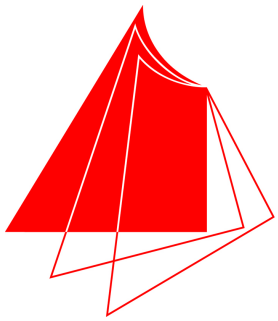


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Test-Driven Development mit Ruby on Rails

am Beispiel der o/ZB Webapplikation

Projektarbeit

vorgelegt von

Briewig, Martin, B.Sc. (Matr.-Nr.: 43509)

Leibel, Michael, B.Sc. (Matr.-Nr.: 43674)

Betreuer: Prof. Dr. Frank Schaefer

Fachbereich: Informatik (Master)

Inhaltsverzeichnis

Abkürzungsverzeichnis

AP Arbeitspaket

o/ZB Ohne Zins Bewegung

RoR Ruby on Rails

SSH Secure Shell

1 Einleitung

Die ohne Zins Bewegung Stuttgart arbeitet nun seit einigen Jahren mit der Hochschule Karlsruhe zusammen an einer Webanwendung. Diese Webanwendung soll die Geschäftsprozesse der Bewegung unterstützen und verwalten. Im Laufe der letzten Jahre ist aus der ursprünglichen Idee eine komplexe Webanwendung entstanden, die nun gründlich auf Fehler überprüft und gegebenfalls ausgebessert werden soll. Diese Aufgabe wird nun uns, Herr Briewig und Herr Leibel, anvertraut. Im Folgenden wird eine detaillierte Aufgabenstellung formuliert, an dieser sich diese Projektarbeit messen lassen wird.

1.1 Aufgabenbeschreibung

Die Aufgaben dieser Projektarbeit sind recht vielfältig. Sie lassen sich in 4 Phasen einteilen. Die Aufgaben pro Phase wurden maßgeblich von den Meetings geprägt.

1.1.1 Phase 1: Korrekturen, Dokumentieren

(Duedate: 02.05.2013)
Die erste Phase wird von den Korrekturen am ER-Diagramm und dem dazugehörigen Relationsmodell dominiert. Das ER-Diagramm weißt noch ein paar Unstimmigkeiten auf, die im Laufe der Zeit entstanden sind. Diese gilt es zu bereinigen und die vorgenommenen Korrekturen in der Implementierung umzusetzen. Ziel ist ein ER-Diagramm, dessen Entitäts-Beziehungen den nicht-historisierten Zustand darstellen. Dieses ER-Diagramm soll 1:1 in der Implementierung vorzufinden zu sein. Es soll vereinfacht dargestellt werden, lediglich die Primär- und Fremdschlüssel, die im historisierten Zustand gültig sind, sollen den Entitäten als Zusatzinformationen dienen.

Neben den Korrekturen am ER-Diagramm sind Dokumentationen der Vorgehens- und Funktionsweise des Datenbank Migrationstools notwendig. Hierfür sind zwei Batch-Skripte angefertigt worden, deren Nutzungs- und Funktionsweisen zu dokumentieren sind.

Im Laufe der Zeit sind die verschiedensten Techniken zur Bereitstellung und Versionierung der o/zb Webanwendung verwendet worden. Es gilt nun, einen einheitlichen Vorgang zu definieren der für die nachfolgenden Projektgruppen verwendet werden soll. Basierend auf der anzufertigten Dokumentation, kann die Bereitstellung und Versionierung stetig verbessert werden.

Zusätzlich soll in dieser Phase ein kritischer Fehler bereinigt werden. Der WebImport, welcher für den Import der Kontobewegungen verantwortlich ist, arbeitet nicht richtig. Zur Zeit ist der Datei-Upload auf dem Testsystem nicht möglich. Darüber hinaus kommt es bei einer lokalen Testumgebung zu Laufzeitfehlern. Auch die im Webfrontend angezeigte Anzahl der importierten Datensätze ist nicht korrekt.

1.1.2 Phase 2: Korrektur der Darlehensverlaufsanzeige

(Duedate: 23.05.2013)
In der zweiten Phase soll das Hauptaugenmerk auf die Korrektur der Darlehensverlaufsanzeige gelegt werden. Diese Anzeige gilt als einer der Kerngeschäftsprozesse. Hier wird der Buchungsverlauf über einen, vom Benutzer festgelegten, Zeitraum dargestellt. Dabei ist es wichtig, die korrekten Punkte- und Währungs-Saldi zu berechnen. Dies funktioniert in der aktuellen Version nicht korrekt. Daher gilt es, die Fehler auszumerzen.

1.1.3 Phase 3: Einrichtung Testdriven Development, Erstellung der Model Tests

(Duedate: 13.06.2013)
Dies Phase steht im Zeichen der Einführung der Testgetriebenen Entwicklung („Testdriven Development“). In der Vergangenheit ist die Webanwendung stark gewachsenen, es sind immer häufiger Fehler und Inkonsistenzen aufegfallen. Ziel ist es nun der Anwendung, mit Hilfe von Tests, zu einem konsistenten und möglichst fehlerlosen Zustand zu verhelfen. Auch sollen die Tests helfen, sich einen genaueren Überblick über den Zustand der Webanwendung zu verschaffen. Für eine erfolgreiche und effiziente Test-Entwicklung, ist eine gute Auswahl der Tools unabdingbar. Der Gebrauch dieser Tools soll auch festgehalten werden.

1.1.4 Phase 4: Datenmodellkorrekturen, Implementierung Integrationtests

(Duedate: 11.09.2013) Nachdem sich ein detaillierter Überblick über den Zustand der Webanwendung verschafft worden ist, wird das Datenmodell festgelegt, implementiert und erneut

getestet. Ist dies abgeschlossen, werden beispielhafte Integrationstest implementiert und dokumentiert. Diese dienen dann den Nachfolgern als Anhaltspunkt für den weiteren Werdegang dieser Webanwendung. Außerdem werden grobe Fehler in der Webanwendung behoben.

2 Hintergrundwissen

2.1 Das o/ZB Projekt

Die Solidargemeinschaft o/ZB Stuttgart wurde am 21. Januar 2005 als GbR von 10 GesellschafterInnen gegründet und hatte Mitte des Jahres bereits 100 Mitglieder. Die o/ZB ist ein regionales Finanzierungsinstrument, das seinen Mitgliedern die Realisierung von Projekten in Selbsthilfe ermöglicht. Anderst als bei klassischen Einrichtungen erfolgen alle Einlagen (z.B. Sparen) und Entnahmen (z.B. Leihen) zinslos und kostenfrei, weil alle anfallenden Arbeiten von ihren Mitgliedern in Selbstverwaltung ausgeführt werden können.
Abbildung 2.1 zeigt das von der o/ZB verwendete Dreiphasenkonzept.

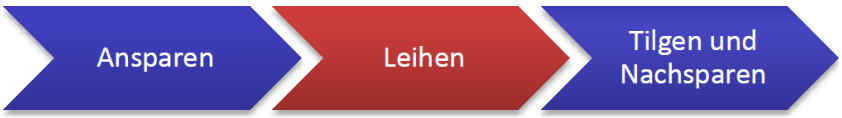


Abbildung 2.1: o/ZB Dreiphasenkonzept

2.2 Das Testsystem

Das Testsystem wird von Alvotech betrieben. Das Kundeninterface ist unter der URL `http://www.alvotech.de/kundenlogin` erreichbar. Das Testsytem bietet ein Debian GNU/Linux System, Kernelversion 3.1.2. Darüber hinaus ist ein MySQL Server in der Version 5.1.49-3 installiert. Auch Ruby ist in der Version 1.9.3p392 vorinstalliert.
Die Webanwendung liegt im Homeverzeichnis. Durch die Einführung von Capistrano geschieht eine Versionierung der Webanwendung. Ein Symlink *ozbapp* führt aus dem Homeverzeichnis zu der aktuellen Version der Webanwendung. Der Web- und Application Server, der für die Ausführung der Webanwendung zuständig ist, heißt *Phusion Passenger* und wird in der Version 3.0.19 verwendet (s. `https://www.phusionpassenger.com/`). Die Webanwendung ist unter der URL `http://ozbapp.mooo.com/` erreichbar. Der Code liegt in einem Git Repository auf

GitHub (`https://github.com/Avenel/FirstApp`). Details zu den einzeln genannten Softwaretechnologien können dem Kapitel 3.3.1 entnommen werden.

2.3 Test-Driven Development

2.3.1 Definition

Test-driven Development (eng. für „Testgetriebene Entwicklung“) ist eine Entwicklungsmethode die besonders in Unternehmen, die agile Softwareentwicklungsmethoden anwenden, zu Hause ist. Der Entwicklungsprozess ist recht klein gehalten und verfolgt einen festen definierten Ablauf. Der Ablauf sieht vor, dass zunächst der Test für eine gewünschte Funktionalität geschrieben wird und im Anschluss daran wird der Programmcode geschrieben, der dafür sorgen soll das der Test korrekt abgeschlossen wird. Die nachfolgende Figur 2.3.1 zeigt den gewünschten Ablauf noch einmal im Detail.

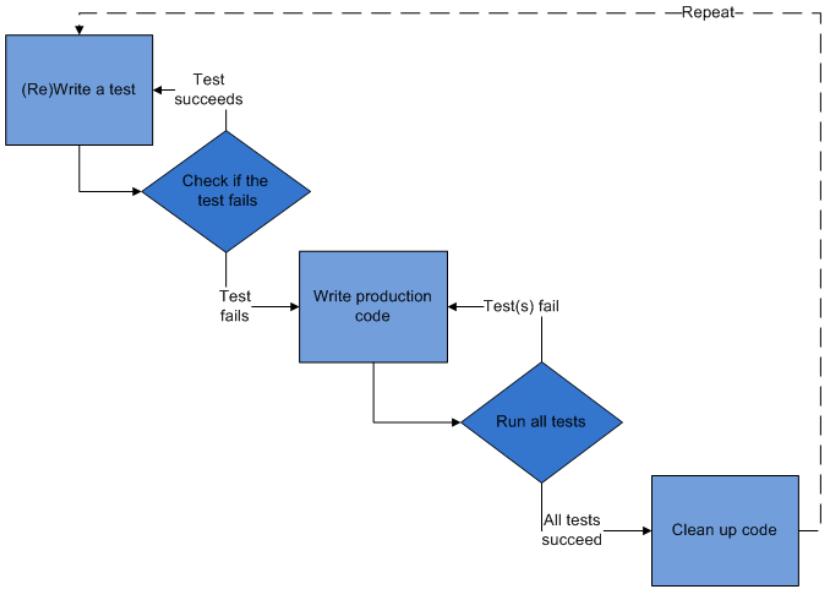


Abbildung 2.2: Testdriven Development Prozess

Diese Entwicklungsmethode stiftet die Programmierer dazu an, sehr schlanken (engl. „dry“) Programmcode zu schreiben und einen anderen Blick auf die Anwendung zu erhalten. Darüber hinaus steigert es während der Entwicklungsphase das Selbstbewusstsein des Programmieres, denn er kann mit Hilfe der Tests den Zustand der Anwendung schnell überblicken und sich sicher sein das sein Programmcode funktioniert - auch wenn der Umfang der Anwendung ansteigt. Außerdem steigert diese Entwicklungsmethode die Code-Qualität enorm und führt zu

einer Steigerung der Produktivität des Entwicklerteams.

Es gibt verschiedene Arten von Tests, die auf verschiedenen Ebenen agieren. In den nachfolgenden Abschnitten werden diese kurz dargestellt und in einen Zusammenhang gebracht. Dabei wird auf die Vorgehensweise bei einer Ruby on Rails Webanwendung Rücksicht genommen.

2.3.2 Modul Tests

Modul Tests, oder auch *Unit Tests*, sind Tests die die einzelnen Komponenten, bzw. Entitäten einer Anwendung testen. Das Ziel dieser Tests ist es, die sowohl fachliche als auch technische korrekte Funktionalität sicherzustellen. Die Tests stellen einen Vertrag dar, die das zu testende Modul erfüllen muss. Diese Testart ist ganz nach vorn im Testprozess einzuordnen und ist die Voraussetzung dafür um mit den Controller Tests fortfahren zu können. Ein Modul ist somit die kleinst zu testendende Einheit in einer Anwendung. Ein Modul Test soll sicherstellen, dass jedes kleine Teil eines Ganzen für sich allein gesehen korrekt funktioniert, sodass das eine Anwendung auf einer soliden Basis aufbauen kann. Bei der Implementierung eines Modul Tests ist zu beachten, dass das zu testende Modul isoliert getestet wird. D.h. es dürfen keine Modulübergreifende Interaktionen implementiert werden. Außerdem testen Modul Tests gemäß des „Design-by-Contract“Prinzips nicht die interne Umsetzung des Moduls, sondern ihre Ausgaben bei verschiedenen Eingaben.

2.3.3 Controller Tests

Der nächste Schritt besteht darin, die Funktionen eines Controllers zu testen. Ein „Controller Test“ist im Grunde nicht viel anders als ein Modul Test, allerdings werden hierbei auch die Interaktionen der verschiedenen Module untereinander getestet, falls der Controller dies leisten soll. Das Ziel ist die Sicherstellung, dass bei fest definierten Benutzereingaben die gewünschte Reaktion geschieht. Auch bei einem Controller Test werden alle Funktionen isoliert getestet. In der Testhierarchie besetzt der Controller Test die mittlere Position, zwischen den Modul Tests und den nun dargestellten Feature Tests.

2.3.4 Feature Tests

Ein sogenannter „Feature Test“ist in etwa vergleichbar mit einem „Integration Test“. Der Feature Test überprüft das Zusammenspiel der verschiedenen Controller. Darüber hinaus werden Testszenarien festgelegt, in denen reale Benutzerinteraktionen beschrieben werden um zu sehen wie die Webanwendung auf diese reagiert. Somit können die Interaktionen eines Benutzers mit

der Webanwendung vollständig nachvollzogen werden. Die Benutzerinteraktionen werden vom Testsystem emuliert und generieren Benutzereingaben, die von den Controllern verarbeitet werden. Hier zeigt sich der Unterschied zu den vorher vorgestellten Controller Tests, denn dort sind die Eingaben bereits festgelegt. Mit Hilfe der Feature Tests können somit weite Bereiche einer Webanwendung getestet werden und sind u.a. ein beliebtes Hilfsmittel um bei Bug-Reports ein Testszenario zu beschreiben.

3 Allgemeine Dokumentationen

3.1 Das korrigierte ER-Diagramm

[Insert content, Micha]

3.2 Das korrigierte Relationenmodell

[Insert content, Micha]

3.3 Deployment und Versionskontrolle

In diesem Abschnitt werden die Begriffe *Deployment* und *Versionskontrolle* erläutert. In jedem Teilabschnitt wird die aktuelle Umsetzung beschrieben. Der letzte Teil dieses Abschnitts beschäftigt sich mit dem *Commit & Deployment Workflow*.

3.3.1 Deployment

In der Softwareentwicklung wird unter dem Begriff *Deployment* der Prozess zur *Bereitstellung* und *Verteilung* einer Software verstanden. Der Prozess ist von Software zu Software unterschiedlich, da z.B. die Konfiguration einer Software immer an die Umgebung angepasst werden muss, in der diese zum Einsatz kommt.

Im Falle der o/zb muss der Deployment Prozess die Bereitstellung der o/zb Webanwendung auf einem Server bewerkstelligen. Dies wirft die folgenden Fragen auf: Welcher Webserver wird eingesetzt? Wie gelangt die RoR Webanwendung auf den Server? Und wie arbeiten Anwendungsserver und Webserver auf dem Server zusammen? Diese Fragen werden die nun folgenden Teilabschnitte (kurz) klären.

Webserver - Apache

Ein Webserver überträgt Daten an einen Client. Dabei kann dieser die Daten nur lokal oder auch weltweit zur Verfügung stellen. Er dient im Falle der o/zb Webanwendung dazu, die von RoR erzeugten Webseiten an den Clienten auszuliefern. Dies geschieht sobald er die Anfrage eines Clienten aufgenommen und an die RoR Anwendung weitergeleitet hat. Daraufhin verarbeitet die RoR Anwendung diese Anfrage und liefert dem Webserver das Ergebnis zurück. Der Webserver sendet nun diese Daten an den Clienten.

Auf dem Testsystem der o/zb (s. 2.2 auf Seite 4) kommt ein Apache Webserver in der Version 2.2.16 zum Einsatz. Auf dem Testsystem laufen bis zu vier Apache Instanzen gleichzeitig, um die Anfragen der Benutzer bedienen zu können. Weitere Informationen zum Apache Webserver sind unter der Adresse `http://www.apache.org` erreichbar.

Anwendungsserver - Phusion Passenger

Ein *Anwendungsserver* bietet einer Anwendung die benötigte Laufzeitumgebung, damit diese auch ausgeführt werden kann. Dazu stellt der Anwendungsserver der Anwendung spezielle Dienste zur Verfügung, die die Anwendung zur Ausführung benötigt.
Im Fall der o/zb Webanwendung wird der weit verbreitete und leistungsstarke *Phusion Passenger* Anwendungsserver in der Version 3.0.19 genutzt. Phusion Passenger ist ein Modul für den Apache Webserver und ist als ein sogenanntes *RubyGem* (entspricht etwa einem Softwarepaket speziell für Ruby) verfügbar. Zudem ist es auch unter dem Namen *mod_rails* oder *mod_rack* bekannt. Weitere Informationen können auf der Webseite von Phusion `https://www.phusionpassenger.com/` entnommen werden. [1]

Capistrano

Wie Eingangs dieses Abschnittes erwähnt worden ist, sind bei einem Deployment Prozess die Schritte notwendig, die die Bereitstellung der Software bewerkstelligen. Diese Schritte müssen nicht immer manuell ausgeführt werden, sondern können automatisiert werden. Dafür eignet sich im Falle der o/zb Webanwendung die Software *Capistrano*. Diese Software ist ein Open Source Werkzeug, das (Batch-) Skripte auf Servern, z.B. mit der Hilfe einer *Secure Shell (SSH)*, ausführt. Dem zur Folge ist ihr Haupteinsatzzweck in der Softwareverteilung wiederzufinden. Capistrano ist genauso wie die o/zb Webanwendung auch in Ruby geschrieben und als RubyGem verfügbar. [2]

[1] vgl. `http://de.wikipedia.org/wiki/Phusion_Passenger`
[2] vgl. `http://de.wikipedia.org/wiki/Capistrano_(Software)`

Bei Capistrano bestimmen sogenannte Deployment Rezepte („Deployment Recipies“) wie der Deployment Prozess verläuft. Auch für die o/zb Webanwendung wurde ein Rezept geschrieben, welches im Abschnitt 3.3.3 auf der nächsten Seite beschrieben wird.

3.3.2 Versionsverwaltung

Für einen stabilen und guten Softwareentwicklungsprozess ist eine Versionsverwaltung in der heutigen Zeit unabdingbar. Die Hauptaufgaben bestehen aus der Protokollierung der vorgenommenen Änderungen an Quelltexten, Skripten und anderen Dokumenten. Der Wiederherstellung von alten Zuständen, sodass versehentliche Änderungen oder Änderungen, die z.B. zu Laufzeitfehlern führten, zurückgenommen werden können. Die Archivierung jedes neuen Projektzustands. Die Koordinierung des gemeinsamen Datei-Zugriffs der am Projekt beteiligten Entwickler. Und zu guter Letzt ermöglicht eine Versionsverwaltung die gleichzeitige Erzeugung mehrerer Entwicklungszweige (sogenannter „Branches“) eines Projektes. [3]

Git

Für das Projekt der o/zb Stuttgart wird die weit verbreitete, freie Software *Git* verwendet. Es wurde ursprünglich für die Quelltext-Verwaltung des Linux Kernels entwickelt.
Git ist im Gegensatz zu den Traditionellen Versionsverwaltungen wie z.B. *SVN* oder *Mercurial* ein verteiltes Versionsverwaltungssystem. Es gibt keinen zentralen Server auf dem das Projekt gespeichert wird, sodass jeder Entwickler eine lokale Kopie des gesamten Repositorys vorliegen hat - *clone*. Dem zur Folge hat der Entwickler die Möglichkeit auch ohne Netzwerkzugriff die einzelnen Zustände seiner Arbeit festzuhalten - *commit*. Besteht wieder ein Netzwerkzugriff kann er seine Änderungen auf das von den Entwicklern gemeinsam genutztes Projekt-Depot (*Repository*) hochladen - *push*. Zuvor muss er sich jedoch mit dem gemeinsamen Repository synchronisieren - *pull*. [4]

Das aktuelle, gemeinsame Repository des o/zb Projektes wird von dem bekannten Git-Hoster *GitHub* bereitgestellt. Die Adresse zum Repository lautet: `https://github.com/Avenel/FirstApp`.

[3] vgl. `http://de.wikipedia.org/wiki/Versionsverwaltung`
[4] vgl. `http://de.wikipedia.org/wiki/Git`

3.3.3 Workflow, Umsetzung

In diesem Teilabschnitt werden die vorher erläuterten Konzepte *Deployment* und *Versionsverwaltung* in Zusammenhang gebracht. Es wird ein Arbeitsablauf („Workflow“) defininert, der den Deployment Prozess beschreibt. Dieser Arbeitsablauf wird in Abbildung 3.1 dargestellt.

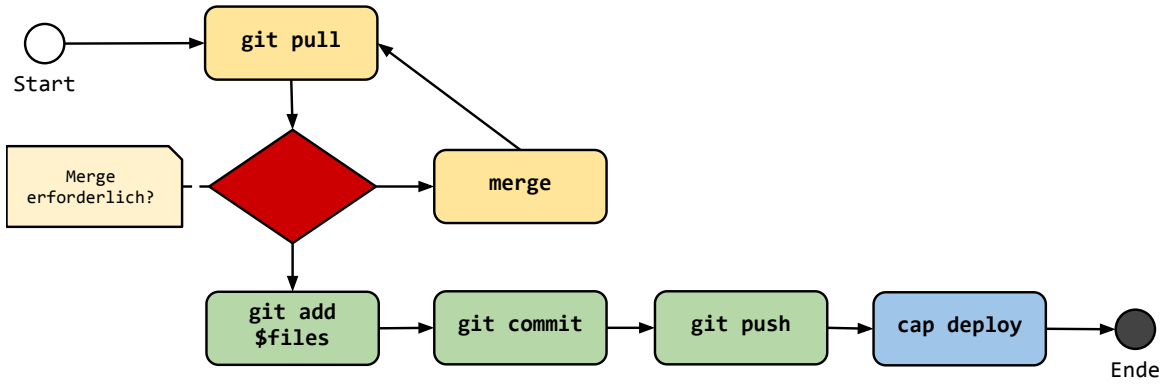


Abbildung 3.1: Der Deployment Workflow

Möchte der Entwickler seine Arbeit auf dem Server bereitstellen, ist er angehalten sich den aktuellen Projektstatus aus dem gemeinsamen Repository zu holen. Er ist dafür verantwortlich sein Projekt vor jedem Commit auf den neuesten Stand zu bringen. Dies geschieht mit der Anweisung *git pull*. Kommt es zu (Datei-) Konflikten die Git nicht automatisch selber lösen kann, muss der Entwickler selber eingreifen (*merge*). Er wiederholt diesen Vorgang solange, bis sein Projekt konfliktfrei und auf dem neuesten Stand ist. Erst dann kann er ausgewählte Änderungen am Projekt für einen neuen *Commit* hinzufügen (*git add*). Im Anschluss schließt er den Commit-Prozess mit dem Befehl *git commit -am "Kommentar"* ab. Damit auch das gemeinsame Repository auf den aktuellsten Stand gebracht wird, erfolgt der Befehl *git push*. Dieser lädt die neuesten Änderungen hoch.

Wurde das gemeinsame Repository nun auf den neuesten Stand gebracht, kann der letzte Schritt im Deployment Workflow durchgeführt werden. Mit *cap deploy* wird das, im nächsten Abschnitt beschriebene, Deploymentskript ausgeführt.

Das Capistrano Deploymentskript (Rezept)

Das Capistrano Deploymentskript bzw. Rezept ist vorerst nicht im öffentlich zugänglichen Git Repository zu finden, da es durchaus sensible Informationen enthält. Ist es vorhanden befindet es sich hier: *config/deploy.rb*.

In dem Deploymentskript werden zuerst sämtliche Variablen festgelegt, der Name der Anwendung und die für einen sicheren SSH Zugriff notwendigen Daten (Serveradresse, sowie auch der Deployment-Benutzer: „ozbapp“). Darüber hinaus werden Informationen zum Repository angegeben, in dem die Projektdateien liegen. Im Anschluss wird der Deployment Ort auf dem Server, sowie eigene Aktionen während des Deployment Vorgangs festgelegt.

```
1  # Name application
2  set :application, "ozbapp"
3
4  # Setup deployment user and server ip
5  server "188.64.45.50", :web, :app, :db, :primary => true
6  set :user, "ozbapp"
7  set :use_sudo, false
8  ssh_options[:forward_agent] = true
9
10 # Setup git repository information
11 set :scm, "git"
12 set :repository, "https://github.com/Avenel/FirstApp.git"
13 set :branch, "master"
14
15 # Setup where to deploy the app on the server
16 set :deploy_to, "/home/#{user}/apps/#{application}"
17 set :deploy_via, :remote_cache
18
19 namespace :deploy do
20
21     desc "Tell Passenger to restart the app."
22     task :restart do
23         run touch "#{current_path}/ozbapp/tmp/restart.txt"
24     end
25
26     desc "Renew SymLink"
27     task :renew_symlink do
28         run "rm /home/ozbapp/ozbapp"
29         run "ln -s /home/ozbapp/apps/ozbapp/current/ozbapp /home/ozbapp/ozbapp"
30     end
31
32 end
33
34 # Execute renew_symlink after update_code
35 after 'deploy:update_code', 'deploy:renew_symlink'
```

Listing 3.1: Capistrano Deployment Rezept

3.4 Datenbank Migration

In der vergangenen Zeit wurde ein Java Datenbank Migrationstool geschrieben, welches die Daten von dem aktuellen Produktivsystem in das neue System übertragen soll. Um den Umgang mit diesem Tool zu erleichtern sind zwei Batch Skripte angefertigt worden. Diese Batch Skript sind im Ordner *tools* des Git Repositorys und auch auf dem Server im *Home Verzeichnis ozbapp* zu finden. Um die Batch Skripte ausführen zu können ist eine SSH Verbindung erforderlich. Diese kann unter Windows mit dem Programm *Putty* (`http://www.putty.org/`) oder unter Linux mit dem Befehl *ssh username@host* (`http://wiki.ubuntuusers.de/SSH`) geöffnet werden. Der Username bzw. der Hostname ist in beiden Fällen *ozbapp* bzw. *ozbapp.mooo.com*. Die benötigte Konfiguration für Putty ist der Abbildung 3.2 zu entnehmen. Ist man verbunden befindet man sich automatisch schon im *Home Verzeichnis* in dem die Batch Skripte vorliegen.

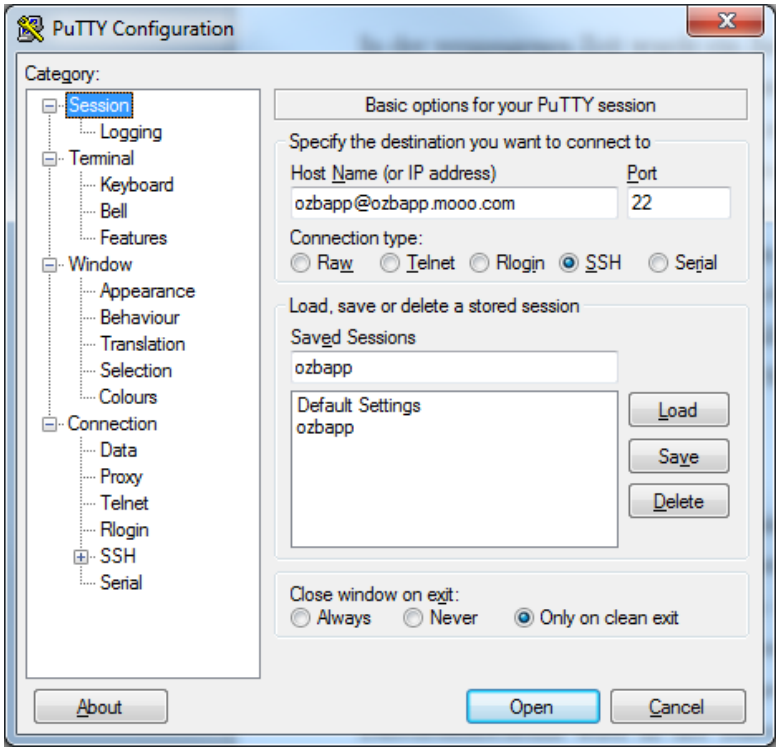


Abbildung 3.2: Putty Konfiguration

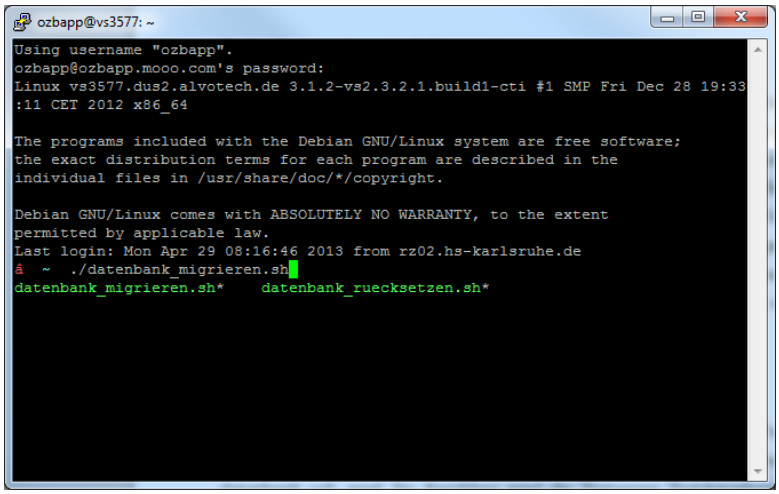


Abbildung 3.3: Offene SSH Session in Putty

Im Folgenden werden die Funktionsweise und die Benutzung dieser beiden Batch Skripte beschrieben.

3.4.1 Datenbank migrieren

Dieses Batch Skript importiert zunächst einen aktuellen Stand der Produktivdatenbank, der als MySql Dump zur Verfügung gestellt wird, in die auf dem Testserver liegende Produktivdatenbank *ozb_prod*. Im Anschluss wird die Testserver Testdatenbank *ozb_test* geleert und neu angelegt. Ist dies geschehen, werden mit Hilfe des Datenbank Migrationstools die Daten aus der Produktivdatenbank *ozb_prod* in die Testdatenbank *ozb_test* übertragen. Das neue Datenbankschema wird in der Datei *create_tables.txt* beschrieben. Sind Veränderungen am Datenbankschema vorgenommen worden, müssen diese in dieser Datei übernommen werden. Ausgeführt wird dieses Skript mit dem Befehl *./datenbank_migrieren.sh*.

```
1  #!/bin/bash
2  PASS="root"
3  echo "drop database ozb_prod" | mysql -u root -p$PASS
4  echo "create database ozb_prod" | mysql -u root -p$PASS
5  mysql -u root -p$PASS ozb_prod < ozb_prod.sql
6  echo "drop database ozb_test" | mysql -u root -p$PASS
7  echo "create database ozb_test" | mysql -u root -p$PASS
8  java -jar OZBMigration.jar -i ./create_tables.txt -u root -p $PASS
9
10 echo "bootstrap (adding default values for developing purposes)"
11 cd ../ozbapp/
```

```
12 bundle exec rake db:seed
13
14 mysqldump -u root -p$PASS ozb_test > dump.sql
```

Listing 3.2: datenbank_migrieren.sh

3.4.2 Datenbank zurücksetzen

In diesem Batch Skript wird die Testdatenbank auf den ursprünglichen Zustand zurückgesetzt. Der ursprüngliche Zustand liegt in dem MySql Dump *dump.sql*. Zur Ausführung genügt auch hier der folgende Befehl *./datenbank_ruecksetzen.sh*.

```
1 #!/bin/bash
2 PASS="root"
3 echo "drop database ozb_test" | mysql -u root -p$PASS
4 echo "create database ozb_test" | mysql -u root -p$PASS
5 mysql -u root -p$PASS ozb_test < dump.sql
```

Listing 3.3: datenbank_migrieren.sh

4 Korrekturen, Bugfixes

4.1 Webimport

In der o/zb Webanwendung sorgt der sogenannte *Webimport* für die Übertragung der in einer CSV Datei hinterlegten Kontobewegungen. Weitere Informationen zu dem Webimport kann der Projekt Dokuementation WS 12/13 (ab S.53) entnommen werden.

Zum Zeitpunkt der Aufnahme der Arbeiten an diesem Projekt funktionierte der Webimport nicht korrekt. Die folgenden Punkte sind aufgefallen:

- Datei Upload auf dem Testsystem funktioniert nicht.
- Laufzeitfehler müssen abgefangen werden.
- Die Anzahl der importierten Datensätze stimmt nicht.

4.1.1 Bugfix: Datei Upload

Bei diesem Fehler akzeptierte der Webimport scheinbar keine Datei. Der Grund hierfür liegt in den abgebildeten Zeilen in Listing 4.1.3 auf Seite 18.

```
1     if (collect_konten.size == 0 )
2       @error += "Keine der zu importierenden Konten in der Datenbank eingetragen"
3     else
4       collect_konten.uniq.each do |ktoNr|
5         b = Buchung.find(
```

Listing 4.1: webimport_controller.rb

In Zeile 4 (bzw. Zeile 426) wurde ursprünglich der Befehl *uniq!* verwendet. Methoden mit einem Ausrufezeichen (!) benutzen meistens eine unsichere Implementierungsweise der Methode. Dieser Befehl gibt laut der offiziellen RubyDoc [1] entweder ein *Array* oder *nil* zurück. Der

[1] vgl. http://www.ruby-doc.org/core-1.9.3/Array.html#method-i-uniq-21

Rückgabewert *nil* tritt ein, sobald das Array keine Elemente vorweisen kann. Da dieser Fall nicht abgefangen worden ist, kommt es in der Zeile 4 (bzw. 426) zu einem Fehler. Abhilfe schafft hier die Methode *uniq*. Diese gibt in jedem Fall ein (ggf. leeres) *Array* zurück.

4.1.2 Laufzeitfehler

Konnte ein Datensatz z.B. aufgrund eines MySql Fehlers nicht importiert werden, wird eine Exception geworfen. Aufgrund dieser Exception wurde der Importvorgang nicht korrekt zu Ende geführt, sondern direkt abgebrochen. Der Benutzer sieht in diesem Moment eine Fehlerseite und kann nicht mehr agieren, da er keine Informationen darüber erhält, was schiefgegangen ist. Der häufigste Fehler tritt auf, wenn versucht worden ist eine Buchung noch ein weiteres Mal zu importieren (MySql Fehler: *duplicate key error*). Nun gilt es mit Hilfe des Exception Handlings das Verhalten der Webanwendung zugunsten des Benutzers zu beeinflussen.

Die Exceptions treten beim Speichervorgang auf, dem zur Folge wurde um jeden Speichervorgang ein *begin-rescue* Block gesetzt. Dieser fängt die Exception auf und hängt die Fehlermeldung der *@error* Variable an. Diese stellt dem Benutzer ggf. die Informationen darüber, was schiefgelaufen ist, dar.

```
1                begin
2                  b.save
3                  collect_konten << kontonummer
4                  imported_records += 1
5                rescue Exception => e
6                  @error += "Etwas ist schiefgelaufen.<br /><br />"
7                  @error += e.message + "<br /><br />"
8                end
```

Listing 4.2: webimport_controller.rb

4.1.3 Anzahl der importierten Datensätze

Eine weitere Auffälligkeit bestand in der angezeigten Anzahl der importierten Datensätze. Diese Zahl zeigte ggf. zu viele importierten Datensätze an. Unter einem Datensatz versteht man eine Kontobewegung in der CSV Datei.

Einer der Fehler tritt in Zeile 318 auf. Dort werden bei einer bestimmten Kontobewegung zwei Buchungen durchgeführt. Beide Buchungen wurden mitgezählt, obwohl der Auslöser nur *ein*

Datensatz gewesen ist. Durch Auskommentieren und einem Hinweiß konnte dieser Fehler korrigiert werden.

```
1                 begin
2                   b.save
3                   collect_konten << kontonummer
4                   # Nur 1x zaehlen!
5                   #imported_records += 1
6                 rescue Exception => e
7                   @error += "Etwas ist schiefgelaufen.<br /><br />"
8                   @error += e.message + "<br /><br />"
9                 end
```

Listing 4.3: webimport_controller.rb

4.1.4 Punkteberechnung

Beim Import der Buchungen werden zum Zeitpunkt der Übernahme noch keine Punkte berechnet. Im Zuge der Modul Tests wurden die mit dem Webimport verbundenen Module nochmals überarbeitet. Auf Grund des immensen Arbeitsaufwands, der in das Refactoring gesteckt werden musste, wurde eine erneute Implementierung der Punkteberechnung im Rahmen dieser Projektarbeit nicht in Betracht gezogen.

4.2 Darlehensverlauf

Der Darlehensverlauf ist zum Zeitpunkt der Übernahme fehlerhaft gewesen. Im Zuge in den von Herrn Kienle angefertigten Tests, konnten die auftretenen Fehler analysiert und kategorisiert werden. Aufgrund des relativ unstrukturierten und unübersichtlichen Programmcodes, wurde beschlossen diese Funktionalität, basierend den Spezifikationen die sich aus den Tests und Dokumentationen ableiten ließen, neu zu implementieren.

Fehlerursachen:

- Punkte wurden nicht richtig berechnet, wenn am Tage des abDatums noch Buchungen getätigt wurden
- Die Sortierung der Buchungen stimmte nicht mit den Vorgaben überein
- Wiederverwendete ZE Konten wurden nicht richtig berücksichtigt

- Startsaldo wurde nicht korrekt berechnet, da die Tagesdifferenz zur letzten Währungsbuchung nicht berücksichtigt wurden
- Die Punkte die bei der Buchung nach dem Startsaldo berechnet wurden, sind falsch da diese einfach aus der DB entnommen wurden
- Variable KKL-Verläufe, KKL veränderte sich während des abgefragten Zeitraumes

Diese Ursachen wurden bei der kompletten Neu-Implementierung beachtet und umgesetzt. Integrationstest, beschrieben in Kapitel **??** sichern die korrekte Funktionalität in den gegebenen Awendungsfällen.

4.3 HistoricRecord

Im Zuge des Test-Driven Development und dem damit verbundenen Refactoring, ist aufgefallen das für jedes historisierte Model im Grunde derselbe Quellcode geschrieben wurde, der für die Historisierung zuständig gewesen ist. Dieser lässt sich nur sehr schlecht warten und im allgemeinen ist großer, redundanter Quellcode, wie hier vorgefunden wurde, schlecht. Daher wurde ein neues Ruby-Modul implementiert: *HistoricRecord*. Dieses Ruby-Modul lässt sich für jedes Model anwenden, wenn dies die folgenden Methoden implementiert und Kriterien beachtet.

get_primary_keys
Diese Methode liefert den primären Schlüssel, bis auf das Attribut GueltigVon (dies ist in jedem primären Schlüssel einer historisierten Klasse enthalten). Abbildung 4.3 zeigt eine beispielhafte Implementierung.

```
1  def get_primary_keys
2    return {"KtoNr" => self.KtoNr}
3  end
```

Listing 4.4: Factory der Adressen

set_primary_keys
Die Methode *set_primary_keys(values)* erwartet die Werte für den primären (zusammengesetzten) Schlüssel. Nach Aufruf entsprichen die Werte der primärschlüssel Attribute den angegebenen Werten. Abbildung 4.3 auf der nächsten Seite zeigt eine beispielhafte Implementierung.

```
1  def set_primary_keys(values)
2    self.KtoNr = values["KtoNr"]
3  end
```

Listing 4.5: Factory der Adressen

getLatest

Die Methode *getLatest* liefert die aktuellste Version der Model Instanz zurück, bevor die Änderungen abgespeichert werden. Sie ist notwendig, um die GueltigBis Zeit für diese Instanz neu zu setzen. Abbildung 4.3 zeigt eine beispielhafte Implementierung.

```
1  def getLatest()
2    return OzbKonto.get(self.KtoNr)
3  end
```

Listing 4.6: Factory der Adressen

Attribute: GueltigVon, GueltigBis

Diese Attribute müssen auf jeden Fall vorhanden sein, sowohl als *accessible attribute* als auch in der Datenbank.

Sind diese Voraussetzungen erfüllt, muss das Modul noch eingebunden werden. Da Ruby keinen *Polymorphismus* (Mehrfachvererbung) unterstüzt, wird *Duck typing* (s. `http://en.wikipedia.org/wiki/Duck_typing`) angewandt. Abbildung 4.3 zeigt, wie es geht.

```
1 require "HistoricRecord.rb"
2
3 class OzbKonto < ActiveRecord::Base
4    include HistoricRecord
```

Listing 4.7: Factory der Adressen

4.4 Umzug der Benutzerattribute in die Tabelle User

[TODO]

5 Testdriven Development

In diesem Kapitel wird die Einführung und Durchführung der Eigenschaften der Testgetriebenen Entwicklung beschrieben. Im Zuge dessen wird zunächst eine Analyse des Ist und des Soll Standes durchgeführt, bevor mit der Umsetzung und Durchführung fortgefahren wird. Das Ziel ist die Einführung und Durchführung von Tests, um den Zustand der Webanwendung darzulegen und um die aufgezeigten Mängel zu beseitigen und mit Hilfe der Tests nicht wieder eintreten zu lassen.

5.1 Analyse

5.1.1 Ist-Stand

Der Ist-Zustand der vorliegenden Webanwendung ist zum Zeitpunkt der Übernahme nur schwer überschaubar. Es gibt viele neue Funktionalitäten und Testberichte die deutliche Mängel aufzeigen. Die vorliegenden Dokumentationen beschreiben die neuen Funktionalitäten und deren Umsetzung, allerdings fehlt ein Überblick über den Gesamtzustand der Webanwendung. Es wurden nur einzelne, wichtige, Funktionalitäten getestet. Allerdings handelt es sich hierbei um Frontend Tests, die jedoch keine ausreichenden Aussagen über die fachliche und technische Korrektheit der Module und deren Beziehungen untereinander liefern können.

Um gültige Aussagen über den Zustand dieser Webanwendung treffen zu können, sind umfangreiche Modultests der erste Schritt. Diese liefern den Zustand über die einzelnen Module, bzw. Entitäten wie z.B. Person, OZBPerson, OZBKonto usw. zurück. Auch liefern diese Aussagen über die Korrektheit der Umsetzung des vorgegeben Datenmodells. Hier sind auf dem ersten Blick bereits einige Unstimmigkeiten aufgefallen.

5.1.2 Soll-Stand

Ein erster Meilenstein stellt die korrekte Umsetzung des fachlichen Datenmodells dar. Dazu gehört, dass jedes Model den fachlichen Beschreibungen entspricht und sauber implementiert wurde. Im Idealfall ist die Struktur aller Module gleich, der Code auf ein Minimum reduziert und konsistent hinsichtlich z.B. der Attributsnamen und der Art und Weise wie bestimmte Funktionalitäten (wie z.B. die Suche nach einer Modulinstanz) implementiert wurden. Die Modul Tests gewährleisten hierbei die fachlich korrekte Umsetzung. Auf die technisch konsistente Umsetzung der Module muss der Entwickler selbst achten.
Der nächste Meilenstein stellt eine einheitliche, technisch und fachlich korrekte Implementierung der Controller und den dazugehörigen Views dar. Hierbei stellt die korrekte Umsetzung der Module das Standbein und somit auch die notwendige Weiche für diesen Meilenstein. Hierbei gewähren die Controller Tests eine fachlich korrekte Umsetzung, ehe sich die Feature Tests auch von der korrekten Umsetzung in einem konkreten Anwendungsfall, der mehrere Controller involviert, überzeugen kann.

5.2 Implementierung

In diesem Abschnitt wird die Umsetzung der Tests beschrieben. Dazu gehört eine kurze Zusammenfassung aller benutzten Gems und wie sie eingesetzt wurden. Im Anschluss werden nacheinander Modul und Controller Tests vorgestellt. Dabei werden Aufbau und Funktionsweise anhand von Beispielen erläutert. Darüber hinaus gibt es eine Auflistung in der alle getesteten Modulen verzeichnet sind.

5.2.1 Verwendete Gems

RSpec

RSpec (s. `http://rubygems.org/gems/rspec`) ist das wohl bekannteste Testing-Framework für Ruby on Rails Applikationen. Es ist im Sinne des *Behaviour-Driven Development* entstanden und soll das Test-Driven Development unterstützen und leichter zugänglich machen. Das Verzeichnis */spec* birgt alle Dateien, die mit dem Testing in Verbindung stehen.

FactoryGirl

Factory_girl (s. `http://rubygems.org/gems/factory_girl`) stellt ein alternatives Framework zu den Ruby Fixtures zur Verfügung, das es erlaubt Daten mit Hilfe einer „Factory“(=Fabrik, s. `http://de.wikipedia.org/wiki/Abstract_Factory`) zu generieren. In diesr Webapplikation

lassen sich somit Beispieldaten für die Tests leichter erzeugen. Abbildung 5.2.1 zeigt einen Ausschnitt einer typischen Factory.

```
1  FactoryGirl.define do
2
3    factory :Adresse do
4      sequence(:Pnr) { |n| "#{n}"}
5      sequence(:SachPnr) { |n| "#{n}"}
6
7      Strasse Faker::Address.street_name
8      Nr Faker::Address.building_number
9      PLZ Faker::Address.zip_code
10     Ort Faker::Address.city
```

Listing 5.1: Factory der Adressen

Faker

Faker (s. `https://rubygems.org/gems/faker`) ist ein Tool für die Erzeugung von Fake Adressen, Namen, Telefonnummern und mehr. Dieses Gem ergänzt sich prima mit den oben genannten Factories. Eine detaillierte Beschreibung aller Möglichkeiten können unter der folgenden URL entnommen werden: `http://rubydoc.info/github/stympy/faker/master/frames`.

5.2.2 Gems die zukünftig verwendet werden könnten

In diesem Abschnitt werden Gems genannt, die in Zukunft den Test-Driven Development unterstützen könnten.

Capybara

Capybara (s. `http://rubygems.org/gems/capybara`) hilft, die Benutzerinteraktionen mit der Web-Applikation zu simulieren. Somit lassen sich Integrations Tests implementieren, die noch enger mit der Weboberfläche verzahnt sind.

Guard-Spec

Guard-spec (s. `http://rubygems.org/gems/guard-rspec`) dient dem Entwickler dazu, Tests automatisch (z.B. nach jedem Speichervorgang) auszuführen.

5.2.3 Entwicklertools

In die Auswahl der Entwicklertools wurde viel Zeit investiert, die im Nachhinein durch gute Auswahl wieder eingespart wurde. Um den nachfolgenden Studenten diesen Aufwand zu ersparen, werden in diesem Abschnitt die verwendeten Entwicklertools kurz aufgelistet und erläutert.

Sublime Text 2

Sublime Text 2 (s. `http://www.sublimetext.com/`) hat sich als ein sehr guter Texteditor (ähnlich VI und Textmate) herausgestellt. Er ist einfach zu bedienen, kann aber genauso viel wie seine Vorbilder. Durch Plugins kann der Funktionsumfang laufend erweitert werden. Diese fügen sich weniger aufdringlich in die Benutzeroberfläche ein (wie z.B. bei Notepad++). Weitere Links zu hilfreichen Webseiten können dem Wiki des Repositories entnommen werden (s. `https://github.com/Avenel/FirstApp/wiki/Sublime-Text-Setup`). Welche Plugins für das Test-Driven Development verwendet wurden, kann dem nun folgenden Abschnitt entnommen werden.

RubyTest

RubyTest (s. `https://github.com/mhartl/rails_tutorial_sublime_text`) ist ein Sublime Text 2 Plugin, dass den Entwickler bei der Ausführung der Tests hilft. Wie die Installation von statten geht und wie das Plugin benutzt wird, kann diesem Youtube Video entnommen werden `http://www.youtube.com/watch?v=05x1Jk4rT1A`. Man hat die Möglichkeit entweder alle Tests eines geöffneten Dokuments mit der Tastenkombination *STRG+SHIFT+T*, oder nur einen einzelnen Test (in dem sich der Cursor befindet) mit der Tastenkombination *STRG+SHIFT+R* auszuführen.

Cygwin

Falls man unter Windows entwickelt ist *Cygwin* eine sehr gute Alternative zu der herkömmlichen Windows Kommandozeile. Die Standardinstallation von Cygwin, zu finden hier: `http://www.cygwin.com/`, reicht in der Regel aus. Der große Vorteil hier ist, dass das Fenster frei skalierbar ist und sich der Text (z.B. die Ausgabe des Serverlogs) leichter lesen und markieren lässt.

5.2.4 Modul Tests

Ein Modul Test besteht aus drei Teilen:

- Testing der Factory

- Testing der Attribut-Validierungen
- Testing der Klassen/Instanz Methoden

Factory Test

Wie bereits in einem früheren Abschnitt gezeigt, werden Factories verwendet um Testdaten zu generieren. Damit sichergestellt ist, dass diese auch korrekt funktionieren besteht der erste Test darin, genau dies sicherzustellen. Abbildung 5.2.4 stellt beispielsweise einen solchen Test dar.

```
1  it "has a valid factory" do
2    expect(FactoryGirl.create(:Bank)).to be_valid
3  end
```

Listing 5.2: Factory der Adressen

Attribut-Validierungen

Attribute müssen in den meisten Fällen bestimmten Kriterien entsprechen. Sie dürfen mal nur eine Nummer sein, mal einem bestimmten regulären Ausdruck entsrechen, müssen vorhanden sein oder auch nicht. Wichtig ist, dass man die folgenden Fälle, die Abbildung 5.2.4 zeigt abdeckt.

```
1  # BIC
2  it "is valid with a valid BIC" do
3    # Valid code = 8 or 11 chars long
4    expect(FactoryGirl.create(:Bank, :BIC => "AS13AS12")).to be_valid
5    expect(FactoryGirl.create(:Bank, :BIC => "AS13AS1223A")).to be_valid
6  end
7
8  it "is invalid without a BIC" do
9    expect(FactoryGirl.build(:Bank, :BIC => nil)).to be_invalid
10 end
11
12 it "is invalid with an invalid BIC" do
13   # invalid = length != 8|11
14   expect(FactoryGirl.build(:Bank, :BIC => "1234")).to be_invalid
15   expect(FactoryGirl.build(:Bank, :BIC => "123456789")).to be_invalid
16   expect(FactoryGirl.build(:Bank, :BIC => "123456789ABCDEF")).to be_invalid
17 end
```

Listing 5.3: Factory der Adressen

Klassen/Instanz Methoden

Jede Klasse besitzt eigene (private) Methoden die auch getestet werden müssen. Das Prozedere ist auch hier dasselbe, man gibt eine Instanz des zu testenden Models vor, führt die Methode aus und erwartet ein Ergebnis.

```
1  # Class and instance methods
2  # Fullname
3  it "returns the fullname of a person" do
4    person = FactoryGirl.create(:Person, :Name => "Mustermann", :Vorname => "Max")
5    expect(person.fullname).to eq "Mustermann, Max"
6  end
```

Listing 5.4: Factory der Adressen

Falls die Methode eine private Methode ist, muss die Methode auf den folgenden Wege aufgerufen werden, wie es Abbildung 5.2.4 zeigt.

```
1    # Private method, therefore using send methode
2    ozbKonto.send(:set_saldo_datum)
```

Listing 5.5: Factory der Adressen

HistoricRecord

Auch dieses Modul wurde gesondert getestet. Abbildung 5.2.4 auf der nächsten Seite zeigt stellvertretend die Art und Weise, wie jede einzelne Methode getestet wurde. Dabei ist es wichtig, dass zuerst jede einzelne Funktion für sich isoliert getestet wurde, bevor jedes einzelne Model nochmals eigenständig auf die korrekte Funktionsweise hin geprüft wurde.

```
1    # OZBKonto
2    it "does historize OZBKonto" do
3      # create origin record
4      oldTime = Time.now
5      ozbKontoOrigin = FactoryGirl.create(:ozbkonto_with_ozbperson)
6      expect(ozbKontoOrigin).to be_valid
7
8      # Asure that only one record exists
9      query = OzbKonto.find(:all, :conditions => ["KtoNr = ? AND Mnr = ?",
10                       ozbKontoOrigin.KtoNr, ozbKontoOrigin.Mnr])
11     expect(query.count).to eq 1
12
13     # Asure GueltigVon and GueltigBis are correct
14     expect(ozbKontoOrigin.GueltigVon.getlocal().strftime("%Y-%m-%d %H:%M:%S")).to
```

```
15                          eq oldTime.getlocal().strftime("%Y-%m-%d %H:%M:%S")
16      expect(ozbKontoOrigin.GueltigBis.getlocal().strftime("%Y-%m-%d %H:%M:%S")).to
17        eq Time.zone.parse("9999-12-31 23:59:59").getlocal().strftime("%Y-%m-%d %H:%M:%S")
18
19      # Change any value
20      ozbKontoOrigin.WSaldo = 42
21
22      # wait a second
23      sleep(1)
24
25      # Save
26      saveTime = Time.now
27      expect(ozbKontoOrigin.save).to eq true
28
29      # Query again, there should be 2 records by now
30      query = OzbKonto.find(:all, :conditions => ["KtoNr = ? AND Mnr = ?",
31                          ozbKontoOrigin.KtoNr, ozbKontoOrigin.Mnr])
32      expect(query.count).to eq 2
33
34      # Check GueltigVon and GueltigBis of both records
35      ozbKontoOrigin = OzbKonto.find(:all,
36                      :conditions => ["KtoNr = ? AND Mnr = ? AND GueltigBis = ?",
37                      ozbKontoOrigin.KtoNr, ozbKontoOrigin.Mnr, saveTime]).first
38      expect(ozbKontoOrigin.GueltigVon.getlocal().strftime("%Y-%m-%d %H:%M:%S")).to
39                      eq oldTime.getlocal().strftime("%Y-%m-%d %H:%M:%S")
40      expect(ozbKontoOrigin.GueltigBis.getlocal().strftime("%Y-%m-%d %H:%M:%S")).to
41                      eq saveTime.getlocal().strftime("%Y-%m-%d %H:%M:%S")
42
43      ozbKontoLatest = OzbKonto.find(:all,
44                      :conditions => ["KtoNr = ? AND Mnr = ? AND GueltigBis = ?",
45                      ozbKontoOrigin.KtoNr, ozbKontoOrigin.Mnr, "9999-12-31 23:59:59"]).first
46      expect(ozbKontoLatest.GueltigVon.getlocal().strftime("%Y-%m-%d %H:%M:%S")).to
47        eq saveTime.getlocal().strftime("%Y-%m-%d %H:%M:%S")
48      expect(ozbKontoLatest.GueltigBis.getlocal().strftime("%Y-%m-%d %H:%M:%S")).to
49        eq Time.zone.parse("9999-12-31 23:59:59").getlocal().strftime("%Y-%m-%d %H:%M:%S")
50    end
```

Listing 5.6: Factory der Adressen

Gesteste Models

Die folgenden Models wurden getestet.

- Bank
- Bankverbindung
- Buergschaft (Skelett)
- EEKonto
- OZBKonto
- OZBPerson
- Person
- Projektgruppe
- User
- Waehrung
- ZEKonto

5.2.5 Controller Tests

Anwendungsfälle der einzelnen Tests

5.3 Auswertung

Die Auswertung befindet sich im Anhang Kapitel 8 auf Seite 32.

6 Neue Features

6.1 PaperTrail - Historisierung

6.2 Deployment E-Mail Benachrichtigung

s. Capify Skript: „cap_notify.rb“

7 Ergebnis und Ausblick

7.1 Ergebnis

7.2 Ausblick

8 Anhang

O/ZB Webanwendung

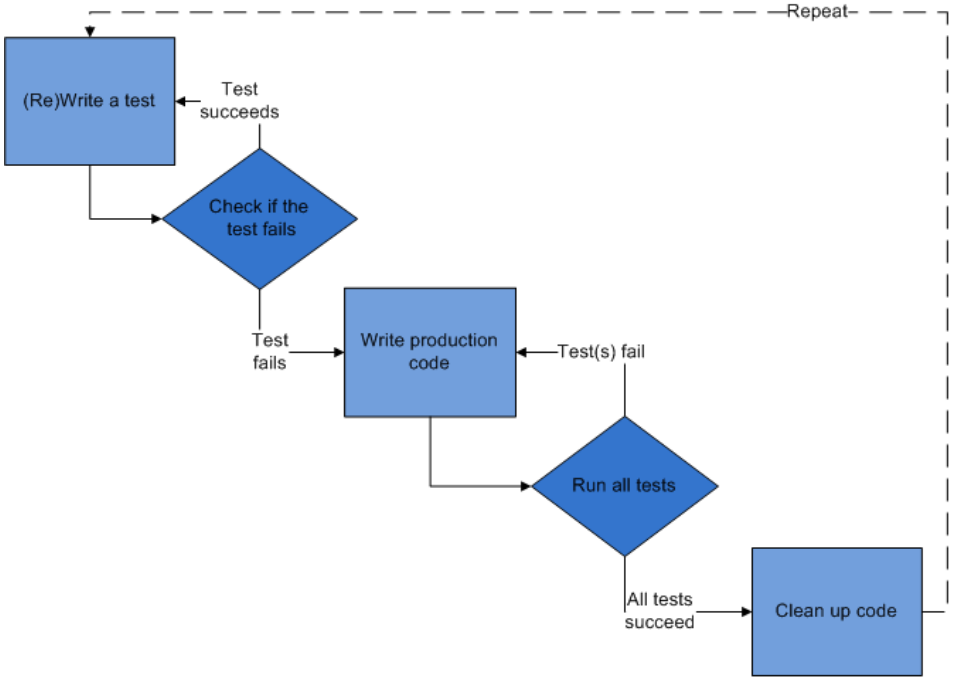
TDD - Zwischenstand

Übersicht

- Was ist Testdriven Development?
- Unit Test
- Controller Test
- Datenbankmodell, Umsetzung in MySQL
- Offene Fragen, Issues

Testdriven Development

- Hauptmerkmal
  - Software-Tests werden konsequent *vor* den zu testenden Komponenten erstellt



Testdriven Development

Ziele und Umsetzung

- Das Hauptmerkmal kann in diesem Projekt erst wieder bei neuen Funktionalitäten angewandt werden
- Mit Hilfe der Tests soll der bisherige Zustand, der historisch gewachsenen Webanwendung, aufgenommen werden
  - Konsistenz: Ruby Anwendung <-> Datenbank
  - Fachlich und technische Validierung der einzelnen Modelle (Attribute, Beziehungen und Funktionen) und Controller (Logik, Controller übergreifende Funktionalitäten)

Unit Test

Ziele

- Es soll sichergestellt werden, dass nur fachlich und technisch korrekte Objekte des Models erzeugt werden können.
- Sicherstellung der korrekten Funktionalität der im Model enthaltenen Funktionen

Unit Test

Voraussetzung

- Fachlich und technisch korrekter Testdatensatz
- Vollständige Implementierung des Models:
  - Vollständigkeit der fachlichen und technischen Attribute, die mit der Datenbank übereinstimmen
  - Validierungen der vorhandenen Attribute (z.B. Beziehungen, Datenformate, Wertebereiche, usw.)
  - Historierungsfunktionalität bereits implementiert, wo diese notwendig ist

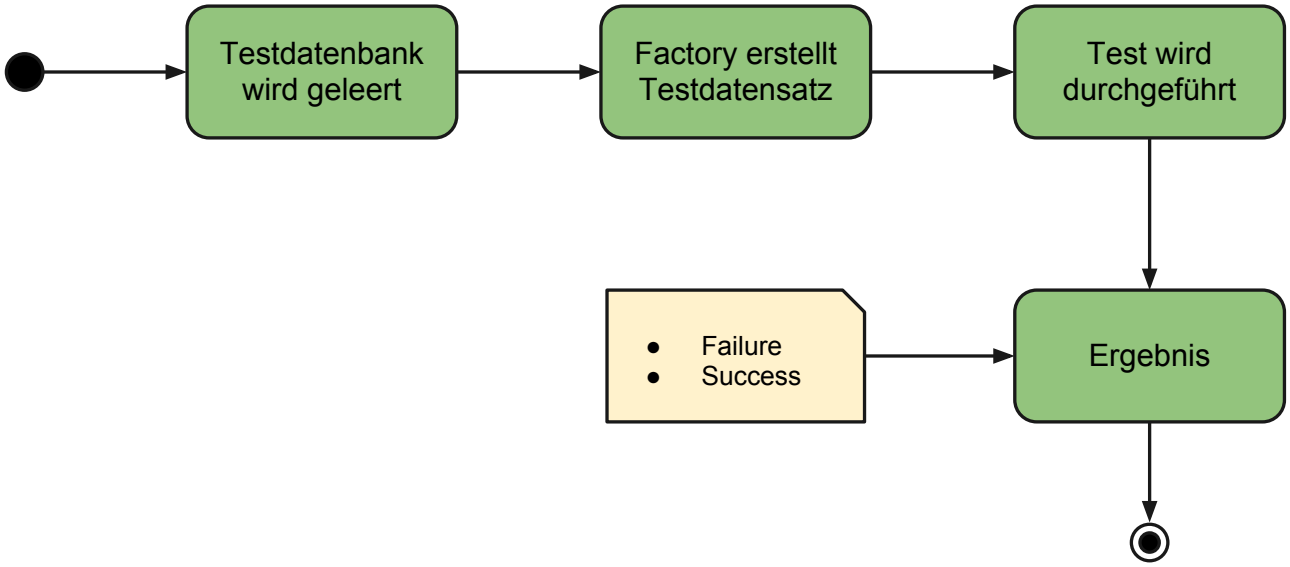
Unit Test

Umsetzung

- Zur Generierung der Testdaten wird eine Factory erstellt
  - Diese wird im Test zuerst validiert
- Jedes Attribut und jede Funktion wird isoliert getestet
  - Dabei werden sowohl die gültigen als auch ungültigen Fälle berücksichtig (z.B. Verletzung des Wertebereiches oder nicht vorhande Werte/Beziehungen)

Unit Test

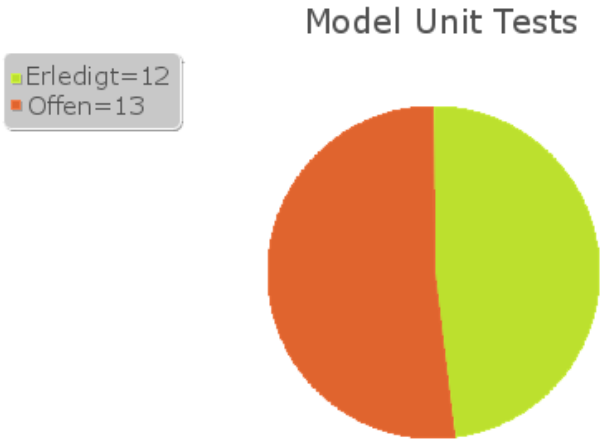
Ablauf



Unit Test

Ist-Stand

- 354 examples, 0 failures, 79 pending
- Die Kern-Modelle sind getest.
  - z.B. Person, OZBPerson, OZBKonto, EE-Konto, ZE-Konto, usw.



Unit Test

Fazit

- Es sind Inkonsistenzen aufgefallen
  - Model stimmte nicht mit der Datenbank überein
  - Validierungen sind nicht vorhanden oder fehlerhaft
  - Umsetzung der Modelfunktionenmit gleicher Funktionalität sind unterschiedlich, oder auch fehlerhaft umgesetzt
- Änderungen im Datenmodell, im Model wirken sich direkt auf die Tests und auf die Anwendunge aus
- Mit einem festgelegten Datenmodell kann die Konsistenz der Modelle, sowie der Anwendung, sichergestellt werden.

Controller Test

Ziele

- Stellt die korrekte Funktionalität der Funktionen im Controller sicher
  - Reagiert der Controller auf die Eingaben des Benutzers korrekt?
- Sicherstellung der korrekten Ausführung privater Funktionen

Controller Test

Voraussetzungen

- Fachlich und technisch korrekter Testdatensatz
- Alle Modelle, die in Verbindung mit dem Controller stehen, sind mit Hilfe von Unit Tests getestet worden
- Anwendungsfälle sind definiert
  - z.B. konkrete Benutzereingaben mit Ergebnissen die erwartet werden

Controller Test

Umsetzung

- Zur Generierung der Testdaten wird eine Factory erstellt
  - Diese wird im Test zuerst validiert
- Jede Funktion wird isoliert getestet
  - Dabei werden sowohl die gültigen als auch ungültigen Fälle berücksichtig (z.B. Verletzung des Wertebereiches oder nicht vorhande Werte/Beziehungen)
  - Vordefinierte Eingabeparameter werden hierbei vorbereitet und mit Hilfe der HTTP Operationen (GET, POST, PUT, DELETE) angewandt
  - Das Ergebnis (= Wert der verfügbaren Instanzvariable des Modells) wird mit den erwarteten Ergebnis verglichen
  - Somit können die Benutzereingaben 1:1 verarbeitet und nachgestellt werden

Controller Test

Ist-Stand

- Testcase: Darlehensverlauf
  - Testdatensatz hierfür ist der vorhandene Datensatz aus der Testdatenbank ozb_test. Er wird mit Hilfe eines Batchskriptes vor jedem Test migriert
  - Implementierung der Test Struktur abgeschlossen
  - Umsetzung für den ersten Testfall abgeschlossen:

```
context "Show Darlehensverlauf of EEKonto 70073"
        context "parameters: anzeigen, vonDatum and bisDatum are nil"
                it "returns the 10 latest buchungen"
```

Controller Test

Fazit

- Controller Tests bieten die Möglichkeit, die Funktionaliäten mit Benutzereingaben zu testen
- Für einen konsistenten Controller Test sind die möglichen Szenarien = Benutzereingaben zu definieren, sowie auch die erwarteten Ergebnisse
- Die Grundvoraussetzung sind die vollständig getesteten Modelle, die auf einem festgelegten Datenmodell basieren.

Datenbankmodell

Nach Änderungen am Datenbankmodell sind folgende Anpassungen notwendig :

- create_tables.txt
- ER-Diagram
- Migration-Tool
- Tabellenübersicht
- Model
  - Validierung, Datenformat, Wertebereich
- Unit Tests
- Testdaten (Factories, SQL-Dumps)
- ...

=> Priorität sollte sein die Korrektheit des Datenbankmodells sicherzustellen und zu finalisieren.

Denn jede Änderung kostet extrem viel Zeit und Aufwand.

Offene Fragen, Issues, Kommentare, ...

Siehe hierzu auch:
https://github.com/Avenel/FirstApp/issues

Issue: KKL Verlauf

- Wenn ein KKL-Verlauf Eintrag gelöscht wird, werden alle zeitlich vergangene (KKLAbDatum) Einträge gelöscht. Weshalb werden ältere Verläufe der Kontenklasse mitgelöscht?

Issue: Buchung

- Beschreibungsnamen für die Felder "wSaldoAcc" und "pSaldoAcc" fehlen.

Issue: Unit Test

- Ist der Sachbearbeiter nun Pflicht?

  JA [ ]
  NEIN [ ]

- Test: Bank: Gibt es einen invaliden Banknamen?
- Bankverbindung:
  - Warum ist BankKtoNr ein varchar und kein Integer?
  - Warum gibt es Datensätze, die einen Bindestrich in der BankKtoNr enthalten?

Issue: Unit Test

Create_Tables:

Ist es richtig, dass in dem Schema des EEKontos:

- der SachPnr *nicht* als ForeignKey ausgewiesen ist?
- die BankID *nicht* als NOT NULL deklariert ist?
- Kreditlimit auf NOT NULL gesetzt, es *muss* einfach ein Kreditlimit geben, sonst machts keinen Sinn.

Habe das mal eben in der create_tables angepasst.

Darf SachPnr NULL sein? Ich lasse dies erst einmal so.

Issue: Unit Test

- pgnr (Projektgruppen-Nr) wird vom Model gefordert, aber in der create tables ist es nicht so. Ich ändere das in der create tables af NOT NULL ab.

Buergschaft:

- Warum sind dort die FOREIGN Keys nicht eingetragen?

Ich ändere das mal.

Issue: Unit Test

Buergschaft: Dort wurde noch keine Historisierung aktiviert. Sprich, jegliche callback Methoden fehlen. Primary Key stimmt nicht. NOT NULL Constraints stimmen auch nicht überein. SichKurzbez, ist das ein enum = {Einzelbuergschaft, Teilbuergschaft} oder nicht? Noch alte Validierungsmethodik. Es wurde offensichtlich seit 1.5 Jahren nix mehr hier getan.

Issue: Unit Test

ZE_Konto:

- Welche Zahlmodi gibt es?
- Welche Möglichkeiten ergeben sich für ZEStatus? Ich habe gerad nur N, D und A gefunden.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings