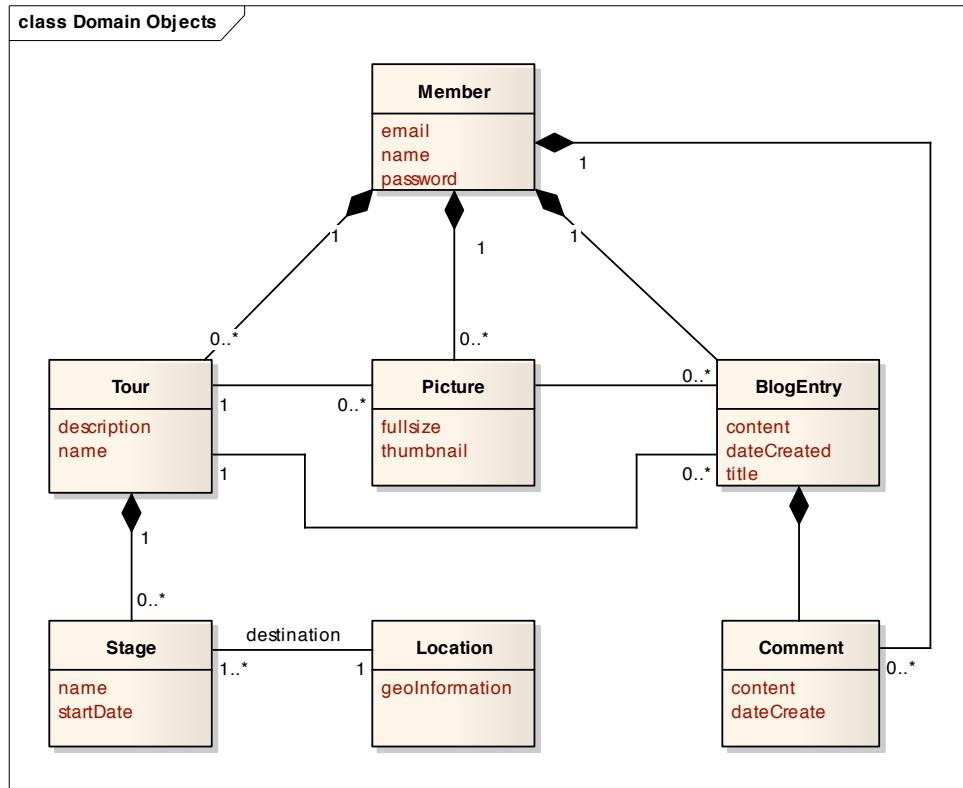
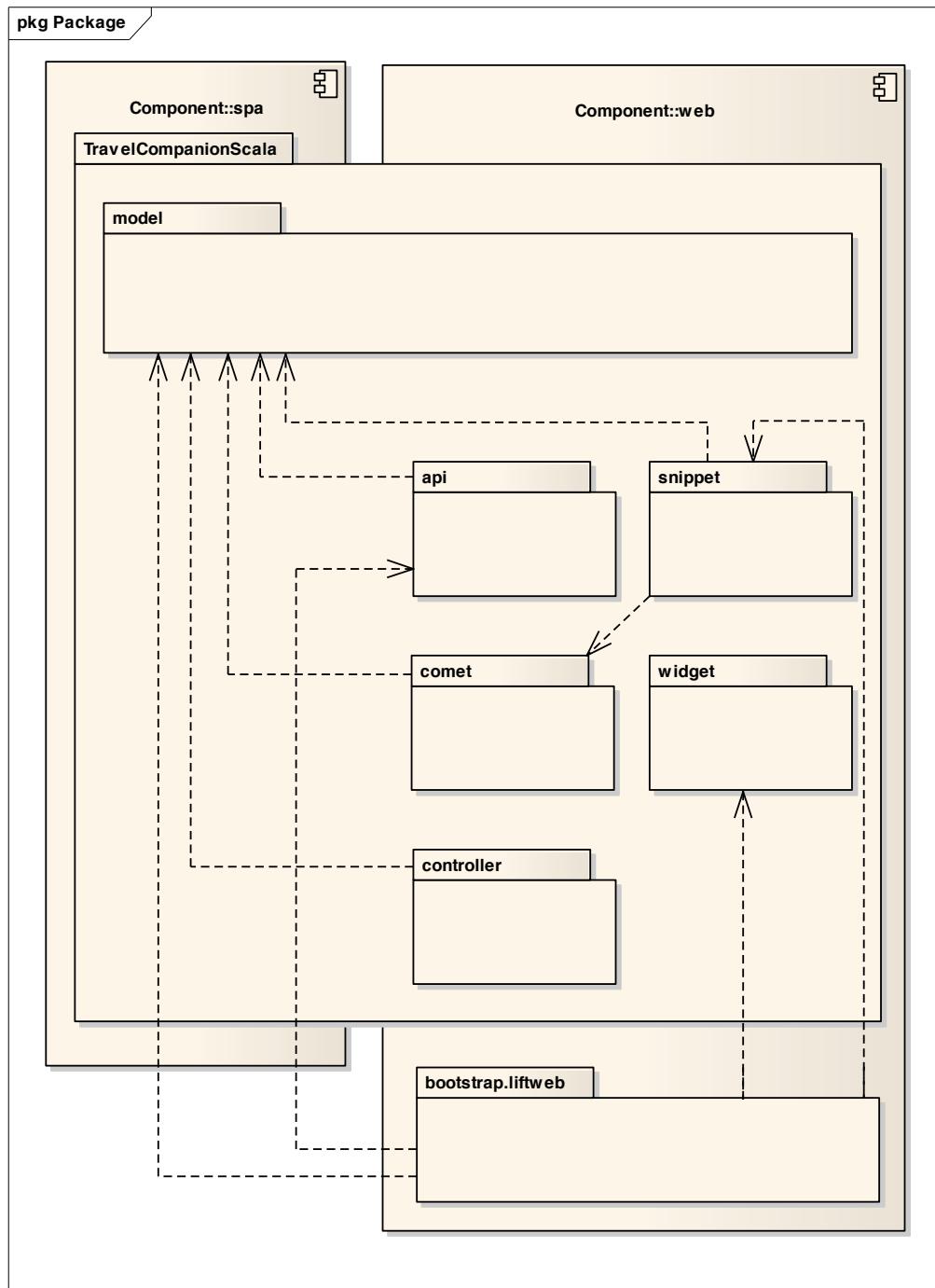


Abbildung 10: Domainmodell TravelCompanion

Die Kompositionen zeigen die Abhängigkeiten zwischen den verschiedenen Entitäten. Die Applikation ist so implementiert, dass die Kompositionen streng eingehalten werden, d.h. beim Erstellungs- und Löschvorgang werden diese entsprechend berücksichtigt. Die einzelnen Komponenten wurden im Diagramm mit den wichtigsten (nicht vollständig) Attributen versehen.

### 7.1.3 Modularisierung

Die Applikation ist in zwei verschiedene Module unterteilt. Der Persistenz Layer liegt im „spa“ (Scala Persistence API) Modul. Die Logik sowie das User Interface liegen im „web“ Modul. Das folgende Diagramm (Abbildung 11) gibt eine Übersicht über den Aufbau und die Struktur der Packages.

Abbildung 11: Komponenten und Package Struktur `TravelCompanionScala`

In Tabelle 17 sind die dargestellten Packages mit ihrer Funktion beschrieben.

Modul	Package	Beschreibung
spa, web	TravelCompanionScala	Dieses Package ist das Hauptpackage der Applikation. Sowohl Klassen aus dem web, wie auch dem spa Modul sind in diesem Package. Es beinhaltet fast alle relevanten Packages der Applikation.
web	bootstrap.liftweb	Dieses Package enthält die Boot Klasse welche für das Bootstrapping und weiterer Controller Funktionalität in Scala verwendet wird.
spa, web	model	Das Model Package enthält alle Domain Entity Klassen der Web Applikation sowie weitere Problem Domain spezifische Klassen.
web	api	Das Package API enthält die Funktionalität für den Zugriff über eine REST API auf die Webapplikation.
web	comet	Dieses Package enthält einen Teil der Comet Funktionalität für den dynamischen Blog.
web	controller	Dieses Package enthält Controllerklassen für den dynamischen Blog mit Comet.
web	snippet	Das Snippet Package enthält sämtliche Snippetklassen welche für die Bereitstellung von dynamischem Inhalt verwendet werden.
web	widget	Das Package Widget enthält Klassen, welche für das Gauge Widget von Bedeutung sind.

Tabelle 17: Package Beschreibungen

Die Struktur der eben beschriebenen Modularisierung orientiert sich an der Standardmodularisierung, welche durch Lift Beispielapplikationen vorgegeben ist.

### 7.1.4 Dynamische Sicht

In diesem Abschnitt werden einige dynamische Vorgänge der Web Applikation im Detail beschrieben.

#### Request Dispatching

Dieser Abschnitt beschreibt die für das Request Dispatching wichtigen Klassen der Web Applikation. Die Verarbeitung eines http Request in Lift ist ziemlich komplex und durchläuft verschiedene Stufen. Dieser Abschnitt beschreibt lediglich die Klassen, welche an verschiedenen Hook Points des Frameworks, zusätzliche Dispatch Logik definiert haben. Die Diagramme, welche zur Beschreibung als Hilfe genommen werden, sind abstrahiert und sollen lediglich die prinzipielle Funktionsweise darstellen. Sie repräsentieren, nicht wie das interne Handling des Dispatching in Lift tatsächlich repräsentiert ist. Der Abschnitt dient der Übersicht, wo überall im Code

Dispatch-Logik vorhanden ist und welche Zugriffsmöglichkeiten auf die Web Applikation überhaupt bestehen.

Der wichtigste Zugriffspunkt auf den Inhalt der Webapplikation geschieht über das Sitemap. Nur auf eingetragene Seiten im Lift Sitemap kann zugegriffen werden. Das Dispatching wird in Abbildung 12 visualisiert.

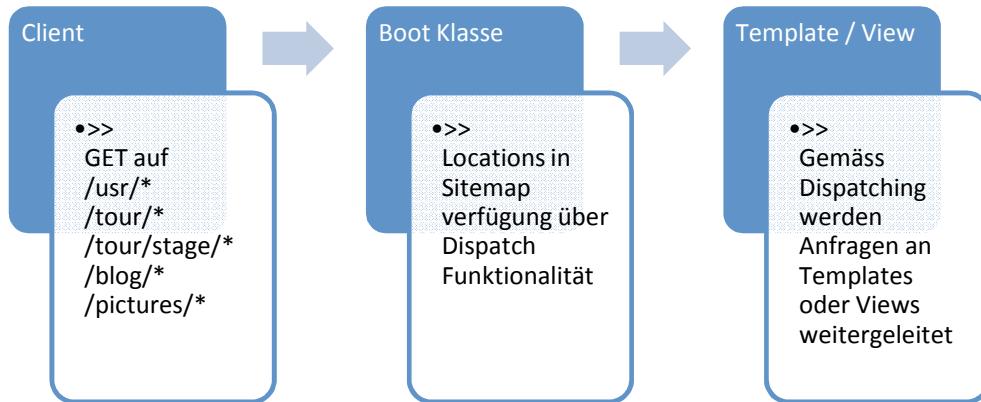


Abbildung 12: Dispatching des Sitemap visualisiert

Tabelle 18 beschreibt die beteiligten Komponenten im Detail.

Klasse	Beschreibung
<b>Boot</b>	Verschiedene Locations werden dem Sitemap über das LiftRules Objekt hinzugefügt. Das Dispatching zur entsprechenden Funktionalität ist im Sitemap festgehalten.
<b>Template</b>	Als Templates werden Markup Files bezeichnet. Requests auf Templates werden zum entsprechenden File (gemäß URI) weitergeleitet.
<b>View</b>	Scala Funktionen, welche Markup Code zurückliefern, werden als Views bezeichnet. Für diese Requests werden die Anfragen zur entsprechenden Scala Funktion weitergeleitet.

Tabelle 18: Beteiligte Klassen des Dispatching über das Sitemap

Die Web Applikation verfügt über eine Rest API. Sie stellt ein Interface zur Verfügung, über welches gewisse Funktionalitäten über die 4 http Grundoperationen (GET, POST, PUT, DELETE) einem entsprechenden Client zur Verfügung gestellt werden. Das erste Diagramm (Abbildung 13) visualisiert die beteiligten Komponenten beim Zugriff auf die Rest API.

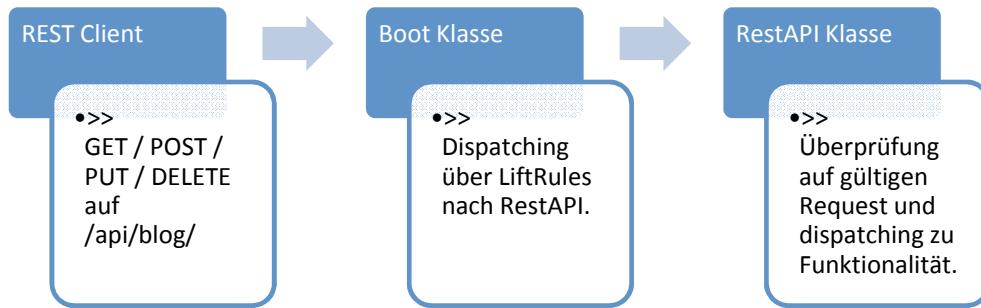


Abbildung 13: REST API dispatching visualisiert

Tabelle 19 beschreibt die beteiligten Klassen im Detail.

Klasse	Beschreibung
<b>Boot</b>	In der Boot Klasse wird über das LiftRules Objekt die RestAPI als zusätzliche Dispatch Klasse angehängt. Somit wird diese Klasse beim Ablauf des Dispatching einbezogen.
<b>RestAPI</b>	Die Klasse hat dispatch Funktionalität für verschiedene Aufrufe (GET, POST, PUT, DELETE) auf URLs unter /api/blog/*.

Tabelle 19: Beteiligte Klassen des Rest API dispatching

Die Web Applikation bietet die Möglichkeit, Bilder für einen Blog Eintrag hochzuladen. Von diesen Bildern wird automatisch ein Thumbnail erstellt. Abbildung 14 visualisiert die beteiligten Komponenten beim Zugriff auf die Bilder von Blog Entries.

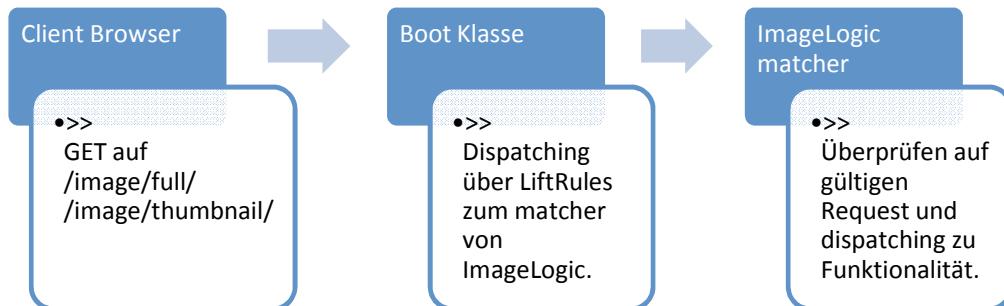


Abbildung 14: Image dispatching visualisiert

Tabelle 20 beschreibt die beteiligten Klassen im Detail.

Klasse	Beschreibung
<b>Boot</b>	In der Boot Klasse wird über das LiftRules Objekt die matcher Methode der ImageLogic Klasse angehängt. Somit wird diese Klasse beim Ablauf des Dispatching einbezogen.
<b>ImageLogic.matcher</b>	Diese Methode hat dispatch Funktionalität für Anfragen nach /image/full/* und /image/thumbnail/* sowie die Logik gemäss dem URI das entsprechende Bild zurückzuliefern.

Tabelle 20: Beteiligte Klassen des Image dispatching

Alle Requests, welche die Web Applikation erfolgreich behandeln kann, werden über die hier beschriebenen Dispatch Mechanismen realisiert. Wenn das Framework einen spezifischen Request nicht behandeln kann, also keine Dispatchmethode für ihn vorhanden ist, wird ein http 404 zurückgemeldet.

## CRUD Operationen auf Domain Entität

In diesem Abschnitt wird das Prinzip der CRUD Operationen auf Domain Entitäten beschrieben. Anhand eines Beispiels einer Edit Operation wird der Prozess allgemein erläutert.

CRUD Operationen werden durch Formulare im User Interface realisiert. Scala als funktionsorientierte Programmiersprache erlaubt es, Funktionsobjekte zu definieren. Über Lift ist es möglich, solche Funktionsobjekte z.B. an ein HTML Link oder an ein Form Submit Event anzuhängen. Von dieser Möglichkeit wird bei der Realisierung der CRUD Operationen stark gebraucht gemacht.

Das Sequenzdiagramm in Abbildung 15 visualisiert eine edit Operation auf eine Entität. Das Lift Framework wurde als einzelnes Objekt abstrahiert. Die Darstellung repräsentiert nicht die tatsächliche Klassenhierarchie. Sie dient der prinzipiellen Visualisierung des Vorganges. Methodenaufrufe sind z.T. ebenfalls abstrahiert dargestellt. Die tatsächliche Aufrufhierarchie unterscheidet sich von der Dargestellten.

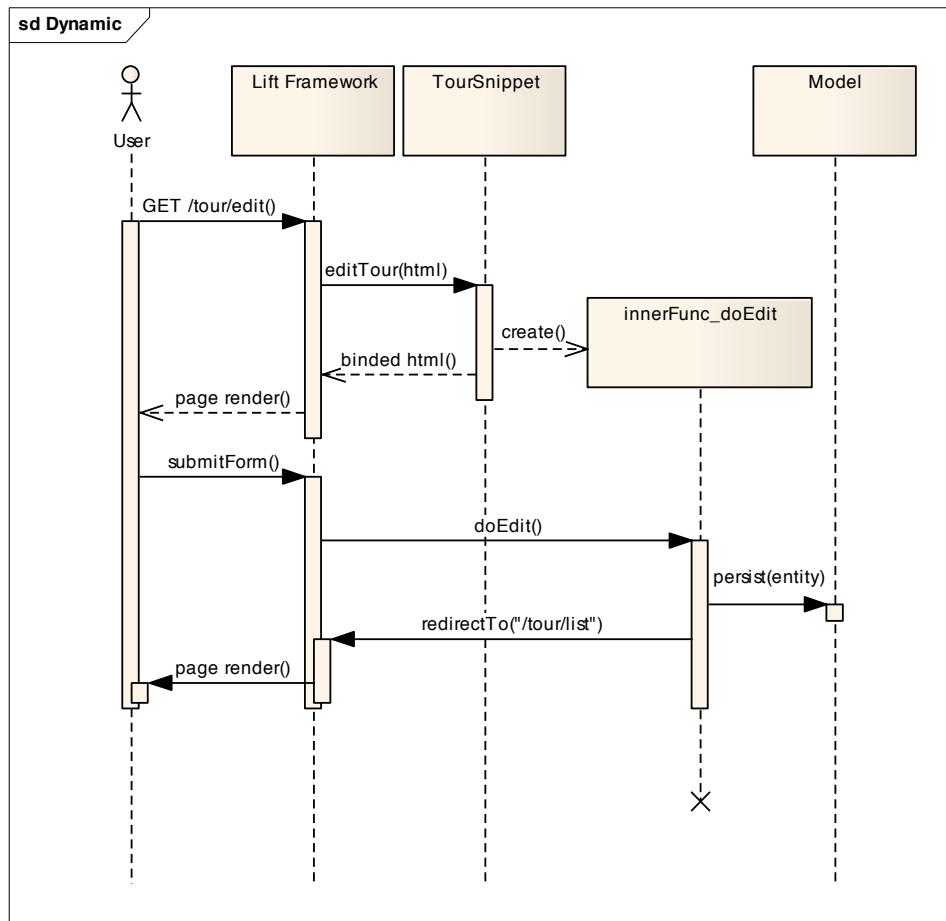


Abbildung 15: Dynamische Sicht auf CRUD Operation einer Domain Entität

Das innerFunc\_doEdit ist ein Funktionsobjekt, welches innerhalb der Methode editTour der Klasse TourSnippet erstellt wird. Dieses Funktionsobjekt beinhaltet die Logik um die Domain Entität zu persistieren und wird an den Submit Event des Edit Formulars gebunden.

Weitere CRUD Operationen auf andere Domain Entitäten der Web Applikation werden auf dieselbe Weise wie das beschriebene Beispiel realisiert.

## 7.1.5 Technische Aspekte

Die einzelnen technischen Aspekte der Web Applikation sind in diesem Dokument im entsprechenden Abschnitt detailliert dokumentiert. Es werden jeweils auch die architektonisch wichtigen Punkte des Aspektes hervorgehoben. Aus diesem Grund können auch die Abschnitte mit der Dokumentation der technischen Aspekte als weiterführende Referenz bezüglich der Architektur genommen werden.

## 7.2 Sicht des Benutzers

### 7.2.1 Startseite

Dieser Abschnitt widmet sich dem Benutzer, welcher die Applikation bedient. Mit diversen Abbildungen soll aufgezeigt werden, welche Anforderungen TravelCompanion an den Benutzer stellt und wie sich dieser darin zurechtfindet.

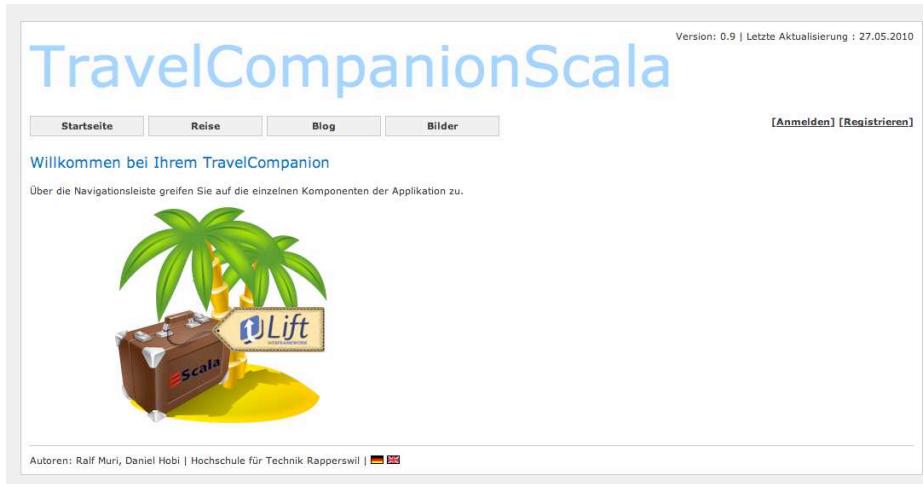


Abbildung 16: Beispielapplikation, Startseite

Abbildung 16 zeigt die Startseite der Beispielapplikation. Im oberen Teil befindet sich die Hauptnavigation, mit welcher durch die Seite navigiert werden kann. Rechts oben gelangt der Benutzer zu der Anmeldung bzw. zu dem Registrierungsformular. Es ist zu beachten, dass sich diese Punkte ändern, falls sich der Benutzer gegenüber der Applikation identifiziert hat (Anmelden). Die Applikation begrüßt den Benutzer in diesem Falle mit seinem registrierten Vor- und Nachnamen und gibt die Möglichkeit sich aus der aktiven Session auszuloggen.

Im mittleren Teil befindet sich der eigentliche Inhalt der Seite. Auf der Startseite ist dieser durch das Logo dieser Arbeit gekennzeichnet.

Im unteren Teil sind die Autoren aufgeführt. Des Weiteren sind die Landesflaggen von Deutschland und Grossbritannien zu sehen. Mit Hilfe dieser kann die Sprache der Seite in Deutsch bzw. Englisch umgestellt werden.

## 7.2.2 Reise

The screenshot shows a web application interface titled "TravelCompanionScala". At the top right, it says "Version: 0.9 | Letzte Aktualisierung : 27.05.2010". Below the title is a navigation bar with tabs: "Startseite", "Reise" (which is selected), "Blog", and "Bilder". On the far right of the navigation bar are links "[Anmelden]" and "[Registrieren]". The main content area has a heading "Reisen anderer Mitglieder" and a sub-instruction "Nachfolgend erhalten Sie eine Liste der von anderen Mitgliedern eingetragenen Reisen:". Below this is a table with the following data:

Name	Beschreibung	Ersteller
Glanus	Mis dihei	dhobi
<a href="#">My Travel</a>	description	Hobi
<a href="#">My Travel 2</a>	description 2	Hobi
<a href="#">Philippines</a>	Coming july is going to rock...	Ralf
<a href="#">Umlauttest</a>	äöü	dhobi
<a href="#">Zürich</a>	So als Demonstration... Eine Ortschaft mit einem Umlaut verursacht ein Problem... Aber nur auf der auf Tomcat deployten Version.	Ralf
<a href="#">myTrip</a>	myTrip	pbernet
<a href="#">öppn of Zuri (Umlauttest)</a>	keine	dhobi

At the bottom left of the content area, it says "Autoren: Ralf Muri, Daniel Hobi | Hochschule für Technik Rapperswil |

Abbildung 17: Beispielapplikation, Reisen

Abbildung 17 zeigt die Übersichtsseite des Navigationspunktes „Reise“. In einer Tabelle sind alle bisher erfassten Reisen dargestellt. Durch einen Klick auf den Tabellenkopf kann die Tabelle in alphabetischer Reihenfolge sortiert werden. Zuständig ist hierfür das im Abschnitt 5.5.1 beschriebene Tablesorter Widget.

## Detaillierte Reiseansicht

Mit einem Klick auf den Reisenamen gelangt man zur detaillierten Reiseansicht. Diese wiederum ist in die Bereiche „Reiseabschnitt“, „Blogeinträge“, „Distanzmesser“, „Bilder“ und „Map“ aufgeteilt.

The screenshot shows the detailed view for the trip "Philippines". At the top, it says "Philippines" and "Coming july is going to rock...". Below this is a table with the following data:

Start	Reiseabschnitt	Zielort	Beschreibung
30.06.2010	<a href="#">Tropic Thunder</a>	Cebu City	checking out the east...
31.07.2010	<a href="#">back home</a>	Zug	where it all began...

At the bottom left is a "Zurück" button.

Abbildung 18: Beispielapplikation, Reise, Reiseabschnitte

Im obersten Teil der Reiseansicht sind die Reiseabschnitte, wie in Abbildung 18 gezeigt, aufgelistet. Sie besitzen einen anklickbaren Namen, welcher zur Detailseite dieses Reiseabschnittes führt, und einen Zielort. Die Reiseabschnitte sind in zeitlicher Reihenfolge sortiert.

The screenshot shows the blog entries for the trip "Philippines". It lists two entries:

- 05.05.2010 00:00: [Taking off](#)  
Dies ist ein Typoblindtext. An ihm kann man sehen, [ [weiterlesen...](#) ]
- 12.05.2010 00:00: [Ein erster Test](#)  
mit Comet [ [weiterlesen...](#) ]

Abbildung 19: Beispielapplikation, Reise, Blogeinträge

Abbildung 19 zeigt die für diese Reise eingetragenen Blogeinträge. Ein Klick auf „weiterlesen...“ führt zur vollständigen Auflistung des Blogeintrags und deren Kommentare.



Abbildung 20: Beispielapplikation, Reise, Distanzmesser

Der Distanzmesser in Abbildung 20 ist in der Lage einen beliebigen Wert zwischen 0% und 100% anzuzeigen. Der Wert ist in der Beispielapplikation statisch auf 70% gesetzt. Die Entstehung des Distanzmessers ist im Abschnitt 5.5.2 erläutert.



Abbildung 21: Beispielapplikation, Reise, Bilder

Unterhalb des Distanzmessers befinden sich die Bilder (Abbildung 21), welche der Reise angehören.



Abbildung 22: Beispielapplikation, Reise, GoogleMap

Im untersten Teil der Reiseansicht befindet sich eine Google Map (Abbildung 22), welche die Reiseabschnitte mit Markern kennzeichnet. Die Marker sind in der Reihenfolge der Reiseabschnitte miteinander verbunden und geben bei einem Klick den Zielort preis.

## 7.2.3 Blog

Der Blog zeigt alle bisherigen Blogeinträge an. Interessant ist hier, dass diese Seite als Single Page Application implementiert ist. Dies bedeutet, dass alle Informationen abgerufen und Aktionen im eingeloggten Zustand getätigten werden können, ohne dass ein sichtbares Neuladen der Seite beobachtet werden kann.

Abbildung 23 zeigt den Navigationspunkt Blog. Sowohl die oberste als auch die unterste Tabelle werden per Comet überwacht. Wird von einer Drittperson ein Blogeintrag oder ein Kommentar geschrieben, wird dieser quasi live in die Tabelle eingefügt.

**Eingetragene Blogeinträge**

Nachfolgend erhalten Sie die Anzeige der Blogs von beliebigen Mitgliedern :

Titel	Vorschau	Letzte Aktualisierung	Ersteller
Taking off	Dies ist ein Typoblinktext. An ihm kann man sehen, [ <a href="#">weiterlesen...</a> ]	05.05.2010 00:00	Ralf
Los gehts	Weit hinten, hinter den Wortbergen, fern der Lände [ <a href="#">weiterlesen...</a> ]	05.05.2010 00:00	dhobi
Ein erster Test	mit Comet [ <a href="#">weiterlesen...</a> ]	12.05.2010 00:00	Ralf
ddsssd	ddsssdad [ <a href="#">weiterlesen...</a> ]	12.05.2010 00:00	dhobi

05.05.2010 00:00  
**Taking off gehört zur Tour: Philippines**  
Dies ist ein Typoblinktext. An ihm kann man sehen, ob alle Buchstaben da sind und wie sie aussehen. Manchmal benutzt man Worte wie Hamburgefonts, Raflenduks oder Handgloves, um Schriften zu testen. Manchmal SÄxtze, die alle Buchstaben des Alphabets enthalten - man nennt diese SÄxtze »Panograms«. Sehr bekannt ist dieser:  
The quick brown fox jumps over the lazy old dog.

**Kommentare**  
Ralf schrieb am 05.05.2010 00:00 :  
The quick brown fox jumps over the lazy old dog.  
dhobi schrieb am 05.05.2010 00:00 :  
The quick brown fox jumps over the lazy old dog.

Abbildung 23: Beispielapplikation, Blog

Mit einem Klick weiterlesen öffnet sich per Ajax Request der vollständige Blogeintrag mit den dazugehörigen Kommentaren. Die Realisierung dieser Seite in Verwendung von Ajax & Comet ist im Abschnitt 5.6 beschrieben.

## 7.2.4 Bilder

Der Navigationspunkt Bilder zeigt alle bisher hochgeladenen Bilder an, wie in Abbildung 24 gezeigt.

**Bilder anderer Mitglieder**

Nachfolgend erhalten Sie eine Anzeige der Bilder von anderen Mitgliedern :

Bild	Beschreibung	Besitzer	Gehört zu
	tale	Ralf	
	aber auch schön	dhobi	Reise Glarus
			Blogeintrag Los gehts

Abbildung 24: Beispielapplikation, Bilder

Ein Klick auf ein Bild führt dabei auf eine detaillierte Ansicht des Bildes.

## 7.3 Sicht des Software Entwicklers

Dieser Abschnitt widmet sich der Struktur der Beispielapplikation. Es wird eine Übersicht der Projektstruktur gegeben. Auf die einzelnen Schichten wird in den Folgeabschnitten detaillierter eingegangen.

### 7.3.1 Übersicht

Abbildung 25 zeigt die Projektstruktur der Beispielapplikation. Der Ordner .idea sowie die \*.iml Dateien werden automatisch von IntelliJ IDEA angelegt und haben keine Bedeutung für die Liftapplikation als solches.

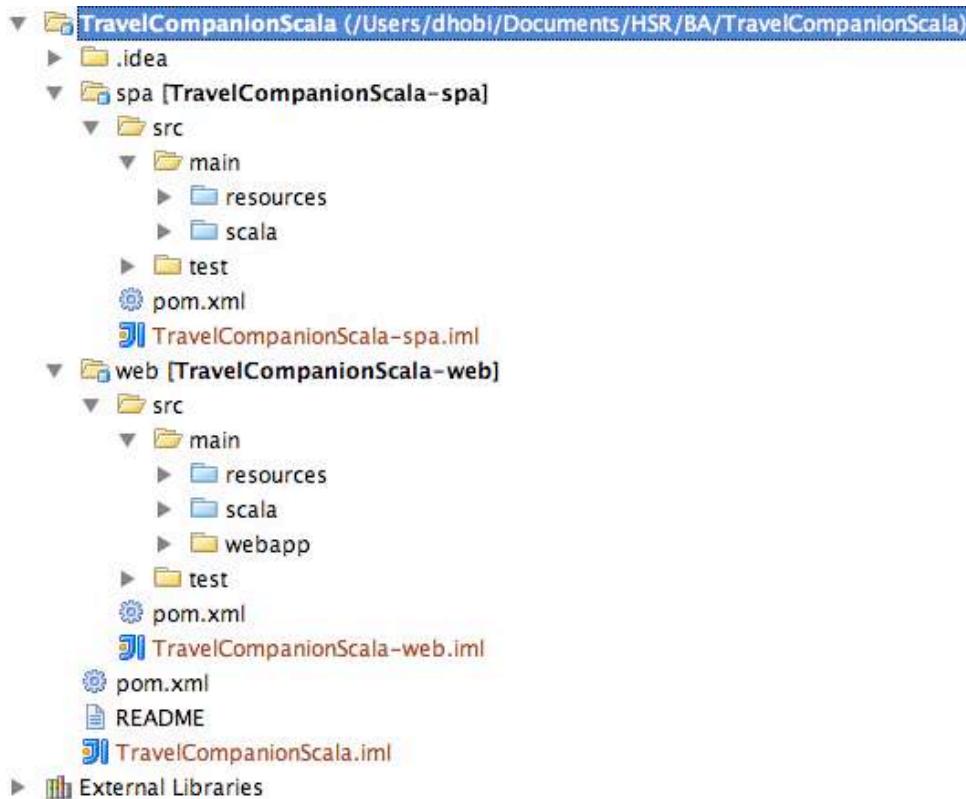


Abbildung 25: Beispielapplikation, Projektstruktur

Aus Abbildung 25 ist ersichtlich, dass zwei Module mit den Namen „spa“ und „web“ existieren. Module sind unabhängig voneinander und werden in getrennten Schritten kompiliert. Des Weiteren existieren drei Maven pom.xml Dateien. Je eines davon ist in dem Modul „spa“ bzw. „web“ abgelegt. Das dritte, hierarchisch gesehen höchste pom.xml ist für das Projekt (Maven bezeichnet dieses als Master) hinterlegt. Die pom.xml Dateien beinhalten Verweise auf benötigte Bibliotheken. Im Master pom.xml sind auch die Repositories angegeben, in welchen nach den Bibliotheken gesucht wird.

### 7.3.2 Das spa Modul

Das spa Modul beinhaltet die Domainentitäten, welche mit JPA Annotationen ausgestattet sind. Dieses Modul repräsentiert das Domainmodel der Applikation.

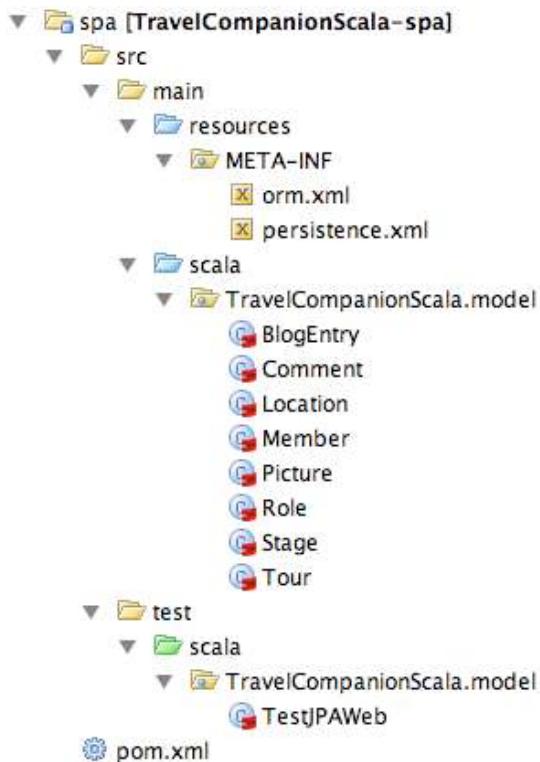


Abbildung 26: Beispielapplikation, spa – Modul

Das spa – Modul ist, wie in Abbildung 26 gezeigt, wiederum in drei Teile gegliedert. Im resources Ordner befinden sich die XML Dateien, welche für die JPA Implementierung verwendet werden (siehe auch Abschnitt 5.3.2). Im Scala Ordner befinden sich die eigentlichen Domainklassen. Der Ordner test beinhaltet eine Testklasse, welche den EntityManager mittels Persistierung und Auslesen von Domainentitäten prüft.

Die Datei pom.xml beinhaltet Verweise auf benötigte Bibliotheken wie EclipseLink, H2 Datenbank, Hibernate Validator, u.s.w.

### 7.3.3 Das web Modul

Das web Modul beinhaltet die eigentliche Liftapplikation. Abbildung 27 zeigt eine Übersicht. Im resources Ordner befinden sich die Language Bundles und der toserve Ordner, welche aus Platzgründen nicht gezeigt werden.

Im scala Ordner befindet sich die Scala Klassen, welche für die Logik der Applikation verantwortlich sind.

Der Ordner webapp beinhaltet XHTML Dateien für die Darstellung der Beispielapplikation und kann somit als view Softwareschicht angesehen werden.

Der test Ordner beinhaltet eine Testklasse.

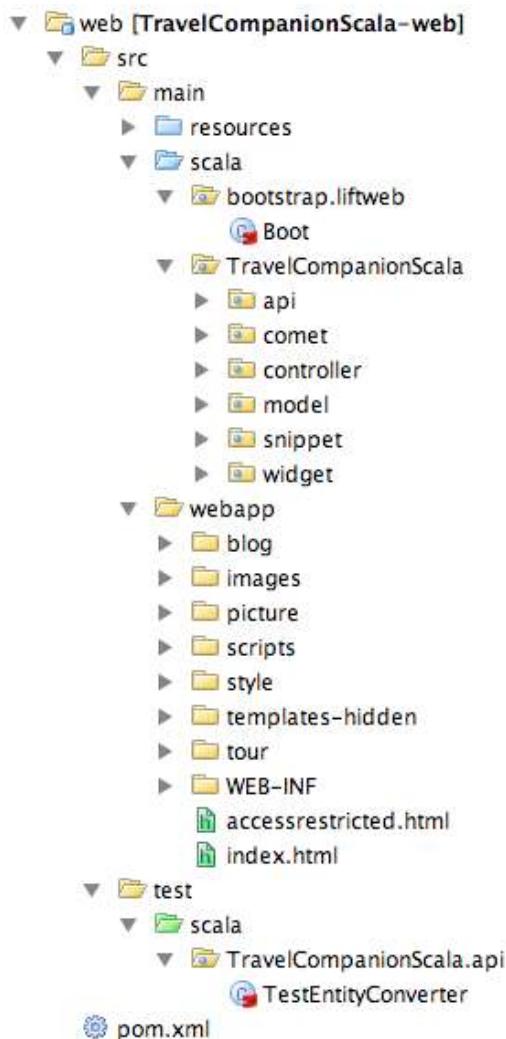


Abbildung 27: Beispielapplikation, web – Modul

Die pom.xml Datei verweist hier vor allem auf Lift Bibliotheken. Zudem ist ein Verweis auf das spa Modul vorhanden, damit dieses auch in der Liftapplikation genutzt werden kann.

## 7.4 Sicht des Deployers

Im ersten Abschnitt wird auf das Jetty Plugin eingegangen. Darauf wird ein Deployment auf einem Tomcat Container betrachtet. Abschliessend wird der externe Zugriff auf die H2 Datenbank durchleuchtet.

### 7.4.1 Jetty Plugin

Die Beispielapplikation ist als Maven Projekt konzipiert und verfügt daher über die bekannten Lifecycle Methoden. Das web Modul bietet zusätzlich das jetty Plugin an, mit welchem das Modul kompiliert und anschliessend in den Jetty Container, welcher ebenfalls automatisch startet, deployed wird. Das pom.xml im web Modul gibt vor, auf welchem Port jetty auf eingehende Requests horcht. Codelisting 79 zeigt einen Ausschnitt aus dieser pom.xml Datei. Der jetty Container horcht demnach auf Port 9090 auf eingehende Requests und prüft das web Modul alle fünf Sekunden auf Änderungen für ein Redeploy.

---

```

<plugin>
  <groupId>
    org.mortbay.jetty
  </groupId>
  <artifactId>
    maven-jetty-plugin
  </artifactId>
  <version>
    6.1.22
  </version>
  <configuration>
    <contextPath>
      /
    </contextPath>
    <scanIntervalSeconds>
      5
    </scanIntervalSeconds>
    <connectors>
      <connector
        implementation="org.mortbay.jetty.nio.SelectChannelConnector">
        <port>
          9090
        </port>
        <maxIdleTime>
          60000
        </maxIdleTime>
      </connector>
    </connectors>
  </configuration>
</plugin>
```

---

Codelisting 79: Beispielapplikation, jetty Plugin

Nebst den Lifecycle Methoden ist durch die Angabe des Codelisting 79 auch ein weiterer Ordner namens Plugins sichtbar. Abbildung 28 zeigt einen Ausschnitt aus dem Maven Projects Menu der IntelliJ IDEA.

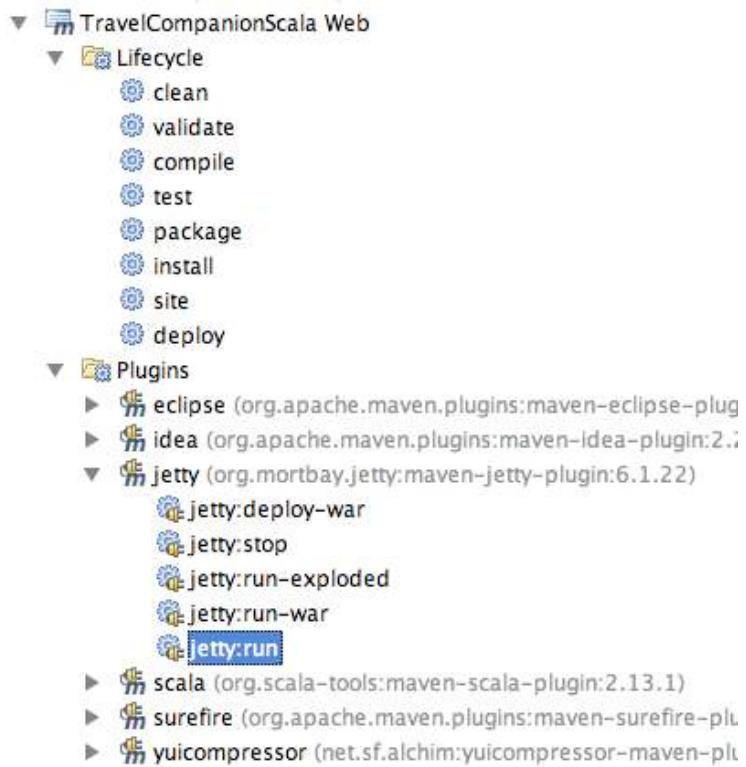


Abbildung 28: Beispielapplikation, jetty Plugin

Ein Doppelklick auf „jetty:run“ startet die Applikation.

#### Warnung



Falls Änderungen im spa Modul getätigt wurden, muss dieses manuell per install Lifecycle Methode neu erstellt werden. Ansonsten setzt das web Modul auf dem zuletzt erstellten spa Modul auf.

## 7.4.2 Deploying auf Tomcat

Das Deploying in einen Tomcat Container ist denkbar einfach. Alles was Tomcat dafür benötigt, ist eine war Datei. Durch ein Aufruf der install Lifecycle Methode im Master Projekt wird diese erstellt.

Für frisch aufgesetzte Tomcat Server gilt es zu beachten, dass diese über keinen Transaction Manager verfügen. Durch die Verwendung von JPA im spa Modul ist dies aber zwingend erforderlich. Der Transaction Manager muss daher manuell nachinstalliert werden. In unserem Falle verwendeten wir den freien Java Open Transaction Manager (JOTM).

## 7.4.3 Zugriff auf H2 Datenbank

Der Zugriff auf die H2 Datenbank geschieht standardmäßig exklusiv. Daraus resultiert, dass während laufendem Betrieb der Applikation keine weitere Möglichkeit besteht, auf die Datenbank zuzugreifen.

Um diesem Umstand abzuheften, genügt es, die persistence.xml, wie in Codelisting 80 gezeigt, anzupassen.

```
<property name="javax.persistence.jdbc.url"  
value="jdbc:h2:file:~/TravelCompanion;FILE_LOCK=NO"/>
```

Codelisting 80: Beispielapplikation, H2 Datenbankzugriff

---

**Warnung**



Durch die manuelle Aushebelung des exklusiven Zugriffs können die ACID Grundsätze nicht mehr erfüllt werden. Es empfiehlt sich den manuellen Zugriff nur für Leseoperationen zu verwenden (ohne Garantie auf Aktualität der Datensätze versteht sich).

## 8 Auswertung und Erfahrungsberichte

---

Dieses Kapitel befasst sich mit der Auswertung des Technologiestudiums und allgemein der Projektdurchführung. Weiter haben wir Erfahrungsberichte verfasst, welche unsere persönliche Erfahrungen und Meinungen bezüglich des Technologiestudiums und der Projektdurchführung reflektieren.

Der erste Abschnitt 8.1 behandelt die Auswertung in welchem verschiedene Aspekte des Technologiestudiums zusammengefasst werden. Der zweite Abschnitt 8.2 enthält die persönlichen Erfahrungsberichte der Projektmitarbeiter. Der Schlussteil dieses Kaptiels bildet der Abschnitt 8.3 mit einem Schlusswort.

## 8.1 Auswertung Technologiestudium

In diesem Abschnitt wird zunächst die Einarbeitungsphase in Scala (8.1.1) bzw. in Lift (8.1.2) näher erläutert. Die Lernkurve für Java Entwickler wird im Abschnitt 8.1.3 näher beschrieben. Über die Zukunft von Scala und Lift wird in Abschnitt 8.1.4 bzw. 8.1.5 spekuliert. Eine Empfehlung über die Verwendung von Scala / Lift wird im Abschnitt 8.1.6 abgegeben. Offene Punkte bzw. Ausbaumöglichkeiten werden in Abschnitt 8.1.7 aufgezeigt.

### 8.1.1 Einarbeitungsphase für Scala

Für die Einarbeitung in Scala stehen recht viele und verhältnismässig gute Hilfsmittel zur Verfügung. Die Scala Community ist seit Einführung der Programmiersprache ziemlich gewachsen was sich auf verschiedene Arten bemerkbar macht. Es wurden bereits einige Bücher über Scala veröffentlicht, darunter das Werk von Martin Odersky (1), dem Scala Erfinder selbst, sowie auch vom Erfinder des Lift Web Framework David Pollak (2) um nur zwei zu nennen. Somit steht für das Studium der Programmiersprache genügend gedruckte Literatur zur Verfügung. Ebenfalls existieren viele nützliche Quellen auf dem Internet. Scala hat bereits seine Runden durch die Java respektive Programmier Community gezogen und seine Spuren hinterlassen.

Das Einarbeiten an sich ist somit kein Hindernis da ausreichend Quellen vorhanden sind. Mittlerweile hat Scala auch einen ziemlich guten Reifegrad. Es werden diverse Open Source Frameworks mit Scala entwickelt und auch in der Wirtschaft gibt es immer mehr Firmen welche Scala in Ihrer Programmierabteilungen einsetzen.

Das Studium der theoretischen Grundlage der Programmierung in Scala durch einschlägige Literatur ist die eine Seite um sich die notwendigen Kenntnisse dieser Technologie zu erarbeiten. Als funktionale Programmiersprache bringt Scala viele Features mit, welche herkömmliche Programmiersprachen nicht bieten. Sich mit diesen funktionalen Elementen vertraut zu machen und sie entsprechend anwenden zu lernen ist die andere Seite. Dieser Prozess ist umfangreicher als das eigentliche Studium der Theorie. Diese Kenntnisse nehmen mit der Programmier- bzw. Anwendungserfahrung von Scala zu. Die Fähigkeit funktionsorientiert und nicht „bloss“ objektorientiert zu denken muss dabei entwickelt werden. Die Vorteile welche durch den Einsatz von Scala zum Vorschein kommen, können nur erreicht werden wenn diese Kenntnisse erarbeitet werden. Dann erlaubt Scala das Einsetzen von neuen Methoden in der Softwareentwicklung welche dichteren und schöneren Code zur Folge haben. Dies wiederum führt zu einer höheren Produktivität und zu sauberer und besserer Software.

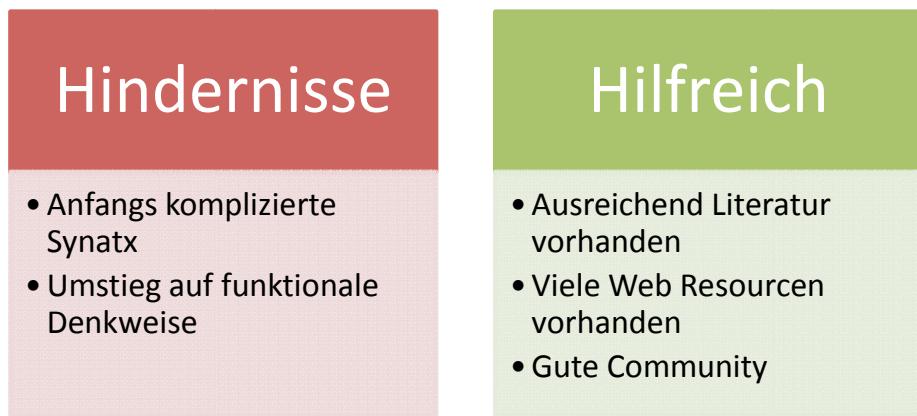


Abbildung 29: Hindernisse und Hilfreiche Dinge bei der Einarbeitung in Scala

Die Abbildung 29 fasst die Hindernisse, sowie auch hilfreiche Dinge bei der Einarbeitung in Scala zusammen.

### 8.1.2 Einarbeitungsphase für Lift

Bei der Einarbeitung in Lift muss unterschieden werden, ob bereits Scala Kenntnisse vorhanden sind, oder diese parallel zur Einarbeitung in Lift erworben werden sollen. Wenn bereits Scala Kenntnisse vorhanden sind findet man sich natürlich viel schneller in Lift zurecht. Der anfangs etwas verwirrende Scala Syntax stellt beim Lesen von Source Code keine Hürde mehr dar. Der rege Gebrauch von allen möglichen Scala Features in Lift kann mit Vorkenntnissen ebenfalls schneller durchschaut werden.

Wenn man über keine Scala Kenntnisse verfügt, gestaltet sich die Einarbeitung in Lift etwas umfangreicher. Lift hat noch Dokumentationslücken. Die Source Code Kommentare sind zwar zu einem grossen Teil vorhanden, jedoch existiert noch ein Mangel an Einführungsliteratur. So ist es erstmal schwer sich einen Überblick zu verschaffen, wie Lift eigentlich funktioniert und was für Prinzipien es verfolgt. Wenn man neu in Scala ist, dann hat man oft Mühe die kryptisch wirkende Scala Syntax zu verstehen. Somit ist auch Beispielcode von Lift Web Applikationen oder Open Source Projekten welche in Lift geschrieben wurden anfangs nur eine bedingte Hilfe, da der Source Code zu einem grossen Teil noch nicht erfasst werden kann. Wenn man einmal das Big Picture von Lift erfasst hat, dann findet man sich auch leichter in den Bibliotheken von Lift zurecht und man kann beginnen die Mächtigkeit des Frameworks auszuschöpfen. Wenn diese Einstiegsschwelle überschritten wurde kann man bereits sehr produktiv in Lift entwickeln.

Der angesprochene Mangel an ausreichender Literatur gestaltet die Einarbeitung in Lift etwas aufwändiger als jene in Scala. Die Zusammenarbeit mit der Lift Community (13) ist sehr fruchtbar. Die Entwickler sind offen für Vorschläge und auch hilfsbereit. Wenn man kommerzielle Software entwickelt und dazu Lift einsetzt, haben die Lift Entwickler auch ein offenes Ohr für Feature Vorschläge. Voraussetzung dafür ist eine Beteiligung am Projekt (Mitentwicklung, Dokumentation, ...). Für grössere Firmen

welche bereits Lift einsetzen, ist dies zweifelsohne ein Vorteil da eigene Wünsche ins Framework eingebracht werden können.

Der Reifegrad von Lift ist noch nicht auf jenem von Scala selbst. Doch das Framework baut auf eine solide Basis und verfügt über sehr innovative Aspekte welche sicherlich ein Fortschritt für die Webentwicklung darstellen. Die Einarbeitung in Lift lohnt sich. Wenn man mit dem Framework vertraut ist, dann kann äusserst produktiv entwickelt werden. Das Framework erlaubt das Erstellen von qualitativ sehr hochstehenden Web Applikationen welche dem modernen Standart mehr als gerecht werden.

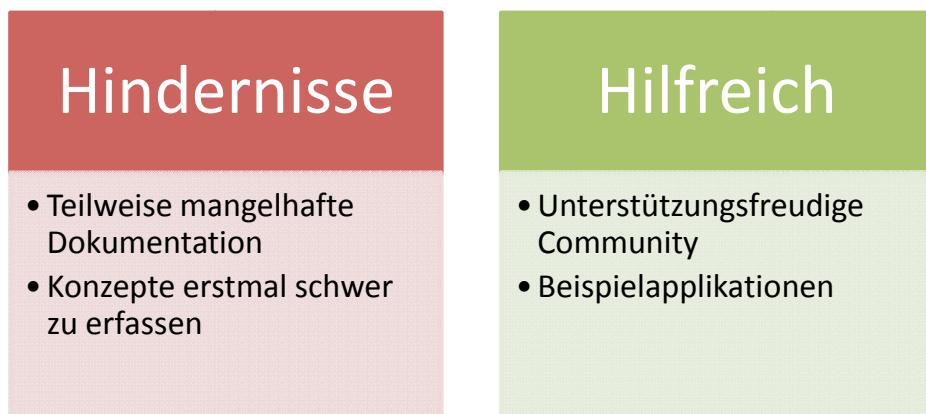


Abbildung 30: Hindernisse und Hilfreiches bei der Einarbeitung in Lift

Die Abbildung 30 fasst die Hindernisse, sowie auch Hilfreiche Dinge bei der Einarbeitung in Lift zusammen.

### 8.1.3 Lernkurve für Java Entwickler

In den vorhergehenden Abschnitten wurden bereits einige Aussagen über den Lernprozess von Scala und Lift gemacht. Dieser Prozess beschreibt das Einarbeiten in diese Technologie etwas genauer.

Die beiden Technologien können in Bezug auf die Lernkurve in verschiedene Kategorien aufgeteilt werden. Die folgende Tabelle 21 listet diese Kategorien auf.

Technologie	Kategorie	Beschreibung
Scala/Lift	Funktionale Konzepte	Funktionale Programmierung von der konzeptuellen Sicht. Die eigene Denkweise muss entsprechend revolutioniert werden.
Scala	Theoretische Grundlagen	Die theoretischen Grundlagen wie Sprachelemente, Syntax etc. müssen erarbeitet werden.
Lift	Konzepte	Die Konzepte des Lift Web Framework müssen erfasst und verstanden werden, damit man in der Lift Denkweise Arbeiten kann.
Lift	Funktionsumfang	Über den Funktionsumfang von Lift muss man sich einen Überblick schaffen damit man erkennen kann wie man spezifische Probleme am besten löst.
Scala/Lift	Entwicklungspraxis	Für Scala wie auch für Lift muss Entwicklungspraxis gesammelt werden damit man sich in dieser Umgebung wohl fühlt.

Tabelle 21: Unterteilung des Lernprozesses von Scala und Lift

Die Einarbeitungsphase in Scala und Lift kann durch diese fünf Kategorien beschrieben werden. Je nach Typ des Entwicklers können einzelne dieser Kategorien mehr oder weniger Zeit in Anspruch nehmen. Die folgenden Abschnitte sind je einem Aspekt des Einarbeitungsprozess zugeordnet.

## Funktionale Konzepte

Einige funktionale Konzepte hat man schnell erfasst und man kann sie auch schnell Umsetzen. Andere jedoch lernt man erst mit der Zeit kennen. Ein guter Ansatz ist wenn man sich fragt ob für die Lösung eines Problemes für welches man den konventionellen objektorientierten Ansatz kennt eine bessere Lösung mit funktionaler Programmierung möglich wäre. Entsprechende Recherchen können auch zu guten Resultaten führen. Für dieses allgemeine Thema gibt es viel Literatur die zur Unterstützung hinzugezogen werden können. Dieser Aspekt ist ein kontinuierlicher Lernprozess. Für den Einstieg reicht eine grobe Übersicht welche man recht schnell gewonnen hat. Will man das maximum aus einer funktionalen Sprache hole ist ein ausführlicheres Studium der funktionalen Aspekte unabdingbar.

## Theoretische Grundlagen von Scala

Um die Grundlagen von Scala zu erarbeiten kann man auf eine gute Auswahl von Hilfsmittel zurückgreifen. Dieser Aspekt ist somit ziemlich schnell abgedeckt. Um sich tiefer mit der Sprache vertraut zu machen hilft es wenn man die Entwicklung von Scala mitverfolgt. So wird man immer tiefer in die Programmiersprache eingeführt und man bekommt ein besseres Verständnis für die Sprache.

## Konzepte von Lift

Die Konzepte von Lift zu verstehen bildet die Grundlage der Arbeit mit dem Framework. Über einige Konzepte wie das View First kann man schnell einen groben Überblick gewinnen. Einige Konzepte wie das handling des Request Cycle wird man erst später verstehen können. Ebenfalls nimmt das Verständnis für Konzepte und deren Tiefe mit der Zeit immer zu. Dieser Punkt benötigt einiges an Zeit und kann zu den Aufwendigsten gezählt werden.

## Funktionsumfang von Lift

Den Funktionsumfang eines Frameworks zu kennen ist ebenfalls eine wichtige Grundlage. Die Stärken, Schwächen und Limitationen zu kennen sind für die Entwicklung wichtig. Sie erlauben richtige Entscheidungen in Bezug auf eine Problemlösung zu fällen. Wenn man die Lift Entwicklung ein wenig verfolgt wächst das Verständnis für das Framework stetig und man bekommt einen guten Überblick des Funktionsumfangs. Dieser Aspekt benötigt nicht so einen grossen Zeitaufwand.

## Entwicklungspraxis

Die Entwicklungspraxis in Scala sowie in Lift ist der grösste Punkt der Einarbeitungsphase. Erst mit Entwicklungserfahrung fängt man sich in dieser Umgebung wohl zu fühlen. Das Anfreunden mit Scala Code und dem Lift Framework ist ein muss wenn man damit entwickeln will. Für das braucht es viel Praxis, das heisst programmieren und noch einmal programmieren.

## Fazit

Da unsere gesamte Arbeit ein Technologiestudium war, kann man eigentlich die gesamte Projektdauer als Einarbeitungsphase betrachten. Auch hat man nie ausgelernt, vor allem wenn eine Technologie noch in der Entwicklung ist und ständig an Funktionsumfang zunimmt. Die folgende Abbildung 31 visualisiert die Hierarchie der Einarbeitung in Scala/Lift für ein besseres Verständnis.

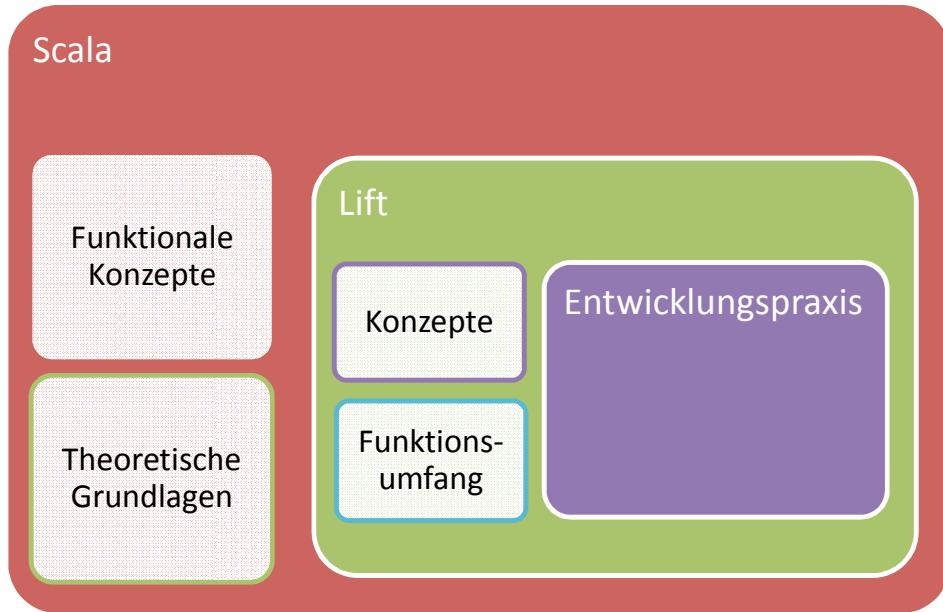


Abbildung 31: Strukturierung der Einarbeitung in Scala/Lift

Man kann sagen dass wir etwa nach 6-8 Wochen die Grundlagen erarbeitet hatten und auf einem Stand waren, wo wir produktiv entwickeln konnten. Während diesen 6-8 Wochen haben wir natürlich nicht Vollzeit gearbeitet. Für die Einarbeitungsphase benötigten wir etwa 60-80% unserer Zeit während diesen 6-8 Wochen. Ein Entwickler welcher ca. 50% seiner Zeit für die Einarbeitung aufwenden kann sollte in einem bis zwei Monaten auf diesem Level sein. Diese von uns gemachte Aussage wird durch einen aktuellen Erfahrungsbericht (41) aus dem Hause Novell bestätigt. Dieser Bericht stammt aus den Erfahrungen mit der Entwicklung von Pulse, einer Webapplikation realisiert mit Lift. Novell schreibt, dass ein Team von kompetenten Entwicklern in ca. ein bis zwei Monaten voll Entwicklungsfähig sein soll. Weiter ist im Bericht erwähnt, dass der Umstieg zu Scala/Lift im Bezug auf das Produkt ein grosser Vorteil war.

### 8.1.4 Zukunft von Scala

Fakt ist, dass die Zukunftsentwicklung von Programmiersprachen in die Richtung der funktional orientierten Programmiersprachen geht. Microsoft geht mit C# diesen Weg und Sun/Oracle möchte mit Java 7 ebenfalls funktionale Features in ihre Programmiersprache einzubauen. Da Java aber ursprünglich nicht mit diesem Gedanken designet wurde, tut sich Sun/Oracle mit der Entwicklung von Java 7 schwer und bisherige Previews von Java 7 Features waren eher ernüchternd. Und das ist genau der Punkt wo Scala ins Spiel kommt. Als moderne, funktionale und vollständig objektorientierte Programmiersprache ist sie eine ideale Ergänzung zu Java. Die Entwicklung von Scala wird rapide voran getrieben (im Gegensatz zu Java) und Scala bekommt immer mehr Aufmerksamkeit. Es ist davon auszugehen, dass Scala viel mehr Potential hat als beispielweise Groovy oder andere Java Erweiterungen. Die Popularität der Sprache wird zweifelsohne stärker zunehmen und Scala hat eine Chance als akzeptierte Programmiersprache aufgenommen zu werden. Java bildet

dabei auch in Zukunft als stabile Laufzeitumgebung eine wichtige Basis für den Erfolg von Scala.

## 8.1.5 Zukunft von Lift

Zurzeit ist Lift das umfangreichste und am weitesten verbreitete Web Framework welches in Scala geschrieben ist. Die Zahl der Web Frameworks in Scala wird mit der steigenden Popularität von Scala sicherlich noch zunehmen. Lift verfügt über eine solide Architektur und die vielen innovativen Features machen es äusserst interessant. Mittlerweile ist der erste Release Candidate RC 1 von Lift 2.0 erschienen und es existieren bereits einige kommerziell grosse Webseiten welche in Lift implementiert wurden. Somit hat Lift die Feuertaufe bereits überstanden und es kann von einem soliden Web Framework mit Zukunft gesprochen werden. Die steigende Mitgliederzahl der Lift Community lässt auf eine weitere Verbreitung schliessen. Mit zunehmender Zahl von Mitarbeitenden sieht es auch so aus, also ob zurzeit noch existierende Schwachstellen, in Zukunft mit mehr Manpower in Angriff genommen werden können. Lift hat das Potential bestehende Java Web Frameworks wie z.B. JSF oder Struts abzulösen. Gegenüber diesen bietet Lift dank seiner Innovativität und dem Fakt, dass es Scala basiert ist, einige Vorteile. Lift ist ebenfalls eine Alternative zu Ruby On Rails. Der Funktionsumfang ist mit demjenigen von Ruby absolut zu Vergleichen. Lift selbst verfolgt ebenfalls einige Ansätze welche Ruby On Rails eingeführt hat. Die Kompatibilität mit der JVM Laufzeitumgebung ist als klarer Vorteil gegenüber der Laufzeitumgebung von Ruby On Rails zu werten.

Wie sich der Markt effektiv entwickeln wird ist schwer vorauszusehen. Lift verfügt jedoch über das Potential sich im heutigen „Web Framework Wettkampf“ zu behaupten und hat gute Chancen sich grossflächig durchzusetzen.

## 8.1.6 Empfehlung

Wir haben Lift als ein sehr mächtiges Web Framework mit innovativen Konzepten kennengelernt. Bei der Entwicklung konnten wir feststellen, dass uns Lift äusserst produktiv arbeiten lässt, ohne dabei die Entscheidungsfreiheit für Problemlösungen nennenswert zu beschränken. Modern Web Applikationen müssen interaktiv sein. Für den Web 2.0 Ansatz bietet Lift eine ausgezeichnete Integration und Unterstützung. Gerade Web Applikationen die auf diese Features angewiesen sind können von Lift in grossem Masse profitieren.

Es wurden bereits diverse Aussagen über Stärken und Schwächen sowie die Möglichkeiten von Lift und Scala gemacht. Deshalb möchten wir hier unsere Empfehlung auf den Punkt bringen.

Gemäss unserer Erfahrung konnte Lift alle gemachten Versprechen halten und hat uns von seinen Qualitäten überzeugt. Unser Gesamteindruck von Lift ist durchwegs positiv und wir können den Einsatz von Lift für die produktive Entwicklung von modernen Web Applikationen nur befürworten.

## 8.1.7 Offene Punkte

In diesem Unterabschnitt werden mögliche Ausbaumöglichkeiten und Known Issues (Bekannte Fehler/Schwächen) beschrieben.

### Ausbaumöglichkeiten

Im Folgenden werden Ausbaumöglichkeiten und weitere Aspekte aufgelistet, die wichtige oder interessant wären weiter zu verfolgen.

---

#### Simple Build Tool



SBT (Simple Build Tool) wird in Zukunft das Default Build Tool für Lift Projekte. Da die Beispielapplikation auf einem Maven Archetype basiert und SBT erst seit einigen Wochen immer mehr aufkommt, wurde SBT im Technologiestudium nicht behandelt. Da es in künftigen Releases das Default Build Tool sein wird würde es Sinn machen SBT detaillierter anzuschauen.

---

#### Logging



Für eine produktive Applikation ist Logging ein essentieller Bestandteil. Dieser Aspekt wurde nicht im Rahmen der Beispielapplikation behandelt. Es gibt in Lift eine Schnittstelle für die SLF4J Facade. Diese wird in einem Artikel (42) auf dem Lift Wiki behandelt.

### Known Issues

Im Folgenden werden bekannt Bugs der Beispielapplikation aufgelistet.

---

#### REST API



Die RestAPI ist nicht vollkommen funktionsfähig aufgrund von Fehlern im Pattern Matching in Scala. REST Methoden welche zu Fehlern führen wurden deshalb auskommentiert. Dieser Fehler ist bekannt und es existiert auch bereits ein Ticket (43) auf dem Scala Trac. Es gibt auch ein Workaround (Beim Ticket beschrieben) welcher in der Beispielapplikation aber nicht umgesetzt ist.

---

#### Blog



Auf der Blog Seite (Single Page Application) werden unter dem Punkt „Einträge anderer Mitglieder“ auch die Einträge des aktuellen Users angezeigt. Das Problem konnte nicht eruiert werden, hat aber vermutlich mit einem ungültigen Session Scope zu tun.

---

**Exception Handling des EntityConverter**

Der EntityConverter welcher für die Umwandlung von Domain Entites zu XML verantwortlich ist, verfügt über kein durchgehendes Exception Handling. Durch fehlerhafte XML Eingaben über die REST API kann ein inkonsistenter Applikationszustand erreicht werden. Für die Lösung dieses Problems müsste ein vollständiges Exception Handling implementiert werden.

---

---

**Mangel an automatischer Testabdeckung**

Die Beispielapplikation verfügt nur über wenige, punktuelle JUnit Testklassen. Der Auftraggeber legte keinen Wert darauf sondern bevorzugte manuelle Tests mit dem Browser. Da die Applikation nur ein Beispiel ist und die Komplexität nicht allzu gross ist dieser Ansatz vertretbar. Für eine Applikation die produktiv eingesetzt werden sollte wäre dies ein Mangel.

---

## 8.2 Persönliche Erfahrungsberichte

In diesem Abschnitt sind die persönlichen Erfahrungen der Projektmitglieder Daniel Hobi und Ralf Muri dokumentiert. Die Abschnitte sind jeweils gemäss den Bedürfnissen der Projektmitglieder unterschiedlich strukturiert.

### 8.2.1 Daniel Hobi

In den letzten 16 Wochen habe ich zweifellos eine Menge Erfahrung gesammelt. Diese sollen nun niedergeschrieben werden. Dabei versuche ich die gemachten Erfahrungen in drei Kategorien aufzuteilen. Die Kategorie Arbeit befasst sich mit dem Ablauf und den Erfahrungen der Bachelorarbeit. Die Kategorie Entwicklersicht befasst sich mit dem Umstieg von Java zu Scala. In der Kategorie Webprogrammierer lege ich meine gemachten Erfahrungen im Bereich Webprogrammierung dar. Zum Schluss ziehe ich ein persönliches Fazit.

#### Projektverlauf

Als interessierter Webprogrammierer habe ich mich mit Ralf Muri um die Arbeit „Webapplikation mit Scala / Lift“ beworben. Es ist bereits die dritte Arbeit, welche ich mit ihm durchführte. Ich wusste deshalb die Zusammenarbeit mit Ralf im Vorherein abzuschätzen. Im Gegensatz zu den letzten zwei Arbeiten, waren wir dieses Mal ein 2er Team, was bis auf die Aufgabenverteilung aber keine Unterschiede ausmachte.

Zu Beginn der Arbeit konzentrierten wir uns ganz auf das Studium von Scala und Lift. Während sich Ralf mehr um die Scala Seite schlug, versuchte ich mich mehr dem Lift Framework anzunehmen. Es wurde uns schnell bewusst, dass eine derartige Aufteilung nur wenig Sinn machte, da Lift häufigen Gebrauch der Features von Scala macht. So mussten sich beide sowohl in Scala als auch in Lift einlesen.

Wir waren in der Projektplanung und Projektgestaltung ziemlich frei, da wir uns nicht an detaillierte Richtlinien halten mussten. Anstatt einer fertigen Applikation oder Prototyps stand die Erfahrung und die Einstufung der Technologie Scala / Lift im Vordergrund.

Anstatt einer von Grund auf neu entwickelten Applikation entschieden wir uns dafür, eine Applikation aus einer Vorarbeit umzusetzen. Die Zeit, welche wir für das Spezifizieren einer neuen Applikation benötigt hätten, konnten wir so zusätzlich für das Technologiestudium verwenden.

Die ersten Feature Iterationen liefen überhaupt nicht rund. Nichts lief auf den ersten Versuch. Immer wieder traten Probleme auf. Auch mit git als Version Control hatten wir anfangs zu kämpfen. Die Situation wurde besser, als wir eine Woche Iterationsstopp einlegten und uns nur um die Beispielapplikation kümmerten. In dieser Woche konnten wir sowohl den Rückstand, der sich in den letzten Wochen aufsummiert hat, wieder aufholen. Diese Woche kann daher gut als Wendepunkt in unserer Arbeit beschrieben werden. Natürlich traten nach dieser Woche noch hie

und da Probleme auf. Die Einstellung mit welcher ich den Problemen begegnete, war dann aber durchwegs positiv.

Interessant und gefährlich zugleich war die Verwendung der neusten Scala und Lift Version. Einerseits war es eine tolle Erfahrung, den Puls der Lift Community zu spüren. Andererseits konnte teils nicht genau erörtert werden, ob es sich um einen Programmierfehler oder um ein Bug in Scala bzw. Lift handelt. In diesem Zusammenhang war die Lift group auf google group eine wertvolle Anlaufstelle, da sie sich offen und hilfsbereit zeigte.

Die 16 Wochen waren sowohl von Erfolgen als auch von anfänglichen Misserfolgen geprägt. Mit dem Endprodukt dieser Arbeit bin ich durchwegs zufrieden und an einigen Stellen auch etwas stolz.

## Entwicklersicht

Seit ca. 2.5 Jahren beschäftige ich mich nun mit Java. Ich erinnere mich noch gut, wie es mir anfänglich erging. Ich hatte etwas Mühe die objektorientierte Programmierung zu verstehen. Durch viel Übung und der detaillierten Auseinandersetzung mit Java Code wurde dies aber allmählich besser und ich fragte mich im Nachhinein oft, wieso ich nicht schon früher objektorientiert programmiert habe.

Mit der Erlernung von Scala erging ist mir in etwa gleich. Nur hiess der Gegner nicht objektorientierte sondern funktionale Programmierung. Auch die neuen Features wie Pattern Matching oder die implizite Konvertierung stiessen mir am Anfang etwas sauer auf. Besonders das Pattern Matching war mir einfach zu „magisch“. Die für mich anfänglich zu wenig strikte Syntax, Methoden welche aus einem Fragezeichen oder Schrägstrich bestehen, das Underscore Zeichen, der Pfeil aus Gleichheitszeichen und „Grösser als“ – Zeichen und Traits sind einige Scalaeigenschaften, welche mich zu Beginn vor ein grosses Verständnisproblem stellten. Die Einstiegshürde war verhältnismässig gross, weiss Scala doch wesentlich mehr zu bieten als Java. Einmal etwas eingearbeitet, erweist sich Scala jedoch als reifere Programmiersprache. Unnötiges von Java wurde weggelassen, eine Menge nützlicher Features hinzugefügt. Programme können mit weniger Codeaufwand geschrieben werden, was die Leserlichkeit erhöht und die gleichzeitig die Fehlerquote deutlich senkt. Sollten dennoch alle Stricke reissen, erweist sich die Kompatibilität zu Java auf Bytecode Ebene als Rettung. In Scala zu programmieren schon die Nerven und macht Spass.

## Webprogrammierer

Ich erstelle seit ca. 8 Jahren regelässig Websites und betreibe selber eine Domain, mit welcher ich mich als Websitenersteller anerbiete. Die bisher erstellten Seiten reichen von kleinen, statischen Seiten bis zu einem eigens programmierten Shopsystem.

Wie wohl viele andere auch war meine gewählte Sprache PHP. Sie ist weit verbreitet und meiner Meinung nach immer noch einer der am besten dokumentierten Sprachen überhaupt. Ich begann im Selbststudium PHP3 zu programmieren und

hatte offen gesagt keine Ahnung von der Materie. Vieles hatte ich mir einfach zusammen kopiert.

Mit dem Eintritt ins Studium und dem Release von PHP5 wurde der Einsatz der Objektorientierung gefördert, was nicht nur mir meinen Fähigkeiten, sondern auch meinen Projekten zugutekam. Obgleich sich die Programmierung verbesserte und zwischen sauberem und unsauberem Code endlich unterschieden wurde, begann mich die Webprogrammierung etwas zu langweilen. Immer dieselben CRUD Implementationen. Immer derselbe Ablauf von Request und Response. Natürlich hätte ich zu diesem Zeitpunkt auf ein geeignetes CMS (TYPO3) bzw. Framework (ZEND) umsteigen sollen, anstatt immer wieder von Scratch zu beginnen. In der Befürchtung den Workflow nicht mehr verstehen zu können, liess ich es aber immer wieder bleiben. Für die Erstellung eines eigenen Frameworks fehlten mir die Motivation und auch das Wissen.

In letzter Zeit wurde die Webentwicklung wieder etwas interessanter. Ich begann mich mit dem Javascript Framework Dojo auseinanderzusetzen und war von Anfang an begeistert. Kurz darauf entwickelte ich meine erste Single Page Application. In einem eigenen, grösseren Projekt nahm ich mir wiederum vor, eine Single Page Application zu erstellen. Langsam aber sicher wurde mir bewusst, dass ich nach wie vor vielmehr Code schrieb, als eigentlich nötig ist.

Mit Scala und Lift erwartete mich sowohl eine neue Programmiersprache als auch mein erstes Webframework. Aus bereits genannten Gründen war ich ein wenig skeptisch gegenüber Lift. Ich hoffte, den Einstieg schnell finden zu können.

Anders als erhofft war der Einstieg recht harzig. Den gesuchten Workflow von Request und Response konnte ich nicht finden, das Lift Buch überforderte mich. Dass Lift weg vom Denken des Request und Response führte und dass dies sehr wohl Sinn macht, wurde mir erst mit der Zeit bewusst.

Nach und nach machte es immer mehr Spass mit Lift zu programmieren. Zu einer programmierten Zeile in Lift, welche im Schnitt fünf Kompilierversuche benötigten, um zu laufen, wurden allmählich 10-20 Zeilen mit einem Versuch. Das neuartige Bindverfahren von Lift wurde von unnötig kompliziert zu ungeheuer mächtig. Die Ajax und Comet Unterstützung von Lift liessen es mich schliesslich akzeptieren: Einmal verstanden, ist Lift ein Segen!

Mit zahlreichen Erfahrungen in Lift sehe ich nun zurück auf meine bisherigen Projekte. Hie und da rümpfe ich die Nase. Ging das nicht viel effizienter mit Lift? Natürlich tut es das. Wieso das Projekt nicht einfach portieren? Nun, vielleicht werde ich das sogar: Ein PHP zu Scala Tool soll es ja bereits schon geben.

## Fazit

Von allen bisherigen Projekten war dieses wohl das an Erfahrungen reichste. Viele Stunden Schweissarbeit sind in dieses Dokument und natürlich auch in die Beispielapplikation geflossen. Ich empfehle jedem Java Webentwickler Scala / Lift

eine Chance zu geben. Es mag am Anfang hart sein, die Mühe zahlt sich aber vollends aus.

## 8.2.2 Ralf Muri

Die folgenden Abschnitte fassen die von mir gemachten Erfahrungen während unserer Bachelorarbeit „Webapplikation mit Scala/Lift“ zusammen. Ich habe meinen Erfahrungsbericht in drei Abschnitte aufgeteilt. Zuerst werde ich meine Erfahrungen bezüglich des Projektverlaufes äussern. Anschliessend beschäftige ich mich mit dem eigentlichen Inhalt der Arbeit. Ein kurzes Fazit welches meine Erfahrungen aufsummiert bildet den Abschluss meines Erfahrungsberichtes.

### Projektverlauf

„Webapplikation mit Scala/Lift“ war unter den Bachelor Themen für welche wir uns beworben haben mein Favourit. Somit war ich über die Zuteilung dieser Arbeit an unsere Gruppe sehr erfreut. Mit meinem Teampartner Daniel Hobi habe ich bisher alle grösseren Projektarbeiten durchgeführt. Wir waren bereits ein eingespieltes Team und jeder wusste wo des anderen Stärken und Schwächen lagen. Entsprechend viel uns die Arbeitsaufteilung während des Projekts auch ziemlich leicht und jeder konnte seine Fähigkeiten gezielt einsetzen.

Aufgrund der Natur unserer Arbeit (Technologiestudium) konnten wir ein ziemlich flexibles Prozessmodell für die Projektdurchführung wählen. Dies liess uns sehr viel Freiheit in der Arbeitsgestaltung. Rückblickend bin ich mit unserer Projektplanung sehr zufrieden. Die Strukturierung in Feature Iterationen liess uns gezielt und problemorientiert arbeiten. Lediglich der Einstieg war ein bisschen harzig da es doch einiges an Einarbeitungszeit benötigt bis man sich in Scala und Lift zurecht findet.

Die Zusammenarbeit mit Paul Bernet von Crealogix E-Business AG war sehr angenehm. Während den Sitzungen konnten wir immer viele konstruktive Inputs mitnehmen und in das Projekt einfließen lassen. Ebenfalls äusserst hilfreich waren die Reviews der bisherigen Arbeit (Code & Dokumentation), welche in der zweiten Projekthälfte durch Paul Bernet durchgeführt wurden. Durch das rege Interesse des Industriepartners an unserem Projekt wurde dies auch viel lebendiger und interessanter für uns.

Die Betreuung von Seite der Schule war mich ebenfalls ideal. Die wöchentlichen Sitzungen mit Prof. Hans Rudin trugen dazu bei das Projekt am laufen zu halten und uns immer wieder der Planung nach auszurichten. Die Reviews unserer Projektplanung und Dokumentation trugen dazu bei diese sinnvoll, zweckgemäss und inhaltlich vollständig zu gestalten. Die Zwischenpräsentation vor unserem Gegenleser Prof. Beat Stettler bereicherte das Technologiestudium mit seinen Inputs um weitere sehr sinnvolle Aspekte.

Insgesamt habe ich sehr viel Zeit in das Projekt investiert (mehr als im Rahmen der Bachelorarbeit vorgesehen). Den tatsächlichen Aufwand des Projekts kann man nicht unbedingt am effektiven Endresultat erkennen. Ein nennenswerter Teil der Arbeit

waren Arbeiten dich nicht direkt ein dokumentierbares Ergebnis lieferten. Trotzdem fand ich es ein sehr spannendes Projekt und aufgrund meines persönlichen Interessens war ich gern bereit diesen Aufwand zu betreiben.

## Technologie Scala/Lift

Scala und Lift waren für mich absolutes Neuland und vor der Bachelorarbeit hatte ich weder von Scala, noch von Lift je gehört. Funktionale Programmierung durften wir in einem Modul an der Schule mit C# bereits ein bisschen kennenlernen.

Die ersten Berührungen mit Scala waren nicht ganz der einfach. Der im Gegensatz zu Java einiges kompliziertere und vielfältigere Syntax ist Anfangs eine echte Hürde. Auch nach dem Studium eines Scala Buches war diese Hürde noch längst nicht überschritten. Erst das Entwickeln in Scala liessen mich langsam aber sicher den Scala Code als angenehm zu empfinden. Einigermassen schnell erkannte ich was für neue Möglichkeiten mir mit funktionaler Programmierung und anderer Features von Scala geboten wurde. Auch wenn ich vieles noch nicht verstand war ich ziemlich begeistert von Scala.

Mit Lift war die Sache ziemlich ähnlich. Vom Konzept her recht verschieden von den mir bekannten Web Frameworks (JSP, JSF, Struts) fehlte mir Anfangs vor allem die Übersicht. Ich wollte viel zu genau wissen wie das ganze funktioniert. Erst als ich mir langsam abgewöhnte immer wissen zu wollen wie jetzt genau ein Request Cycle aussieht und was tatsächlich passiert konnte ich mich mit den Lift Konzepte anfreunden und ein tiefes Verständnis des Frameworks erarbeiten. Ähnlich wie bei Scala war ich ziemlich schnell von den Innovationen und ausgeklügelten Lösungen für bestimmte Probleme von Lift begeistert. Auch wenn ich erst einen ziemlich beschränkten Überblick des Frameworks erarbeitet hatte, war ich schon sehr begeistert.

Das Interesse an beiden Technologien, wie auch die Begeisterung haben bis jetzt nicht abgeklungen. Mit der weiteren Entwicklungspraxis fing ich langsam an mich in diesem Umfeld zu Hause zu fühlen und ich konnte anfangen sehr produktiv zu arbeiten. Die Vorteile von Scala und Lift haben mich überzeugt und wiegen meiner Meinung nach wesentlich mehr auf als die Probleme die der Umstieg mit sich bringt.

Mittlerweile würde ich Lift jedem anderen Web Framework vorziehen (natürlich habe ich damit nun auch am meisten Erfahrung sammeln dürfen). Scala ist für mich äußerst interessant und ich würde mich gerne tiefer darin einarbeiten. Künftige Projekte, für welche ich für den Technologieentscheid (mit-)verantwortlich wäre, würde ich in Java mit Scala als Ergänzung realisieren. In Java verfüge ich über viel mehr Entwicklungserfahrung und kann auch sehr produktiv arbeiten. Die Vorteile von funktionaler Programmierung für die Lösung bestimmter Probleme möchte ich mir aber nicht entgehen lassen. Das parallele Verwenden beider Sprachen dank der einwandfreien Kompatibilität macht Java mit Scala zu einem sehr mächtigen Werkzeug.

## Fazit

Abschliessend kann ich sagen dass ich mit dem Projekt sehr zufrieden bin. Der Inhalt der Arbeit war für mich viel interessanter als ich es zu Beginn erahnen konnte. Die Möglichkeit sich während der Bachelor Arbeit in eine neue Programmiersprache und ein neues Web Framework einzuarbeiten fand ich äusserst wertvoll. Beide Technologien könnten Zukunftsweisend sein. Wir hatten die Gelegenheit uns in diese Technologien einzuarbeiten. Ein solches Einarbeiten wäre in einem anderen Rahmen wohl nicht möglich gewesen. Ich konnte von dieser Arbeit sehr viel für mich selber profitieren.

Der Projektverlauf erfolgte für uns nach Plan und mit dem Ergebnis bin ich recht zufrieden. Wir konnten die Anfangs definierten Anforderungen erfüllen und das Projekt nahm sogar noch etwas an Umfang zu als ursprünglich geplant. Die Zusammenarbeit mit Paul Bernet von Crealogix E-Business AG war äusserst fruchtbar und für mich stehts angenehm und zufriedenstellend.

Die Arbeit hat mir derart gut gefallen, dass ich eigentlich gerne im selben Rahmen noch weiterarbeiten würde. Es gäbe noch einige interessante Aspekte zu entdecken und auch sonst hätte ich noch einige Ideen welche man umsetzen könnte. Scala und Lift hat mich überzeugt und ich werde mich wohl künftig in meiner Freizeit ein bisschen damit beschäftigen.

## 8.3 Schlusswort

---

Abschliessend können wir sagen, dass wir mit dem Resultat unserer Arbeit zufrieden sind. Wir hoffen dass die Ergebnisse unserer Arbeit weiterverwendet werden können und sie für die Crealogix E-Business AG den gewünschten Nutzen bringen.

Wir möchten uns bei Herrn Paul Bernet für die konstruktive Zusammenarbeit bedanken. Die Besprechungen waren immer äusserst ergebnisreich und seine detaillierten Reviews verhalfen der Arbeit zu mehr Qualität. Ebenfalls bedanken möchten wir uns bei der Betreuung der Hochschule durch Prof. Hans Rudin und Dominik Wild.

Die Arbeit war für uns sehr interessant und wir konnten viele wichtige Erfahrungen sammeln. Die Zusammenarbeit im Team, mit dem Auftraggeber und den Betreuern war stets angenehm. Wir schätzen die uns gegebenen Freiheiten in Bezug auf die Gestaltung des Projekts.

## 9 Literaturverzeichnis

---

1. **Martin Odersky, Lex Spoon, Bill Venners.** *Programming in Scala*. s.l. : Artima Inc., 2008. 978-0981531601.
2. **Pollak, David.** *Beginning Scala*. s.l. : Apress, 2009. 978-1430219897.
3. **Scala.** The Scala Programming Language. [Online] Lausanne, École Polytechnique Fédérale de, 2010. <http://www.scala-lang.org/>.
4. —. The Scala Community. [Online] École Polytechnique Fédérale de Lausanne. <http://www.scala-lang.org/node/1707>.
5. —. Scala. [Online] École Polytechnique Fédérale de Lausanne. <http://lampsfn.epfl.ch/trac/scala>.
6. **Java to Scala with the Help of Experts.** [Online] École Polytechnique Fédérale de Lausanne. <http://www.scala-lang.org/node/960>.
7. **Perrett, Timothy.** *Lift in Action Early Access Edition*. s.l. : Manning, 2010. ISBN: 9781935182801.
8. **Chen-Becker, Derek, Danciu, Marius and Weir, Tyler.** *The Definitive Guide to Lift*. s.l. : Apress, 2009.
9. **Lift Community.** Lift Wiki. [Online] 2010. <http://www.assembla.com/wiki/show/liftweb>.
10. Step auf [github.com](http://github.com). [Online] Alan Dipert. <http://github.com/alandipert/step>.
11. pinky auf [github.com](http://github.com). [Online] Peter Kovac Hausel. <http://github.com/pk11/pinky>.
12. sweetscala. [Online] <http://code.google.com/p/sweetscala/>.
13. **Lift Community.** Lift Group. [Online] 2010. <http://groups.google.com/group/liftweb>.
14. **Hobi, Daniel and Muri, Ralf.** *Technische Aspekte*. Rapperswil : HSR, 2010.
15. **Scala.** Scala 2.8 Preview. [Online] École Polytechnique Fédérale de Lausanne. <http://www.scala-lang.org/node/1564>.
16. **Pollak, David.** Liftweb GitHub. [Online] 2010. <http://github.com/dpp/liftweb>.
17. **JetBrains.** JetBrains Plugin Repository. [Online] JetBrains. <http://plugins.intellij.net/plugin/?id=1347>.
18. **Apache Software Foundation.** Apache Maven. [Online] 2010. <http://maven.apache.org/>.
19. —. Apache. [Online] 2010. <http://www.apache.org>.
20. **Torvalds, Linus and Hamano, Junio.** Git - Fast Version Control System. [Online] <http://www.git-scm.com/>.
21. **GitHub.** Github - Social Coding. [Online] 2010. <http://www.github.com>.
22. **Muri, Ralf and Hobi, Daniel.** TravelCompanionScala - GitHub. [Online] 2010. <http://github.com/rmuri/TravelCompanionScala>.
23. **GitHub.** Generating SSH keys (Win/msysgit). [Online] 2010. <http://help.github.com/msysgit-key-setup/>.
24. **Lift Community.** Lift Mapper 2.0-M5 API. [Online] 2010. <http://scala-tools.org/mvnsites/liftweb-2.0-M5/framework/lift-persistence/lift-mapper/scaladocs/index.html>.

25. —. Lift WebKit 2.0-M5 API. [Online] 2010. <http://scala-tools.org/mvnsites/liftweb-2.0-M5/framework/lift-base/lift-webkit/scaladocs/index.html>.
26. **Sun.** Java Persistence API. [Online] 2010. <http://java.sun.com/javaee/technologies/persistence.jsp>.
27. **Java Community Process.** Java Specification Request JSR303. [Online] 2010. <http://jcp.org/en/jsr/detail?id=303>.
28. **JBoss Community.** Hibernate Validator. [Online] 2010. <http://www.hibernate.org/subprojects/validator.html>.
29. —. JSR 303 Reference Implementation Reference Guide. [Online] 2010. [http://docs.jboss.org/hibernate/stable/validator/reference/en/pdf/hibernate\\_reference.pdf](http://docs.jboss.org/hibernate/stable/validator/reference/en/pdf/hibernate_reference.pdf).
30. **Resig, John.** jQuery: JavaScript Library. [Online] 2010. <http://www.jquery.com/>.
31. **Yahoo!** YUI Library. [Online] 2010. <http://developer.yahoo.com/yui/>.
32. **Ext Team.** Ext Core: Cross-Browser JavaScript Library. [Online] 2010. <http://www.extjs.com/products/core/>.
33. **Russel, Alex.** *Comet: Low Latency Data for the Browser / Infrequently Noted*. [Online] [Cited: 06 14, 2010.] <http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/>.
34. **Lift Community.** Templates and Binding. [Online] 2010. [http://www.assembla.com/wiki/show/liftweb/Templates\\_and\\_Binding](http://www.assembla.com/wiki/show/liftweb/Templates_and_Binding).
35. **Suereth, Josh.** Using Partial Functions and Pattern. [Online] 2010. <http://suereth.blogspot.com/2008/11/using-partial-functions-and-pattern.html>.
36. **Spiewak, Daniel.** Interop Between Java and Scala. [Online] 2009. <http://www.codecommit.com/blog/java/interop-between-java-and-scala>.
37. **Odersky, Martin and Sabin, Miles.** JavaConversions. [Online] 2010. [http://www.scala-lang.org/archives/downloads/distrib/files/nightly/docs/library/scala/collection/JavaConversions\\$.html](http://www.scala-lang.org/archives/downloads/distrib/files/nightly/docs/library/scala/collection/JavaConversions$.html).
38. **Spiewak, Daniel.** Scala interop: transparent List and Map conversion. [Online] 2009. <http://stackoverflow.com/questions/1519838/1520179#1520179>.
39. **Pollak, David.** Scala-Tools Repository. [Online] 2010. <http://scala-tools.org/>.
40. **Hardegger, Stefan, Lutz, Cyril and Vogler, Thomas.** *GRails und das Springframework: Vergleichsstudie*. Rapperswil : HSR, 2008.
41. **Novell.** Novell Pulse - Lift at heart. [Online] <http://www.scala-lang.org/node/6618>.
42. **Hartmann, Mads.** Logging in Lift. [Online] 2010. <http://www.assembla.com/wiki/show/liftweb/Logging>.
43. **extempore.** nested extractors generate exponential-space bytecode. [Online] 2008. <https://lampsvn.epfl.ch/trac/scala/ticket/1133>.
44. **Subramaniam, Venkat.** *Programming Scala*. s.l. : Pragmatic Bookshelf, 2008.
45. **Pollak, David.** A Menu DSL. [Online] April 23, 2010. [http://groups.google.com/group/liftweb/browse\\_thread/thread/aabd67170f85f9f4](http://groups.google.com/group/liftweb/browse_thread/thread/aabd67170f85f9f4).
46. **Ortiz, Jorge.** scalaj-collection on GitHub. [Online] 2010. <http://github.com/scalaj/scalaj-collection>.