

UNIVERSITÄT DUISBURG-ESSEN

**Web-Engineering im Web 2.0-
Umfeld mit Ruby On Rails**

Seminararbeit

Vorgelegt dem Fachbereich Wirtschaftswissenschaften
der Universität Duisburg-Essen

von: Torsten Kraemer
Richard-Wagner-Str. 54
45128 Essen
Matrikelnummer: 1465071

Sommersemester 2007, 10. Studiensemester
voraussichtlicher Studienabschluss: Sommersemester 2008

Inhaltsverzeichnis

Inhaltsverzeichnis	I
Tabellenverzeichnis	II
Abkürzungsverzeichnis	III
1 Einführung	1
2 Der Begriff des Web 2.0 und das Ruby On Rails Framework	2
2.1 Eine neue Generation der Webanwendungen.....	2
2.2 Technische Realisierung des Web 2.0.....	3
2.3 Eine Einführung in das Ruby On Rails Framework.....	4
2.3.1 Hintergründe der Entwicklung.....	4
2.3.2 Erste praktische Schritte.....	4
2.3.3 Einsatzmöglichkeiten.....	5
2.3.4 Testing 6	
3 Agile Softwareentwicklung mit Ruby On Rails	8
3.1 Agilität im Web-Engineering	8
3.2 Ruby On Rails im Vergleich mit PHP und ASP.NET.....	9
3.2.1 ASP.NET.....	9
3.2.2 PHP 10	
3.2.3 Direkter Vergleich.....	11
4 Programmierung eines Weblog mit Ruby On Rails	13
4.1 Datenmodell.....	13
4.1.1 Kategorien.....	13
4.1.2 Kommentare.....	14
4.1.3 Blogeinträge.....	14
4.1.4 Benutzer.....	14
4.1.5 Zusammenhänge.....	14
4.2 Anwendungslogik.....	14
4.2.1 ApplicationController.....	15
4.2.2 UserController.....	15
4.2.3 BlogController.....	15
4.2.4 CategoryContoller.....	16
4.2.5 LoginController.....	16
4.3 Präsentationsschicht.....	16
4.3.1 Layouts.....	16
4.3.2 Login 17	
4.3.3 Category und User.....	17
4.3.4 Blog 17	
5 Fazit und Ausblick	18
Anhang A: Quellcode	19
Anhang B: Literaturverzeichnis	31
Eidesstattliche Erklärung	32

Tabellenverzeichnis

Tabelle 2.1: Verbreitung von Webservern Ende 1990er Jahre.....	3
Tabelle 2.2: Apache-Module Ende 1990er Jahre.....	3
Tabelle 3.1: ASP.NET, PHP und Ruby On Rails im Vergleich.....	11/12

Abkürzungsverzeichnis

ASP.....Active Server Pages

IDE.....Integrated Development Environment

PHP.....PHP Hypertext Preprocessor

RoR.....Ruby On Rails

WWW.....World Wide Web

XP.....Extreme Programming

1 Einführung

In den letzten beiden Jahren liest man sowohl in der einschlägigen Fachpresse, die sich mit innovativen Webanwendungen beschäftigt, als auch in der gängigen Tagespresse immer mehr vom sogenannten *Web 2.0*. Was steckt hinter dem Begriff *Web 2.0* und warum wird er sogar in großen Printmedien vermehrt eingesetzt? Handelt es sich nur um einen Hype für die schnelllebige Webentwickler-Community oder sind auch normale Benutzer in ihrem Umgang mit Inhalten aus dem WWW betroffen?

Zumindest letztere Frage ist für die Programmiersprache *Ruby* und besonders das Framework *Ruby on Rails* schnell beantwortet, da es für sich den Anspruch erhebt, Webentwicklern das Programmieren neuer Applikationen so schnell und so einfach wie nie zuvor zu gestalten. Was steckt hinter diesem Framework und was macht es letztendlich so innovativ?

Diese Seminararbeit befasst sich zunächst in Kürze mit einer Definition des Phänomens *Web 2.0*, seinen Auswirkungen auf das Benutzerverhalten im WWW und seinem technischen Hintergrund. Ohne eine umfassende Historie von *Ruby*¹ oder *Ruby On Rails*² wiederzugeben, wird dann dieses relativ junge Framework genauer beleuchtet und eine erste praktische Einführung in die Möglichkeiten gegeben.

Anschließend geht es dann darum, wie im *Web 2.0* eine neue Anwendung konzipiert und umgesetzt werden kann. Dazu betrachte ich vornehmlich agile Softwareentwicklungsmethoden und stelle dem *Ruby On Rails*-Framework die Sprache *PHP* und das *ASP.NET*-Framework gegenüber.

Um einen gängigen Entwicklungsprozess zu illustrieren, wird im dritten Teil dieser Arbeit exemplarisch ein sehr einfaches Weblog von Grund auf programmiert. Es handelt sich hierbei nicht zuletzt also auch um eine praktische Einführung in dieses moderne Framework und eine Demonstration dessen Leistungsfähigkeit.

Abschließend folgen ein Fazit über den derzeitigen status quo in der Webanwendungsentwicklung und ein Ausblick auf die nächsten Jahre.

¹ Siehe dazu <http://swik.net/Rails-Ruby/TechKnow+Zenze/The+History+of+Ruby/nrpd>

² Siehe dazu http://de.wikipedia.org/wiki/Ruby_on_Rails#Geschichte

2 Der Begriff des Web 2.0 und das Ruby On Rails Framework

2.1 Eine neue Generation der Webanwendungen

Der in den letzten Jahren immer weiter verbreitete Begriff *Web 2.0* wurde kurz nach Ende des Internet-Booms Anfang der 2000er Jahre erstmals verwendet. Er definiert sich nicht über eine konkrete Technologie, sondern beschreibt lediglich eine neue Sichtweise („2.0“) auf das WWW („Web“), bei der es primär um die Organisation von Inhalten geht. Inhalte im *Web 2.0* kommen oftmals nicht mehr von einer kleinen Autorengruppe, sondern werden von einem viel größeren Personenkreis gepflegt. In manchen Fällen können Inhalte sogar weitestgehend anonym eingepflegt werden, wenn man von der Protokollierung der IP-Adresse durch den Webseitenbetreiber absieht: Das wohl prominenteste Beispiel in diesem Zusammenhang ist sicherlich die Wikipedia (<http://www.wikipedia.org>), bei der ohne Registrierung und vorige Prüfung eigene Inhalte in ein Online-Lexikon eingepflegt werden können.

Im Laufe der Zeit ist das *Web 2.0* jedoch nicht nur zu einem Synonym für „lebendige“ Webseiten, die durch ihre Benutzer gepflegt werden können, geworden – es steht nun vielmehr auch für „Next Generation (Web) Applications“, die entweder dadurch besonders innovativ sind, dass sie entweder eine völlig neue Idee umsetzen oder eine geschickte Kombination von bereits bestehenden Ideen darstellen.

Als Beispiel für ersteres kann man „Social Networking Websites“ nennen; also Webseiten, die dazu dienen, Besucher nach gewissen Kriterien (Wohnort, Hobbies, Beruf, etc.) zu gruppieren, um ihnen eine Kommunikationsplattform mit persönlichen Seiten bereit zu stellen – dies war ein völlig neuartiges Konzept im Internet, das Facebook (<http://www.facebook.com>) oder MySpace (<http://www.myspace.com>) als eine der ersten Webseiten umsetzten.

Eine Browseranwendung wie GoogleMail (<http://www.googlemail.com>) ist hingegen eine *Web 2.0* Anwendung zweiter Kategorie; sie ist weder die erste Webseite, die ihren Benutzern E-Mail-Zugriff über eine Browseroberfläche bietet, noch die erste, die Javascript einsetzt. Durch eine technisch ansprechend gelöste Kombination ergab sich jedoch eine sehr leicht und komfortabel zu bedienende Anwendung, die einem klassischen lokal installierten E-Mail-Programm ähnelt und die zu einem Vorbild für viele andere E-Mail-Anbieter avancierte.

In der Praxis wird der Begriff *Web 2.0* gelegentlich auch dann als Schlagwort missbraucht, wenn eine Webseite relauncht oder eine neue Community gegründet wird: Foren oder Gästebücher waren jedoch schon im letzten Jahrtausend viel genutzte Webanwendungen; es drängt sich der Verdacht auf, dass wieder einmal in der IT-Welt alter Wein in neue Schläuche gefüllt wird. Eine Differenzierung ist schwierig,

da es sich beim *Web 2.0* – wie eingangs erwähnt – um einen weit gefassten Begriff und keine spezielle Technologie wie HTML, XML oder CSS handelt, deren Definition einem Gremium wie dem *World Wide Web Consortium* unterliegt.

2.2 Technische Realisierung des Web 2.0

In den Jahren seit der Kommerzialisierung des Internets und dem damit verbundenen rapiden Anstieg der Zahl der Internetnutzer – laut [ITUn2007] waren 1990 0,3% der Weltbevölkerung online, heute sind es 15,3% – veränderte sich auch die Technik hinter dem WWW. Während bis in die mittleren 1990er Jahre statische Webseiten dominierten, die dem Benutzer einen festgelegten Inhalt präsentierten, fand zum Ende des Jahrtausends eine starke Orientierung auf neue Technologien wie Perl, ASP oder PHP statt: Die Anzahl der Webserver, auf denen dynamische Technologien eingesetzt werden konnten³, und die Verbreitung der PHP- und Perl-Module für den meist genutzten Webserver Apache stiegen stark an.

Plattform	Verbreitung 1998	Verbreitung 2000
Microsoft IIS	210.892	625.903
Apache	492.321	1.268.923

Tabelle 2.1: Verbreitung von Webservern Ende 1990er Jahre

Quelle: in Anlehnung an [SSWS1998; SSWS2000]

Plattform	Anteil 1998	Anteil 2000
Perl	2,63%	9,97%
PHP	7,96%	32,94%

Tabelle 2.2: Apache-Module Ende 1990er Jahre

Quelle: in Anlehnung an [SSAM1998; SSAM2000]

Damit war es möglich, aus dem passiven Besucher einen Teilnehmer an interaktiven Inhalten zu machen: Durch die Implementierung einer Anwendungslogik konnten Webseiten auf Eingaben reagieren und mit Datenbanken verknüpft werden. Rückblickend wird dabei in der Regel vom *Web 1.0* gesprochen, das im Gegensatz zu den mit *Web 0.5* betitelten statischen Seiten eine hohe Dynamik für Webentwickler und Nutzer brachte.

Nach und nach wurden die technischen Voraussetzungen geschaffen, die letztendlich zum *Web 2.0*-Konzept (vgl. Kapitel 2.1) führten: Im Wesentlichen handelt es sich bei den heutigen Anwendungen um eine Kombination von Technologien, die schon seit vielen Jahren bekannt sind. Eine beliebte und verbreite Variante ist

³ Anm.: Um auf die Verbreitung von ASP zu schließen, muss man auf die Anzahl von Microsoft IIS Installationen zurückgreifen, da dieser als einziger Webserver das Microsoft-eigene proprietäre ASP unterstützt

Apache als Webserver, *MySQL* als Datenbank, *PHP* als Programmiersprache, *HTML* als Auszeichnungssprache zur Ausgabe der Webseiten und *AJAX* als Kombination von *JavaScript* und *XML* zur asynchronen Datenübertragung zwischen dem Server dem Browser des Besuchers.

2.3 Eine Einführung in das Ruby On Rails Framework

2.3.1 Hintergründe der Entwicklung

In einem ganz anderen Bereich der Informatik – aber ebenfalls in den 1990er Jahren – entstand in Japan eine neue objekt-orientierte Programmiersprache namens *Ruby*. Die einfache Syntax und die Tatsache, dass der Programmierer nach einem beliebigen Programmierparadigma arbeiten kann⁴, führten in relativ kurzer Zeit zu einer hohen Verbreitung der Sprache in Japan. Seit etwa 2000 ist diese Sprache auch international bekannter geworden.

Als Durchbruch kann man sicherlich das 2004 von David Heinemeier Hansson entwickelte Framework *Ruby On Rails* bezeichnen: Die zu Grunde liegende einfache *Ruby*-Syntax, eine strikte Einhaltung des Model-View-Controller Prinzips und die beiden Leitsätze „*Don't repeat yourself!*“ (Code wiederverwenden statt redundant programmieren) und „*Convention over Configuration*“ (Konfigurationsdateien vermeiden) führen dazu, dass Webanwendungen in dieser Architektur effizient und mit schnell sichtbaren Erfolgen entwickelt werden können.

Das quelloffene *RoR*-Framework ist aktuell unter der MIT-Lizenz⁵ veröffentlicht und für die gängigen Betriebssysteme (Linux, Unix-Derivate, MacOS, Windows) verfügbar. Hinter dem Framework steckt das *Ruby On Rails Core-Team*⁶, in das Programmierer aufgenommen werden können, die schon über einen längeren Zeitraum aktiv an der Weiterentwicklung beteiligt sind. Dieses Team ist maßgeblich für den *RoR*-Quellcode verantwortlich; Verbesserungsvorschläge, neue Ideen oder Fehler können jedoch über ein webbasiertes Tracking-Tool⁷ gemeldet werden.

2.3.2 Erste praktische Schritte

Die Grundvoraussetzung ist eine *Ruby*-Installation, über die mit dem eingebauten Paketmanager *RubyGems* die Erweiterung *Ruby On Rails* nachinstalliert wird. Auf der Kommandozeile des Betriebssystems kann durch die Eingabe des Befehls

```
rails [projektname]
```

ein Script aufgerufen werden, das die komplette Verzeichnisstruktur für ein neues Projekt erzeugt. Die wichtigsten dieser Verzeichnisse sind:

⁴ So können Programme geschrieben werden, die rein funktional wenige Zeilen Code umfassen, während zum Beispiel in Java zunächst eine komplette objekt-orientierte Grundstruktur erzeugt werden muss

⁵ Siehe <http://www.opensource.org/licenses/mit-license.php>

⁶ Siehe <http://www.rubyonrails.org/core>

⁷ Siehe <http://dev.rubyonrails.org/>

app

Dieses Verzeichnis beinhaltet die verschiedenen Komponenten einer *RoR*-Anwendung (controllers, models, views, helpers)

config

Hier liegen alle Dateien zur Konfiguration der *RoR*-Anwendung, wie zum Beispiel die Definition einer Verbindung zur Datenbank in der Datei `database.yml`

log

Applikationsspezifische Protokolldateien liegen in diesem Verzeichnis.

public

In diesem Verzeichnis werden Elemente des Webdesigns wie Bilder, CSS- oder Javascript-Dateien aufbewahrt.

Obwohl *RoR* einen eigenen Webserver namens *WEBrick* mitbringt, ist eine Einbindung in den verbreiteten Webserver *Apache* problemlos machbar, indem man `mod_ruby` als Erweiterung lädt.

Den am schnellsten sichtbaren Erfolg erzielt man mit dem Erzeugen eines sogenannten Scaffolds („Gerüst“) über die Kommandozeile:

```
ruby script\generate scaffold example
```

Ruby ordnet dem Ausdruck `example` durch das integrierte englische Wörterbuch den Plural „examples“ zu – wenn die referenzierte Datenbank⁸ eine Tabelle „examples“ enthält, kann nach dem Start des Webserver auf diese nun unter

http://[adresse des webserver]/example/

zugegriffen werden⁹. *RoR* erstellt ein Scaffold, das die grundlegenden Operationen zum Datenbankzugriff (**CRUD** – **C**reate, **R**ead, **U**ppdate, **D**elete) zur Verfügung stellt. Legt man in der Tabelle Felder eines bestimmten Typs (zum Beispiel SQL-Typen wie Varchar, Integer, Datetime oder Text) an, sind diese sofort in der *RoR*-Anwendung verfügbar.

2.3.3 Einsatzmöglichkeiten

Auf Grund der hohen Flexibilität von *RoR* ist ein Einsatz in fast jeder Situation möglich, in der Datenmodelle für die Verwendung im Webbrowser aufbereitet werden müssen. In der Natur eines Webentwicklungs-Frameworks liegt es natürlich, dass mit *RoR* keine klassischen Softwareprodukte programmiert werden können, die als eigenständige Applikation auf einem Desktop-PC lauffähig sind. Trotzdem ist es möglich, über eine gemeinsame Datenbasis (üblicherweise ein zentraler Datenbankserver) auch *RoR*-Anwendungen zu entwickeln, die klassische Software ergänzen oder deren Funktion als Webapplikation abbilden.

⁸ Zu definieren unter `/config/database.yml`

⁹ Anmerkung: Auf dieser Ebene unterscheidet *RoR* nicht zwischen Groß- und Kleinschreibung. Der Zugriff über `/example/` ist genauso möglich wie über `/EXAMPLE/`

In eine bestehende homogene Applikationslandschaft (Beispiel: Ein Unternehmen setzt entlang der gesamten Wertschöpfungskette das .NET-Framework ein und möchte bestehende Applikationen in das Web transferieren) lässt sich *RoR* zwar ebenfalls integrieren, allerdings ist es in meisten Fällen deutlich weniger aufwendig, die bestehende Datenmodell- und Steuerungsschicht zu nutzen und nur die Präsentationsschicht neu zu schreiben.

2.3.4 Testing

Ein wichtiger Bestandteil von *RoR* ist die Möglichkeit automatische Tests durchzuführen. Auf diese Weise kann man schon während der Entwicklung einzelne Module einer Anwendung hinsichtlich der Funktionalität und des Umgangs mit fehlerhaften Eingaben testen.

In *RoR* gibt es dazu folgende Möglichkeiten:

- Unit-Tests prüfen einzelne Modelle
- Functional-Tests prüfen den Ablauf eines Controllers
- Integration-Tests prüfen das Zusammenspiel mehrerer Controller

Um Tests durchzuführen empfiehlt es sich, einen eigenen Datenbestand zu definieren, indem man im `test`-Bereich der Datei `/config/database.yml` eine neue Datenbank referenziert und anschließend das Schema der Entwicklungsdatenbank in diese neue Testdatenbank kopiert. Dies geschieht mit dem Befehl:

```
rake db:test:clone
```

Um den Testablauf am Beispiel eines Unit-Tests in Anlehnung an [WiBo2007] zu illustrieren, erzeugt man mit dem Befehl

```
ruby script/generate model Example
```

zunächst das Modell *Example*¹⁰. Es werden nicht nur die für das Modell benötigten Dateien erzeugt, sondern darüber hinaus auch noch ein leerer Unit-Test. In der Datei `/test/fixtures/examples.yml` kann man mit einer sehr simplen Syntax einen Datenbestand anlegen:

```
first:
  id: 1
  title: Erster Eintrag
another:
  id: 2
  title: Und noch einer!
```

¹⁰ Annahme: Das Modell besteht nur aus den Feldern 'id' und 'title'

Um zum Beispiel zu testen, ob ein neues Element eingefügt werden kann, wird in der Datei `/test/unit/example_test.rb` ein neuer Testfall definiert:

```
def example_create
  a = Example.new :title => 'Geht es?'
  a.save!
end
```

Mit der Eingabe des Befehls

```
ruby test/unit/example_test.rb
```

erhält man dann folgende Ausgabe:

```
Started
..
Finished in 0.076603 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Offensichtlich tritt also kein Fehler auf, wenn man ein neues Element mit dem Titel „Geht es?“ einfügen möchte. Wenn das Datenmodell jedoch um einen Validator ergänzt wird, der eine Titellänge von mindestens 10 Zeichen erwartet, scheitert der Test:

```
Started
E.
Finished in 0.072085 seconds.

1) Error:

example_create(ExampleTest):

ActiveRecord::RecordInvalid: Validation failed: Title is too
short (minimum is 10 characters)

(...)
test/unit/user_test.rb:13:in `example_create'

1 tests, 1 assertions, 0 failures, 1 errors
```

Es können beliebig viele Testmethoden für ein Modell geschrieben werden, die in beliebiger Komplexität Daten auslesen, einfügen, ändern oder löschen können. Grundlage für diese Tests ist immer der Programmcode, der unterhalb des Verzeichnisbaums `/app/` liegt.

3 Agile Softwareentwicklung mit Ruby On Rails

Es hat sich bereits im vorigen Kapitel angedeutet, dass das *RoR*-Framework einen Programmierstil fördert, der auf schnell sichtbaren Erfolgen und unkompliziert durchführbaren Tests basiert. Dieser testgetriebene Entwicklungsprozess wirkt vielen Risiken entgegen, die einen großen Einfluss auf den Softwareentwicklungsprozess haben. Dazu zählen nach [EiBS2005, 3] Terminverzögerungen, hohe Fehlerraten, unrentable Systeme, grundlegende Missverständnisse bei den zu implementierenden Geschäftsprozessen und weitere Gefahren.

3.1 Agilität im Web-Engineering

Webanwendungen stellen durchaus eine Besonderheit im Softwarebereich dar, da sie nicht nur auf ein schnelllebiges Medium aufbauen (Änderungen an der Software sind „just in time“ weltweit zu sehen), sondern auch eine geografische Distanz zwischen den Entwicklern sehr einfach machen (Test- und Produktivsysteme sind zentral über das Internet verfügbar).

Es gibt also besondere Anforderungen an einen Softwareentwicklungsprozess für Webanwendungen, die über die traditionellen Entwicklungsmodelle wie beispielsweise das Wasserfall- oder das Spiralmodell nach Boehm hinausgehen. Die agile Softwareentwicklungsmethode *Extreme Programming* basiert auf vier Werten, die diesen Anforderungen besonders gerecht werden (siehe [BeAn2005, 17 ff]):

- **Kommunikation**
Das Internet bietet heute viele Möglichkeiten zur direkten und zeitnahen Kommunikation zwischen den Entwicklern, um einen effizienten Softwareentwicklungsprozess zu unterstützen.
- **Einfachheit**
Agile Lösungen sind Lösungen, die schnell und einfach den gewünschten Effekt haben. Sobald eine Software flexibel ist und in kurzer Reaktionszeit eine neue Anforderung umgesetzt werden kann, ist sie im Internet wettbewerbsfähig.
- **Feedback**
Konstruktive Rückmeldungen zwischen dem Auftraggeber und dem Entwickler sind dafür wichtig, dass eine Software zielgerichtet entwickelt wird und die Geschäftsprozesse richtig umgesetzt werden. Es ist kein Treffen oder die Übergabe eines Speichermediums mit der neuen Softwareversion notwendig, da die Anwendung weltweit (auch gesichert) verfügbar gemacht werden kann.

- **Mut**
Der große Konkurrenzdruck bei Webanwendungen und Anpassungen an den Markt erfordern immer wieder eine Änderung der Prioritäten. Daraus resultieren teilweise fundamentale Änderungen an der Software, die schnellstmöglich umgesetzt werden müssen.

Ein einfaches Design, kurze Feedbackzyklen, andauernde Tests, ständiges Refaktorisieren und die Einhaltung von Programmierstandards sind nur einige der Prinzipien des *XP*, die diese vier Werte nachhaltig sichern sollen. Je modularer und übersichtlicher eine Software aufgebaut wird, desto einfacher können Komponenten ausgetauscht, erweitert und hinzugefügt werden: Funktionalitäten sollen erst dann implementiert werden, wenn diese auch benötigt werden.

Werden diese Prinzipien eingehalten, steht der schnellen Entwicklung einer Webanwendung nichts mehr im Weg. Die beste Architektur ist jedoch nur so gut wie das Werkzeug, mit dem sie sich umsetzen lässt. Bei der Auswahl der richtigen Programmiersprache bzw. des richtigen Frameworks gibt es zahlreiche Alternativen – der Unterschied liegt dabei nicht nur im Detail, sondern macht sich auch schon bei der Umsetzung von grundlegenden Funktionen bemerkbar.

3.2 Ruby On Rails im Vergleich mit PHP und ASP.NET

RoR muss sich einer am Markt etablierten Konkurrenz stellen, wenn es um die Frage geht, in welchem Framework man eine neue Webanwendung entwickeln soll. Auch wenn die Sprache *Python*¹¹ mit den Frameworks *Django*¹² oder *TurboGears*¹³ einen *Ruby/RoR*-ähnlichen Weg geht, gehören PHP als Apache-Modul und ASP.NET auf dem Microsoft IIS zu den heute wohl verbreitetsten Lösungen.

3.2.1 ASP.NET

ASP.NET ist ein von Microsoft entwickeltes Framework, das nicht sprachgebunden ist. Das heißt, dass nur der Rahmen bereit gestellt wird, in dem sich der Programmierer für die unterstützte Sprache seiner Wahl (C++, C#, VB.NET, u.a.) entscheiden kann. Innerhalb des Frameworks können auf Grund der Typensicherheit theoretisch jedoch mehrere Sprachen parallel genutzt werden. Die Compiler übersetzen den jeweiligen Quellcode in eine gemeinsame Bytecode-Sprache, die dann in der „Common Language Runtime“ genannten Virtual Machine ausgeführt wird.

Die einfache Konfiguration von *ASP.NET* hängt damit zusammen, dass dieses Framework sehr stark an das Betriebssystem Windows gekoppelt und nur unter diesem lauffähig ist. Eine alternative Laufzeitumgebung für andere

¹¹ Siehe <http://www.python.org/>

¹² Siehe <http://www.djangoproject.com/>

¹³ Siehe <http://www.turbogears.org/>

Betriebssysteme namens *Mono*¹⁴ wird unter der Federführung von Novell entwickelt. Vertraut man aber auf Erfahrungsberichte, die im Internet diskutiert werden, ist dieses Projekt aber noch nicht ausgereift und reicht nicht an die Funktionalität des Originals heran.

Auf Grund der tiefen Integration in Windows ist das Framework jedoch auch für viele Sicherheitslücken anfällig, die im Hostsystem bekannt werden. Dieser Aspekt sollte gerade bei großen geschäftskritischen Applikationen nicht unberücksichtigt bleiben.

Microsoft bemüht sich, viele umfangreiche und einfach zu verwendende Web Services bereitzustellen, die – ebenso wie die restliche Dokumentation – im Microsoft Developers Network sehr vorbildlich und übersichtlich dargestellt werden.

Die Standard-IDE ist Microsoft Visual Studio.NET. In [McAm2005] wurden die wenigen Alternativen verglichen, die dem Funktionsumfang entsprechen, allerdings gibt es abgesehen von der Kostenfrage keinen Grund, umzusteigen.

3.2.2 PHP

PHP ist eine extrem einfach zu erlernende und weit verbreitete Skriptsprache. Mittlerweile bieten selbst die günstigsten Hostingangebote *PHP*-Unterstützung, so dass eine breite Anwenderschicht ohne besondere Vorkenntnisse relativ schnell einfache Anwendungen entwickeln kann. Die quelloffene Sprache *PHP* ist für die gängigen Betriebssysteme verfügbar und beschränkt sich auch nicht auf eine bestimmte Serverapplikation.

Genau wie *RoR* – und im Gegensatz zu *ASP.NET* – wird Quellcode nicht kompiliert, sondern während der Laufzeit geparkt. Dies kann bei unzureichenden Tests dazu führen, dass schon einfache Syntaxfehler die Anwendung unbenutzbar machen.

PHP bringt sehr Erweiterungen und Funktionen mit, die in einer sehr guten Online-Dokumentation erläutert werden. Da *PHP* sehr verbreitet ist, findet man im Internet eine große Anzahl an Problemlösungen oder Anleitungen.

Das größte Manko bei der Entwicklung komplexer Applikation begründet sich durch die historische Entwicklung dieser Sprache, da ursprünglich kleine Codefragmente in ein bestehendes HTML-Gerüst eingebaut wurden. Funktionen stehen oftmals in der gleichen Datei wie die Ausgabe, so dass Anwendungslogik und Design mitunter vermischt und schwer durchschaubar sind. Abhilfe können diverse Frameworks schaffen, mit deren Hilfe sauber getrennter Code ermöglicht wird.

Besonders große Projekte müssen akribisch geplant, gut strukturiert und mit hoher Disziplin umgesetzt werden, um erfolgreich abgeschlossen werden zu können.

¹⁴ Siehe <http://www.mono-project.com/>

Bei der Wahl einer Entwicklungsumgebung ähnelt PHP wiederum viel mehr *RoR* als dem .NET-Framework. Es finden sich sowohl hoch-integrierte IDEs wie das Eclipse PHP IDE Project¹⁵ als auch unzählige Texteditoren mit Syntax-Highlighting. Da die Sprache sehr simpel aufgebaut ist, können auch mit diesen Texteditoren ansprechende Ergebnisse erzielt werden.

3.2.3 Direkter Vergleich

So ähnlich die Resultate mit den genannten Umgebungen auch sein können, so unterschiedlich sind die Wege, die zu diesen führen. Als klaren Sieger oder Verlierer kann man sicherlich keine Variante bezeichnen, da jede Alternative in ihrem speziellen Anwendungsfeld eine Daseinsberechtigung hat und es natürlich auch von dem persönlichen Erfahrungsschatz eines Entwickler(s)/-teams abhängt, wie stark die jeweiligen Stärken und Schwächen einer Sprache für ein Softwareprojekt ins Gewicht fallen.

	ASP.NET	PHP	Ruby On Rails
Erlernbarkeit	Komplexer Aufbau, Syntax von Sprache abhängig	Skriptsprache mit eingängiger Syntax	Einfacher Aufbau, simple Syntax
Sprache	C#, C++, VB.NET, u.v.m.	PHP	Ruby
Lizenz	Proprietäre kostenpflichtige Software	Kostenlos	Kostenlos
Verfügbarkeit	Windows-Systeme	Gängige Betriebssysteme	Gängige Betriebssysteme
Dokumentation	Vorbildlich und mit vielen Beispielen	Sehr gut, sehr viele Online-Anleitungen zu finden	API dokumentiert, viele gute Anleitungen zum Einstieg
Compiler / Parser	Compiler übersetzt Code in Bytecode	Komplettes Parsing beim Aufruf	Komplettes Parsing beim Aufruf
IDE-Angebot	Wenige Alternativen zu Studio.NET	Unzählige Editoren	Gutes und großes Angebot, auch ein Texteditor genügt

¹⁵ Siehe <http://www.eclipse.org/pdt/>

	ASP.NET	PHP	Ruby On Rails
Software-Tests	z.B. über NUnit	z.B. über PHPUnit	Integriert
Konfigurationsaufwand	Gering, sehr gut in Windows integriert	Einfach, auf vielen Servern bereits vorkonfiguriert	Einfach, da „ <i>Convention over Configuration</i> “
Ajax-Tauglichkeit	Diverse Frameworks stehen zur Verfügung	Gute Frameworks sind Mangelware und umständlich zu bedienen	Integriert, sehr einfach zu bedienen
Trennung der MVC-Schichten	Mit Design Pattern von Microsoft möglich ¹⁶	Nur mit externem Framework möglich	strikte Trennung

Tabelle 3.1: ASP.NET, PHP und Ruby on Rails im Vergleich

¹⁶ Siehe <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/EspWeb-PresentationPatterns.asp>

4 Programmierung eines Weblog mit Ruby On Rails

Den abschließenden Teil dieser Arbeit bildet die Programmierung eines einfachen Weblog-Systems, kurz Blog genannt. Obwohl es dafür bereits zahllose fertige Lösungen gibt, die beständig weiterentwickelt werden, kann man an einem Projekt dieser Art sehr schnell die Funktionsweise von *RoR* erkennen und die Sprache richtig zu nutzen lernen.

Das hier vorgestellte Blog hat eine einfache Benutzerverwaltung¹⁷, eine einfache Kategorienverwaltung und bietet die Möglichkeit, Einträge zu kommentieren. Diese Grundfunktionen sind in ein entsprechendes Layout integriert, das mit einer kleinen AJAX-Demonstration beim Umblättern ein Web 2.0 „Look&Feel“ vermitteln soll.

Zunächst wird eine RoR-Anwendung erzeugt:

```
rails blog
```

Nachdem in der Datei `/config/database.yml` die Datenbankverbindungen angepasst wurden, kann man den Server starten:

```
ruby script/server
```

4.1 Datenmodell

Bevor eine Anwendungslogik implementiert wird, legt man im Datenbank-Frontend seiner Wahl die benötigten Tabellen mit den jeweiligen Feldern in der Datenbank an und lässt *RoR* die Modelle erzeugen, die wir zum Zugriff auf diese benötigen. Im einzelnen sind dies:

4.1.1 Kategorien

Um Blogseinträge thematisch voneinander zu trennen, benötigt man ein Modell, dem beliebig viele Blogseinträge zugeordnet werden können. Die entsprechende Tabelle 'categories' besteht aus einer 'id' und einem Feld 'name' für den Namen der Kategorie. Erzeugt wird das Modell mit:

```
ruby script/generate model Category
```

Namen sollen nur einmal vorkommen und beim Anlegen einer Kategorie natürlich auch nicht leer sein dürfen. In `/app/models/category.rb` fügt man also hinzu:

```
validates_presence_of :name,  
                      :message => "ist nicht vorhanden"  
validates_uniqueness_of :name
```

¹⁷ Es sind hierbei keine Rollen vorgesehen – jeder Benutzer hat alle Rechte.

4.1.2 Kommentare

Für Kommentare erfasst man in einer Tabelle 'comments' die Felder 'id', 'body' für den Kommentartext, 'post_id' als Zuordnung zu einem Blogeintrag, 'created_at' als Information über den Erstellungszeitpunkt, 'author' und 'email' als Informationen über den Verfasser. Das Modell wird äquivalent zu 4.1.1. erzeugt und angepasst.

4.1.3 Blogeinträge

Dieses Modell stellt den Kern der Anwendung dar. Die zugehörige Tabelle 'posts' enthält eine 'id', einen Titel 'title', einen Inhalt 'body', eine Zuordnung zu einer Kategorie 'category_id' und die Informationen über Erstellungs- 'created_at' und Änderungszeitpunkt 'updated_at'. Der Inhalt soll mindestens ein Zeichen umfassen. Das Modell wird äquivalent zu 4.1.1. erzeugt und angepasst.

4.1.4 Benutzer

Das Blog soll durch eine möglichst simple Benutzer-Authentifizierung geschützt werden. Dazu genügt eine Tabelle 'users' mit den Feldern 'username' und 'password'. Das Passwort soll mindestens 10 Zeichen umfassen. Das Modell wird äquivalent zu 4.1.1. erzeugt und angepasst, außerdem definieren wir die Eigenschaft, mit der die korrekte Kombination aus Benutzernamen und Passwort geprüft wird:

```
def self.authenticate(name, password)
  find(:first,
    :conditions => [ "username = '%s' AND password = '%s'",
name, password ]
  )
end
```

4.1.5 Zusammenhänge

Die Modelle Category, Comment und Post hängen folgendermaßen zusammen:

- Category hat viele Posts (has_many :posts)
- Comment gehört zu einem Post (belongs_to :post)
- Post gehört zu einer Category und hat viele Comments (has_many :comments und belongs_to :category)

4.2 Anwendungslogik

Der Kern des Blogs ist die Anwendungslogik. In den Controllern werden die Aktionen definiert, die für den jeweiligen Bereich der Anwendung verfügbar sind. Außerdem erfährt *RoR* so, welche Views es geben muss.

4.2.1 ApplicationController

Dieser Controller ist die standardmäßige Oberklasse, von der sich alle anderen Controller ableiten. In `/app/controllers/application_controller.rb` definiert man das globale Layout und welche Aktion aufgerufen werden soll, falls ein geschützter Bereich aufgerufen wird:

```
layout 'transparentia'
protected
def authenticate
  unless session["person"]
    redirect_to :controller => "login"
    return false
  end
end
```

4.2.2 UserController

Dieser Controller wird komplett gesichert, so dass er unangemeldeten Benutzern nicht zur Verfügung steht. Er enthält in `/app/controllers/user_controller.rb` nur ein einfaches Scaffold, um die Benutzerliste zu bearbeiten:

```
before_filter :authenticate
scaffold :user
```

4.2.3 BlogController

Der BlogController ist das Herzstück der gesamten Anwendung. Er gewährt nur öffentlichen Zugriff auf die Aktionen 'index' (Standard), 'list', 'show' und 'comment':

```
before_filter :authenticate, :except => [ :index, :list,
:show, :comment ]
```

Die Aktionen sind im Einzelnen:¹⁸

- **index:** (Standard) Führe die Aktion 'list' aus
- **list:** Zeigt die Blogbeiträge in einer Übersicht über mehrere Seiten; das Umblättern wird über eine AJAX-Komponente realisiert
- **show:** Zeigt einen einzelnen Blogbeitrag an
- **new:** Zeigt das Formular zum Anlegen eines neuen Blogbeitrags
- **create:** Erzeugt den Blogbeitrag und zeigt dann die Übersicht an
- **edit:** Zeigt das Formular zum Bearbeiten eines Blogbeitrags
- **update:** Aktualisiert die Felder und zeigt dann den Blogbeitrag
- **destroy:** Löscht einen Blogbeitrag und zeigt dann die Übersicht an

¹⁸ Zur ausführlichen Implementierung siehe Anhang A: Quellcode

- **comment:** Speichert einen zu einem Eintrag gehörenden Kommentar und zeigt diesen Blogeintrag dann an
- **delete_comment:** Entfernt einen Kommentar und zeigt dann wieder den zugehörigen Blogeintrag an

4.2.4 CategoryController

Auch dieser Controller ist – wie der UserController – komplett gesichert, so dass er unangemeldeten Benutzern nicht zur Verfügung steht. Er enthält in `/app/controllers/category_controller.rb` ebenfalls ein einfaches Scaffold:

```
before_filter :authenticate
scaffold :category
```

4.2.5 LoginController

Dieser Controller mit der Aktion 'login' wird vom ApplicationController für den Login-Prozess verantwortlich gemacht. Dazu benutzt er die Eigenschaft 'authenticate' des Benutzer-Modells, um entweder den Login-Bildschirm (Aktion 'index' mit entsprechender View) oder nach erfolgreicher Anmeldung die Blogübersicht anzuzeigen. Die Aktion 'logout' zerstört die Sitzung.¹⁹

4.3 Präsentationsschicht

Da sowohl das Datenmodell als auch die die Anwendungslogik implementiert ist, fehlt nur noch die grafischen Darstellung. Für jede definierte Aktion erwartet *RoR* eine entsprechende View, falls nichts Anderes definiert wurde. In Views kann zwar *Ruby*-Code benutzt werden, er sollte jedoch nur dazu eingesetzt werden, Daten auszugeben, damit die Anwendungslogik weiterhin in den Controllern liegt.

Zur Präsentationsschicht gehören ebenfalls Stylesheets, Bilder oder eigene Javascript-Dateien. Diese liegen unterhalb des Ordners `/public/`.

4.3.1 Layouts

Die Datei `/app/views/layouts/transparentia.rhtml` muss genauso heißen, damit das im ApplicationController definierte Layout „transparentia“ für die Ausgaben benutzt wird. In dem Layout setzt man Variablen wie zum Beispiel `<%= yield %>` (Platzhalter für Views der Aktionen), `<%= flash[:notice] %>` (Hinweise) oder `<%= error_messages_for 'post' %>` (Fehlermeldungen vom PostController), die dann zur Laufzeit von *RoR* mit den entsprechendem Content gefüllt werden.²⁰

¹⁹ Zur ausführlichen Implementierung siehe Anhang A: Quellcode

²⁰ Zur ausführlichen Implementierung siehe Anhang A: Quellcode

4.3.2 Login

RoR erwartet für den LoginController nur die View auf die Aktion 'index'. Die Datei `/app/views/login/index.rhtml` erzeugt ein simples Login-Formular, dessen Eingaben an die Aktion 'login' übermittelt werden:

```
<h1>Login</h1>

<div class="item">
  <% form_tag :action => :login do %>

    <p><label for="username">Username</label><br/>
    <%= text_field 'user', 'username' -%></p>
    <p><label for="password">Password</label><br/>
    <%= password_field 'user', 'password' -%></p>

    <%= submit_tag 'Login' %>
  <% end %>
</div>
```

4.3.3 Category und User

Für die Controller User und Category sind keine Views notwendig, da RoR standardmäßig die integrierten Views für ein Scaffold nutzen kann. Diese Views sind zwar nicht aufwendig gestaltet, reichen aber für den einfachen Zweck aus, Benutzer und Kategorien zu editieren.

4.3.4 Blog

Der Großteil der Views findet sich unter `/app/views/blog/`. Dazu gehören:

- **edit**: Rahmen für ein Formular, das an die Aktion 'update' geschickt wird und das hier aus einem Partial gerendert wird.
- **list**: Die Übersichtseite des Blogs mit dem Link zum Anlegen eines neuen Eintrags und Partials für eine Ladeanzeige und die Liste der Blogeinträge
- **new**: Rahmen für ein Formular an Aktion 'create' mit Partial
- **show**: Darstellung eines einzelnen Blogeintrags mit Kommentaren

Die oben erwähnten Partials sind wiederkehrende Designelemente, die in eine separate Datei ausgelagert wurden, damit in den eigentlichen Views keine Redundanz herrscht („Don't repeat yourself!“). Dazu gehören:

- **_form**: Das eigentliche Formular mit allen Feldern eines Blogeintrags
- **_indicator**: Eine Ladeanzeige
- **_post_list**: Die Schleife zur Darstellung der Übersicht aller Blogeinträge

5 Fazit und Ausblick

Beim Kennenlernen des *RoR*-Frameworks habe ich viele (positive) Überraschungen erlebt. Einzelne Aspekte waren mir bereits aus anderen Frameworks und/oder Sprachen bekannt, doch in ihrer Gesamtheit habe ich sie bisher so nur in *RoR* gesehen; eine schöne Parallele zu vielen Web 2.0 Anwendungen, die ihren Mehrwert aus einer Kombination gesammelter Ideen beziehen.

RoR ist nicht das ultimative Framework, das alle anderen überflüssig macht. Es ist jedoch auf Grund der jungen Geschichte ein dem aktuellen Web-Engineering-Stand besonders angemessenes Werkzeug, um effektiv zu entwickeln, ohne sich mit der komplexen Syntax von beispielsweise AJAX auseinander zu setzen.

Der Erfolg der letzten drei Jahre und der große Zulauf werden wohl auch in Zukunft dafür sorgen, dass *RoR* von einem aktiven Programmierer-Team betreut und weiterentwickelt wird. Neuentwicklungen bei Datenbanken oder Technologien zur Browserdarstellung werden weiterhin ihren Weg in das Framework finden.

Wenn man über die funktionale Programmierung hinausdenkt, die über viele Jahre in den Köpfen der meisten Webentwickler verankert war, erkennt man schnell die Vorteile einer klar gegliederten Softwarearchitektur und mehrerer Schichten. Das simple Datenmodell macht den direkten Zugriff auf die darunter liegende Technologie überflüssig – man kann sich alleine auf die Strukturen und deren Weiterentwicklung konzentrieren. Gerade diese Strukturen sind es, die immer weiter in den Fokus einer neuen Generation des Web-Engineerings rücken werden:

Ist die Innovation im Web 2.0 noch die Nutzer- und Designorientierung, so wird nach Meinung der Experton Group in Zukunft „...vor allem die sukzessive Integration semantischer Technologien (...) wesentliche Impulse und Innovationen im E-Commerce bringen. Die unter dem Begriff Semantisches Web oder Web 3.0 bekannten Ansätze verfolgen das Ziel, nicht nur die Nutzer, sondern komplette Datenbestände kontextuell und inhaltlich miteinander zu verknüpfen“. [Velt2007]

Mit *Ruby On Rails* kann man dieser Entwicklung gelassen entgegen schauen.

Anhang A: Quellcode

/app/controllers/application.rb

```
class ApplicationController < ActionController::Base

  layout 'transparentia'

  # Pick a unique cookie name to distinguish our session data from others'
  session :session_key => '_blog_session_id'

  protected

  def authenticate
    unless session["person"]
      redirect_to :controller => "login"
      return false
    end
  end
end
```

/app/controllers/blog_controller.rb

```
class BlogController < ApplicationController

  layout 'transparentia'

  before_filter :authenticate, :except => [ :index, :list, :show, :comment ]

  def index
    list

    if !request.xml_http_request?
      render :action => 'list'
    end
  end

  # GETs should be safe (see
  # http://www.w3.org/2001/tag/doc/whenToUseGet.html)
  verify :method => :post, :only => [ :destroy, :create, :update ],
        :redirect_to => { :action => :list }

  def list
    @teaser_length = 300
    @total = Post.count

    @post_pages, @posts = paginate :posts, :order_by=> "created_at DESC",
    :per_page => 5

    if request.xml_http_request?
      render :partial => "posts list", :layout => false
    end
  end
end
```

```
end

end

def show
  @post = Post.find(params[:id])
  @comment_count = @post.comments.count
end

def new
  @post = Post.new
end

def create
  @post = Post.new(params[:post])
  @post.created_at = Time.now

  # Neuen Blogeintrag speichern
  if @post.save
    flash[:notice] = 'Eintrag wurde erfolgreich angelegt.'
    redirect_to :action => 'list'
  else
    render :action => 'new'
  end
end

def edit
  @post = Post.find(params[:id])
end

def update
  @post = Post.find(params[:id])
  @post.updated_at = Time.now
  if @post.update_attributes(params[:post])
    flash[:notice] = 'Eintrag wurde erfolgreich bearbeitet.'
    redirect_to :action => 'show', :id => @post
  else
    render :action => 'edit'
  end
end

def destroy
  Post.find(params[:id]).destroy
  redirect_to :action => 'list'
end

def comment
```

```

#   Post.find(params[:id]).comments.create(params[:comment])
#   flash[:notice] = "Kommentar eingetragen."

  @new_comment = Post.find(params[:id]).comments.create(params[:comment])
  if @new_comment.save
    flash[:notice] = "Kommentar eingetragen."
    redirect_to :action => "show", :id => params[:id]
  else
#     @post = Post.find(params[:id])
#     @comment_count = @post.comments.count
#     render :action => "show", :id => params[:id]
    @post = Post.find(params[:id])
    @comment_count = @post.comments.count
    render :action => "show", :id => params[:id]
  end

end

def delete_comment
  Comment.find(params[:id]).destroy
  redirect_to :action => 'show', :id => params[:blog_id]
end

end

```

/app/controllers/category_controller.rb

```

class CategoryController < ApplicationController

  # Alles sichern!
  before_filter :authenticate

  # Einfachstes Geruest, um Kategorien zu aendern
  scaffold :category

end

```

/app/controllers/login_controller.rb

```

class LoginController < ApplicationController
  def index
    # show login screen
  end

  def login
    if session["person"] = User.authenticate(params["user"]["username"],
      params["user"]["password"])
      redirect_to :controller => "blog"
    else
      redirect_to :action => "index"
    end
  end
end

```

```
def logout
  session["person"] = nil
  redirect_to :controller => "blog"
end
end
```

/app/controllers/user_controller.rb

```
class UserController < ApplicationController
  # Alles sichern!
  before_filter :authenticate

  # Einfachstes Geruest, um Kategorien zu aendern
  scaffold :user
end
```

/app/helpers/blog_helper.rb

```
module BlogHelper

  def pagination_links_remote(paginator)
    page_options = {:window_size => 1}
    pagination_links_each(paginator, page_options) do |n|
      options = {
        :url => {:action => 'list', :params => @params.merge({:page => n})},
        :update => 'table',
        :before => "Element.show('spinner')",
        :success => "Element.hide('spinner')"
      }
      html_options = {:href => url_for(:action => 'list', :params =>
@params.merge({:page => n}))}
      link_to_remote(n.to_s, options, html_options)
    end
  end

  def comment_author_email_link(comment)
    if comment.email.length > 0
      link_to image_tag("email-contact-blue.gif", :border => 0)+
"+comment.author, "mailto:"+comment.email
    else
      return comment.author
    end
  end
end
```

/app/models/category.rb

```
class Category < ActiveRecord::Base
  has_many :posts
  validates_presence_of :name,
                       :message => "ist nicht vorhanden"
  validates_uniqueness_of :name
end
```

/app/models/comment.rb

```
class Comment < ActiveRecord::Base
  belongs_to :post

  validates_length_of :body,
                    :minimum => 1,
                    :message => " muss mindestens %d Zeichen umfassen!"
end
```

/app/models/post.rb

```
class Post < ActiveRecord::Base
  has_many :comments
  belongs_to :category

  validates_length_of :title,
                    :in => 3..50,
                    :too_long => "darf höchstens %d Zeichen umfassen",
                    :too_short => "muss mindestens %d Zeichen umfassen"
  validates_presence_of :body,
                       :message => "ist nicht vorhanden"
end
```

/app/models/user.rb

```
class User < ActiveRecord::Base

  validates_length_of :password,
                    :minimum => 10

  def self.authenticate(name, password)
    find(:first,
        :conditions => [ "username = '%s' AND password = '%s'", name,
password ]
    )
  end
end
```

/app/views/blog/_form.rhtml

```

<!--[form:post]-->
<p><label for="post_title">Titel</label><br/>
<%= text_field 'post', 'title' %></p>

<p>
  <label for="post_category_id">Kategorie</label><br/>
  <%= select("post", "category_id", Category.find(:all).collect {|c|
[c.name, c.id] }) %>
  <%= link_to "[ Kategorien bearbeiten ]", :controller => :category,
:action => :list %>
</p>

<p><label for="post_body">Inhalt</label><br/>
<%= text_area 'post', 'body' %></p>

<!--[eoform:post]-->

```

/app/views/blog/_indicator.rhtml

```

<%= image_tag("indicator.gif",
              :align => 'absmiddle',
              :border=> 0,
              :id => "spinner",
              :style=>"display: none; padding: 10px;" ) %>

```

/app/views/blog/_posts_list.rhtml

```

<% if @post_pages.page_count > 1 %>
  <p>
    Seite:
    <%= pagination_links_remote @post_pages %>
  </p>
<% end %>

<% for post in @posts %>

  <div class="item">
    <h1>
      <%= link_to post.title, :action => 'show', :id => post %>
      <% if session["person"] %>
        <%= link_to image_tag("edit-page-blue.gif", :border => 0,
:alt => "Seite bearbeiten"), :action => 'edit', :id => post %>
        <%= link_to image_tag("delete-page-blue.gif", :border => 0,
:alt => "Seite entfernen"), { :action => 'destroy', :id => post }, :confirm
=> 'Bist Du sicher?', :method => :post %>
      <% end %>
    </h1>

    <div class="descr">

```

```

    <%= post.category.name %>
    |
    <%= post.created_at.to_s(:long) %>
    <% if post.updated_at > post.created_at %>
      <b>
        | Update:
        <%= post.updated_at.to_s(:long) %>
      </b>
    <% end %>

    <%
      comments_count = post.comments.count
      if comments_count > 0 %>
        |
        <%= comments_count %>
        <%= comments_count == 1 ? "Kommentar" : "Kommentare" %>
      <% end %>
    </div>

    <p>
      <% # durch helper ersetzen - das waers! %>
      <%= textilize(post.body.first(@teaser_length)) %>
      <% if post.body.length > @teaser_length %>
        ...
        <%= link_to "[ weiter ]", :action => 'show', :id => post %>
      <% end%>
    </p>
  </div>
<% end %>

```

/app/views/blog/edit.rhtml

```

<h1>Eintrag bearbeiten</h1>

<div class="item">

  <% form_tag :action => 'update', :id => @post do %>
    <%= render :partial => 'form' %>
    <%= submit_tag 'Bearbeiten' %>
  <% end %>

  <%= link_to '[ Zurück ]', :action => 'list' %>

</div>

```

/app/views/blog/list.rhtml

```

<% if session["person"] %>
  <div class="item">
    <h1><%= link_to 'Neuen Eintrag anlegen', :action => 'new' %></h1>
  </div>
<% end %>

<%= render :partial => "indicator" %>

<div id="table">
  <%= render :partial => "posts_list" %>
</div>

```

/app/views/blog/new.rhtml

```

<h1>Neuer Eintrag</h1>

<div class="item">

  <% form_tag :action => 'create' do %>
    <%= render :partial => 'form' %>
    <%= submit_tag "Anlegen" %>
  <% end %>

  <%= link_to 'Zurück', :action => 'list' %>

</div>

```

/app/views/blog/show.rhtml

```

<div class="item">

  <h1><%= @post.title %></h1>
  <div class="descr">
    <%= @post.category.name %>
    |
    <%= @post.created_at.to_s(:long) %>
    <% if @post.updated_at > @post.created_at %>
      <b>
        | Update:
        <%= @post.updated_at.to_s(:long) %>
      </b>
    <% end %>
  </div>
  <p>
    <%= textilize(@post.body) %>
  </p>
  <%= link_to '[ Zurück ]', :action => 'list' %>
</div>

```

```

<div class="item">
  <h3>Kommentare: (<%= @comment_count %>)</h3>
  <br />
  <%
    if @comment_count > 0
      for comment in @post.comments
    %>
    <div class="commentar_item">
      <div class="descr">
        <%= comment_author_email_link comment %> schrieb <%=
comment.created_at.to_s(:long) %>:
        <% if session["person"] %>
          <%= link_to image_tag("delete-comment-blue.gif", :border =>
0), {:action => "delete_comment", :id => comment.id, :blog_id => @post},
:confirm => 'Bist Du sicher?', :method => :post %>
        <% end %>
      </div>
      <cite><%= textilize(comment.body) %></cite>
    </div>
  <%
    end
  else
  %>
    <p>Es wurden noch keine Kommentare verfasst - vielleicht bist Du der
erste?</p>
  <%
    end
  %>
  <%= form_tag :action => "comment", :id => @post %>
  <p>
    <label for="comment_author">Name</label><br/>
    <%= text_field "comment", "author" %>
  </p>
  <p>
    <label for="comment_email">E-Mail</label><br/>
    <%= text_field "comment", "email" %>
  </p>
  <p>
    <label for="comment_body">Name</label><br/>
    <%= text_area "comment", "body" %>
  </p>
  <%= submit_tag "Kommentieren!" %>
</form>
</div>

```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>

<head>
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1"/>
<meta name="description" content="description"/>
<meta name="keywords" content="keywords"/>
<meta name="author" content="author"/>
<%= stylesheet_link_tag 'transparentia' %>
<%= javascript_include_tag :defaults %>
<title>Web-Engineering mit RoR</title>
</head>

<body>

<div class="container">

  <div class="main">

    <div class="header">
      <div class="title">
        <h1>Web-Engineering mit RoR</h1>
      </div>
    </div>

    <div class="content">

      <% if flash[:notice] %>
        <div class="flashnotice">
          <%= flash[:notice] %>
        </div>
      <% end %>

      <%= error_messages_for 'post' %>
      <%= error_messages_for 'comment' %>

      <%= yield %>

    </div>

    <div class="sidenav">

      <h1>Blog</h1>
      <ul>
        <li><a href="/blog/">Einträge</a></li>
      </ul>
    </div>
  </div>
</div>
</body>
</html>
```

```

                <% if session["person"] %>
                    <li><a
href="/category/">Kategorien</a></li>
                <% end %>
            </ul>

            <% if session["person"] %>
                <h1>User-Verwaltung</h1>
                <ul>
                    <li><%= link_to "Übersicht", :controller =>
"User" %></li>
                </ul>
            <% end %>

            <h1>Session</h1>
            <ul>
                <% if session["person"] %>
                    <li><a
href="/login/logout">Logout</a></li>
                    <% else %>
                        <li><%= link_to "Login", :controller =>
"Login" %></li>
                    <% end %>
                </ul>

            </div>

            <div class="clearer"><span></span></div>

        </div>

        <div class="footer">
            <span class="left">&copy; 2007 - Valid <a
href="http://jigsaw.w3.org/css-validator/check/referer">CSS</a> &amp; <a
href="http://validator.w3.org/check?uri=referer">XHTML</a></span>
            <span class="right"><a
href="http://templates.arcsin.se">Website template</a> by <a
href="http://arcsin.se">Arcsin</a></span>
            <div class="clearer"><span></span></div>

        </div>
    </div>
</body>
</html>

```

/app/views/login/index.rhtml

```

<h1>Login</h1>
<div class="item">
  <% form_tag :action => :login do %>

    <p><label for="username">Username</label><br/>
    <%= text_field 'user', 'username' -%></p>

    <p><label for="password">Password</label><br/>
    <%= password_field 'user', 'password' -%></p>

    <%= submit_tag 'Login' %>
  <% end %>
</div>

```

Gesamtübersicht laut Ausgabe von 'rake stats':

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
Controllers	154	103	5	14	2	5
Helpers	34	30	0	2	0	13
Models	35	31	4	1	0	29
Libraries	0	0	0	0	0	0
Components	0	0	0	0	0	0
Integration tests	0	0	0	0	0	0
Functional tests	146	105	8	19	2	3
Unit tests	46	32	4	5	1	4
Total	415	301	21	41	1	5

Code LOC: 164 Test LOC: 137 Code to Test Ratio: 1:0.8

Anhang B: Literaturverzeichnis

- ITUn2007 International Telecommunication Union: Main (fixed) telephone lines, mobile cellular subscribers, and Internet users (per 100 population). <http://www.itu.int/ITU-D/ict/mdg/storyline/index.html>, 2007-05-07, Abruf am 2006-08-08
- SSWS1998 Security Space: Web Server Survey. http://www.securityspace.com/s_survey/data/199807/index.html, 1998-08-01, Abruf am 2006-08-08
- SSWS2000 Security Space: Web Server Survey. http://www.securityspace.com/s_survey/data/200007/index.html, 2000-08-01, Abruf am 2006-08-08
- SSAM1998 Security Space: Apache Module Report. http://www.securityspace.com/s_survey/data/man.199807/apache_mods.html, 1998-08-01, Abruf am 2006-08-08
- SSAM2000 Security Space: Apache Module Report. http://www.securityspace.com/s_survey/data/man.200007/apache_mods.html, 2000-08-01, Abruf am 2006-08-08
- WiBo2007 Wiki Books: Ruby On Rails: Erste Schritte: Testen. http://de.wikibooks.org/wiki/Ruby_on_Rails:_Erste_Schritte:_Testen, 2007-05-16, Abruf am 2007-07-28
- EiBS2005 Eicker, Stefan; Beul, Michael; Spies, Thorsten: Paradigmen und Konzepte der Softwareentwicklung I (Skript zur Vorlesung, Version 1.1). 2005.
- BeAn2005 Beck, Kent; Andres, Cynthia: Extreme Programming Explained: Embrace Change (Second Edition). Addison Wesley, Boston 2005.
- Helf2003 Helff, Martin: PHP, ASP, J2EE und CFML im Vergleich. In: MX Magazin (2003) 1, S. 66 - 68
- McAm2005 McAmis, David: Im Test: .NET-Entwicklung ohne Visual Studio. <http://www.zdnet.de/builder/architect/0,39023548,39135838,00.htm>, 2005-08-18, Abruf am 2007-08-01
- Velt2007 Velten, Carlo: Kunden prägen Online-Geschäft. In: COMPUTER ZEITUNG 38 (2007) 30-31, S. 15

Eidesstattliche Erklärung

Ich versichere an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift