Tirade

Dokumentation
für den
Administrator

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Buch werden die englischen Fachbegriffe benutzt, damit ein Umstieg auf die zum größten Teil englischsprachige Literatur leichter fällt. Erklärungen für die meisten Begriffe wie Model, Controller oder Webserver werden ggf. kurz definiert.

1.1 Ruby on Rails

1.1.1 Ruby

Frei nach `http://de.wikipedia.org/wiki/Ruby_(Programmiersprache)`

> Ruby (engl. für Rubin) ist eine interpretierte, objektorientierte Programmiersprache, die mehrere weitere Programmierparadigmen (Prozedurale Programmierung, Funktionale Programmierung, Nebenläufigkeit) unterstützt.
>
> Ruby wird seit dem 24. Februar 1993 von Yukihiro „Matz“ Matsumoto in Japan entwickelt und wird heute als Open-Source-Projekt weitergepflegt. In Europa und Amerika wurde Ruby ab dem Jahr 2000 durch erste nicht-japanische Literatur bekannt.

> Der Name basierte ursprünglich auf dem Edelstein Rubin und wird heute auch als Anspielung auf die Programmiersprache Perl verstanden.

1.1.2 Ruby on Rails

Frei nach `http://de.wikipedia.org/wiki/Ruby_on_Rails`

> Ruby on Rails, kurz Rails oder RoR, ist ein von David Heinemeier Hansson in der Programmiersprache Ruby geschriebenes und quelloffenes Web Application Framework. Es wurde im Juli 2004 zum ersten Mal der Öffentlichkeit vorgestellt wurde.
>
> Es basiert auf den Prinzipien „Don't Repeat Yourself" (DRY) und „Convention over Configuration", d.h. statt einer variablen Konfiguration sind Konventionen für die Namensgebung von Objekten einzuhalten, woraus deren Zusammenspiel sich automatisch ergibt. Diese Funktionen ermöglichen eine rasche Umsetzung von Anforderungen und damit agile Softwareentwicklung.

1.1.3 Motivation

Der Grund, Ruby on Rails zur Erstellung eines Web-Basierten Content Management Systems zu benutzen, erschließt sich aus leicht vorangegangenem.

Es stellt die Grundlegenden Funktionalitäten, welche in einem CMS gebraucht werden, schnell, einfach und geordnet bereit, ohne dabei negativ einschränkend zu wirken oder sich dem Entwickler in den Weg zu stellen.

Es baut auf existierenden und bewährten Standards auf, zum Beispiel SQL als Anfragesprache für die Datenbank oder CSS, wie es (fast) jeder Webbrowser uterstützt. Dadurch wird das Rad nicht neu erfunden, sondern auf solidem Boden ein komplexes Fundament errichtet, auf dem Ruby on Rails und deren Benutzer aufbauen können.

Zudem stellt die Benutzergemeinde um Ruby on Rails herum eine riesige

Menge aus Dokumentationen, Experimenten oder nützlichen Plugins bereit, mit der man die eigene Applikation erweitern kann. Dabei ist der benutzte Programmcode leicht erreichbar (offener Quellcode) und zum größten Teil auch verständlich, so dass das Nachprüfen auf z.B. Sicherheitsfragen einfach zu bewerkstelligen ist.

Somit bestand der Großteil der Arbeit an `Tirade`, die existierenden Techniken, Ideen und Konzepte sinnvoll zu kombinieren und den Endnutzern zugänglich zu machen.

Kapitel 2

Installation

2.1 Voraussetzungen

Für den Betrieb von `Tirade` sind eine Reihe von Paketen, Programmen und Zusätzen vonnöten. Zum größten Teil gehören diese zum sogenannten `Stack`. Dieser besteht aus Betriebsystem, Webserver (ggf. mit Proxy), einer Datenbank, dem Ruby on Rails Framework und dem darauf aufsetzenden Tirade-Quellcode.

Eine Installation kann sowohl auf einem nativen als auch virtuellen System erfolgen. Aus Gründen der Leistung wird allerdings eine native Maschine empfohlen. Alternativ sollte das virtuelle System über genügend Speicher verfügen.

Geht man davon aus, dass der gesamte Stack auf einer Maschine betrieben wird, stellt ein Server mit ca. 1 GHz Taktrate und mindestens 256 MByte (besser 512 MByte) die untere Grenze dar. Kommt eine graphische Pberfläche zum verwalten des Servers dazu, muss der Speicher ggf. entsprechend erhöht werden.

2.1.1 Betriebsystem

Als grundelgendes Betriebsystem eignen sich prinzipiell alle handelsüblichen, auf welche Ruby portiert wurde. Dies schließt alle Linux-Distributionen, prinzipiell alle BSDs und Microsoft Windows ein.

Zum Betrieb von Tirade wird ein Unix-basiertes System empfohlen, insbesondere die aktuelle Version von Ubuntu Linux (`http://ubuntulinux.com`). Dieses vereint die Stabilität von kommerziellen Linuxdistributionen mit der Aktualität der mitgelieferten Softwarepakete eines selbst aufgesetzen Systems wie Gentoo. Somit sind alle benötigten Pakete in ausrechend fortgeschrittener Version vorhanden.

Wir setzen ab nun eine frische Standardinstallation voraus, auf welcher der lesende Admin root/sudo Rechte hat und das Internet verfügbar ist, da ein großer Teil der benötigten Software direkt über das Netz (nicht nur HTTP) installiert und aktualisiert wird.

2.1.2 Ruby on Rails

Ruby

Es wird Ruby mit der Version $\geq$ 1.8.6 empfohlen. Unter Ubuntu sind dazu folgende Pakete zu installieren:

```
aptitude install ruby rdoc ruby1.8 ruby1.8-dev rake irb
```

Ältere Versionen weisen kleine, aber signifikante Fehler auf, die allerdings zu speziell sind, um hier auf sie einzugehen.

RubyGems

Ruby besteht zum einen aus der Sprache selbst, zum anderen aus einer großen Gemeinschaft von Entwicklern. Diese stellen ihre Eigenentwicklungen meist

als sogenanntes `rubygem` zur Verfügung. Um deren Installation zu erleichtern, bietet sich mit den RubyGems ein Paketverwaltungsprogram an.

Die RubyGems können

1. in der aktuellen Version von der Projektseite `http://rubygems.org/` heruntergeladen oder
2. mit Hilfe von `apt(itude) install rubygems` mit Hilfe der Distribution

installiert werden.

Anschließend reicht `gem install`, um ein freigegebendes Zusatzpaket zu installieren.

Rails

Ruby on Rails mit Hilfe der RubyGems installieren. Dabei sollte man sich bei dem Betrieb von Tirade „Version 1“ nur die angegebene Version installieren, da Tirade noch nicht auf Rails 2.* angepasst wurde.

```
gem install rails -v 1.1.4  -y
```

Dabei werden auch die Abhängigkeiten (ActiveRecord, ActionPack usw.) mitinstalliert.

Zusätzliche Gems

Weitere Gems, die für die Textformatierung benötigt werden:

```
gem install RedCloth
```

2.1.3 Webserver

Als Webserver eignen sich alle Webserver, die entweder FastCGI oder ähnliche Techniken unterstützen, um sogenannte CGI-Skripte auszuführen. Allerdings ist das aufsetzen auf Grund der Komplexität des Ruby on Rails Frameworks nicht so einfach, wie zum Beispiel mit PHP (dies hat sich erst kürzlich durch das Passenger Projekt geändert).

Die gängige Art und Weise in 2006, eine Rails-Applikation zu betreiben, stellte Lighttpd (`http://www.lighttpd.net/`) mit FastCGI dar.

Dazu sind in Ubuntu folgende Pakete zu installieren:

```
aptitude install lighttpd libfcgi-ruby1.8
```

2.1.4 Datenbank

Es kann jeden von Rails unterstütze Datenbank benutzt werden. Hier wird PostgreSQL verwendet. Unter Ubuntu installiert man folgende Pakete:

```
aptitude install postresql libpgsql-ruby1.8
```

2.1.5 Zusätze

Für die automatische Skalierung von hochgeladenen Bildern werden ImageMagick inkl. der passenden Bindings für Ruby benötigt.

```
aptitude install libmagick9-dev librmagick-ruby1.8
```

2.1.6 Tirade

Der Quellcode von Tirade ist offen lesbar und befindet sich jederzeit in Überarbeitung und Weiterentwicklung. Deswegen wird dieser in einem

öffenltich zugänglichen Subversion-Repository (SVN) angeboten, wobei es diverse Stränge gibt, zwischen denen man bei der Installation wählen kann. Hier benutzen wir den als 'stable' getaggten Zweig, in den keine weiteren Features, aber Bugfixes eingefügt werden.

Zuerst muss Subversion selbst installiert werden:

```
aptitude install subversion
```

Danach kann der Befehl `svn` benutzt werden, um den Quellcode von Tirade 'auszuchecken', also eine lokale Kopie der aktuellen Version (im Zweig) zu erzeugen.

```
mkdir -p /var/www
cd /var/www
svn checkout svn://lanpartei.de/tirade/tags/stable tirade
```

Dabei wird ein Verzeichnis `/var/www/tirade/` angelegt, was wir von nun als als RAILSROOT bezeichnen.

Kapitel 3

Konfiguration

Hier wird de Konfiguration des oben erwähnten und installierten `Stack` erläutert.

3.1 Datenbank

Um eine neue Datenbank und zugehörigen Benutzer zu erzeugen, muss/sollte man unbter Ubuntu den dafür eingerichteten Benutzer 'postgres' verwenden.

```
sudo su - postgres
createuser --no-superuser --no-createdb --no-createrole -P tirade
createdb -e -O tirade tirade
exit
```

Das dabei eingegebenen Passwort kann man nun in die entsprechende Konfigurationsdatei eintragen:

```
cd /var/www/tirade
cp config/database.yml.example config/database.yml
vim config/database.yml
```

Darin sollte der Abschnitt 'production:' komplett ausgefüllt werden, zb:

```
production:
  adapter: postgresql
  database: tirade
  username: tirade
  password: allyourbasearebelongtous
  host: localhost
```

Von nun an, ist es hilfreich, die Umgebungsvariable 'RAILS_ENV' auf 'production' zu setzen, da sonst Rails/Tirade langsamer läuft und/oder die falsche Datenbank benutzt. Dazu muss folgendes auf der Shell (zB. bash) eingegeben und in `/.bashrc` hinzugefügt werden:

```
export RAILS_ENV=production
```

Nun kann probiert werden, Tirade sich selbst installieren zu lassen. Im RAILSROOT:

```
rake install
```

Bitte genau auf die Ausgaben achten. Der Befehl muss ggf. ein zweites Mal eingegeben werden. Die häufigsten Probleme treten bei der Datenbankverbindung auf. Da es sich höchstwahrscheinlich nicht um ein Tirade-spezifisches Problem handelt, kann meist auch eine Google-Suche nach der Fehlermeldung weiterhelfen. Falls auch dies nicht zum Erfolg führt, kann ein Ticket im trac erzeugt werden (`http://trac.lanpartei.de/trac/tirade/newticket`).

3.2 Webserver

Um Tirade mit Lighttpd zu betreiben, ist eine Konfigurationsdatei vonnöten. Diese kann als Modul die Anfragen für einen Virtualhost annehmen und entsprechend an Rails weiterleiten.

```
$HTTP["host"] =~ "^tirade\.example\.com" {
   server.document-root       = "/var/www/tirade/public/"
   server.error-handler-404 = "/dispatch.fcgi"
   index-file.names           += ( "dispatch.fcgi" )
   fastcgi.server    = ( "/dispatch.fcgi" =>
     ((
         "socket" => "/tmp/tirade.socket",
         "bin-path" => "/var/www/tirade/public/dispatch.fcgi",
         "min-procs" => 1,
         "max-procs" => 4,
         "bin-environment" => ("RAILS_ENV" => "production"),
       )))
   server.errorlog = "/var/log/tirade.error.log"
}
```

Diese Datei kann man unter `/etc/lighttpd/conf-available/23-tirade.conf` abspeichern und mit `lighty-en-mod tirade` aktivieren. Die Pfade sind eventuell anzupassen. Dabei muss beachtet werden, dass:

1. 'server.document-root' auf das Verzeichnis 'public' im RAILSROOT zeigt, und
2. 'bin-path' auf das darin liegende 'dispatch.fcgi'.

.

Startet man nun den lighttpd neu `sudo /etc/init.d/lighttpd restart`, sollte man mit einem Webrbrowser auf dem entsprechenden Host (hier 'http://tirade.example.com') sein Ergebnis bewundern können dürfen.

Kapitel 4

Aufbau und Anpassung

Im folgenden Abschnitt wird davon ausgegangen, dass Tirade in ein bestimmtes Verzeichnis installiert wurde. Dieses Verzeichnis (oder auch Ordner) wird ab jetzt als RAILSROOT bezeichnet. Dies rührt daher, dass die Verzeichnisstruktur bei jeder Ruby on Rails-Anwendung vorzufinden ist.

Wenn also Tirade im Verzeichnis `/var/www/tirade/` installiert ist, verweist `app/` auf das Verzeichnis`/var/www/tirade/app/`. Alle Kommandos sollten vom Railsroot aus ausgeführt werden, damit zum Beispiel bei einem Update (siehe 5.1) die gesamte Verzeichnisstruktur von Tirade aktualisiert wird und nicht nur das aktuelle Verzeichnis.

4.1 Objekttypen

4.1.1 Controller

Controller sind dafür da, die Anfragen, welche der Benutzer mit seinem Webbrowser mit einer URL stellt (zB. `http://daschube.lanpartei.de/manage/show/23`), entgegenzunehmen und zu verarbeiten.

Dabei ist meist der erste Teil der URL nach dem Rechnernamen meist der

Name des Controllers. Dies ist eine der automagischen Verhaltensweisen, die Ruby on Rails mit sich bringt. In der obigen URL wird im *ManageController* die Methode *show* aufgerufen. Dabei wird ein Parameter übergeben, und zwar *:id => 23*. Die Methode *ManageController.show* sucht dann ein Model (Document oä.) mit der *id* 23 aus der Datenbank und zeigt dieses mit Hilfe des Views `app/views/admin/show.rhtml` an. Auch hier sieht man, dass die Dateien der Views passend benannt wurden, damit sie automagisch gefunden werden.

Dieses Verhalten kann selbstverständlich auch geändert werden. Das Zerlegen der URLs und zuordnen zu den Controllers wird durch Routen (siehe 2.1.6) erledigt. Andere Views können aus dem Controller heraus mit *render* aufgerufen werden. Somit sind auch Urls wie `http://daschube.lanpartei.de/content/Meta/Benutzerhandbuch` möglich.

4.1.2 Model

Models sind dafür da, den Zustand der Applikation zu speichern. Dazu speichern sie Daten, bestimmen das Verhalten in Bezug auf diese Daten und sorgen für Validität.

Die meistverwendete Form des Models erbt von *ActiveRecord::Base*. Diese Klasse ist einer der wichtigsten Bestandteile von Ruby on Rails und stellt im Prinzip eine objektorientierte Schnittstelle zu einer Datenbank dar. Kurz zusammengefasst ergeben sich damit folgende Eigenschaften/Möglichkeiten:

- Models sind im Singular benannt, ihre Tabellen dagegen im Plural (Model *User* vs. SQL-Tabelle *users*).
- Es werden verschiedene Klassenmethoden zum finden, suchen usw bereitgestellt: *Document.find_by_title(„hail eris“)* usw.
- Relationen (1:n,1:1,n:m) durch *has_many*, *belongs_to* usw.
- leichter Erweiterbarkeit durch Acts-As-Methoden: die Hierarchie der Folder wird durch *acts_as_tree* erzeugt, welche dann automatisch Methoden wie *#child* oder *#siblings* zur Verfügung stellt.

- eigene Methoden die sehr einfach selbst geschrieben werden können, zb der Pfad für jeden Content-Typ

In Tirade werden folgende Models verwendet:

- *User*: stellt einen Benutzer da, eine individuelle Person, welche Tirade benutzt.
- *Content*: Die Oberklasse für alle Inhalte, die über die Webseite erreichbar sein sollen. Davon erben:
  - *Document*: Eine Seite, die Text enthält.
  - *Folder*: Ein Seite, welche mehrere Unterseiten enthält. Diese bilden eine Hierarchie (*acts_as_tree*)
  - *Address*: Eine Adresse. Deren Felder wie Straße, Hausnummer, PLZ usw werden serialisiert im *body* gespeichert.
  - ...
- *Image*, *Attachment*: Bilder und Anhänge
- *Permission*, *Granting*, *Role*: Zugriffsrechte, Rollen und Gewährungen - siehe 1.1.3

4.1.3 View

Views (Ansichten) sind für die Benutzerschnittstelle zuständig und nehmen dabei meist die (Daten der) Models zur Hilfe. Zum Beispiel ist die einfache Liste der Documents in einem Folder ein View; die Tabelle, um diese Documents nach kriterien zu Ordnen (Größe, Erstellungsdatum) ein komplett anderer View.

Auch Formulare für das Erstellen oder Ändern einer oder mehrerer Models sind Views. Sie zeigen Daten also nicht nur an, sondern bieten auch eine benutzerfreundliche Schnittstelle zum Ändern derselben.

Views können in verschiedenen Formen auftreten:

- Mit `.rhtml`-Dateien können leicht Webseiten in HTML generiert werden.
- `.rxml`-Dateien erzeugen XML-Code jeglicher Art und Weise, zum Beispiel RSS/Atom-Feeds, RPC-Aufrufe oder auch Office-Dokumente im OpenDocument-Standard.
- weiteres: Grafische Oberflächen (Fenster), LaTex, PDF usw.

In Tirade werden zum jetzigen Zeitpunkt zum größten Teil nur Webseiten mit Hilfe von `.rhtml` erzeugt. Das R in `.rhtml` steht für Ruby (siehe 1.1.1) und verweist darauf, dass alle dynamischen Inhalte mit hilfe dieser Programmiersprache erzeugt werden.

Hier ein schon nichtmehr trivialiales Beispiel eines Views für eine sortierbare Liste von Inhalten in einem Folder.

```
# app/views/manage/list.rhtml
<p id="description">
  <%= finalize_body(@folder.description) %>
</p>

<ul id="content_list">
  <%= render :partial => 'content_list_item', :collection => @folder.contents
</ul>
<%= sortable_element 'content_list',
  :url => { :action => 'order' },
  :update => 'content',
  :complete => visual_effect(:highlight, "content_list", :duration => 2.3)
%>

<% content_for('actions') do %>
<%=  render :partial => 'folder_bar', :object => @folder %>
<% end %>
```

Wie man leicht sieht, werden in HTML-Code dynamische Elemente eingefügt. Diese Vorgehensweise erinnert leicht an PHP. Allerdings wird empfohlen, in den Views so wenig Logik einzubringen wie möglich und diese besser

in Models oder Controller auszulagern. Zum Beispiel sollten in einem View nie explizit Anfragen an die Datenbank gestellt werden.

`<% end %>`: Der Ruby-Code innerhalb der spitzen Klammern mit Prozentzeichen wird ausgeführt, aber dessen eventuelles Resultat nciht ausgegeben. Diese Konstrukt eignet sich zum Beispiel für kleine Schleifen oder Viewspezifische Bedingungen (zb dass ein Button nur angezeigt wird, wenn der eingeloggte Benutzer das Recht dazu hat, darauf zu klicken).

`<%= @user.fullname %>`: Hier wird der Ruby-Code ausgeführt und der erzeugte Wert ins HTML-Dokument eingefügt. In diesem Fall würde also der volle Name des in der Instanzvariable `@user` gespeicherten Benutzers angezeigt werden.

Im obigen Beispiel sind noch zwei weitere praktische Aspekte zu entdecken.

4.1.4 Helper

Zum einen stehen dem View verschieden Funtionen zur Verfügun, mit deren Hilfe die Anzeige extrem vereinfacht werden kann. Diese Funktionen werden Helper (siehe 4.1.4 genannt. `finalize_body` ist ein Tirade-eigener Helper, der den vom Benutzer eingegebenen Text in HTML umwandelt (z.B: „viele *dicke* Preiselbeeren“ → „viele ¡b¿dicke¡/b¿ Preiselbeeren.); `sortable_element` erzeugt JavaScript-Code, der dafür sorgt, dass die Elemente der angegebenen Liste per Drag&Drop umsortiert werden können.

4.1.5 Partials

Zum anderen werden die Elemente der Liste „content_list“ (bzw. Inhalte des Folders) mit Hilfe eines sog. Partials angezeigt. Partials sind kleine Schnipsel von `.rhtml`-Code, die wiederverwendbar sind. In diesem Beispiel wird für jedes Element vom Inhalt des Folders (`@folder.contents`) folgendes gerendert (mit Hilfe vom Argument `:collection`.

```
# app/views/manage/_content_list_item.rhtml
```

```
# partials beginnen immer mit einem Unterstrich

<li id="item_<%= content_list_item.id -%>">
<%= content_drag(content_list_item) %>
</li>
```

Im Partial selbst ist das aktuelle Element als lokale Variable verfügbar, die den selben Namen wie das Partial trägt (`content_list_item`).

In Tirade werden Partials etwas missbraucht, indem für jede Klasse, die von Content erbt, ein eigenes Partial zur verfügung gestellt wird. Nehmen wir als Beispiel die Klasse Document, welche zum Speichern von größeren Mengen von Text geeignet ist. Folgende Partials stehen zu Verfügung:

- `app/views/manage/_document.rhtml`: Vollansicht des Dokumentes mit Beschreibung und Haupttext im Manage-Bereich.
- `app/views/manage/_document_preview.rhtml`: Vorschauansicht des Dokumentes, wird im im Editor verwendet, um das Aussehen des Dokumentes beim Bearbeiten zu beobachten.
- `app/views/manage/_document_form.rhtml`: Das Formular für den Editor. Zusätzliche Felder, die bearbeitet werden können, werden hiere zum eitieren angeboten.
- `app/views/public/_document.rhtml`: Die öffentliche Vollansicht des Dokumentes.
- `app/views/public/_document_preview.rhtml`: Die öffentliche Vorschau des Dokumentes, wenn der Elternordner angezeigt wird. Meist soll nur ein Auszug angezeigt werden.
- `themes/eigenes_thema/views/public/_document.rhtml`: Die öffentliche Vollansicht des Dokumentes im eigenen Thema (siehe 4).
- `themes/eigenes_thema/views/public/_document_preview.rhtml`: Die öffentliche Vorschau des Dokumentes im eigenen Thema (siehe 4).

Wenn also im Manage-Bereich ein Document in Vollansicht gezeigt werden soll, wird die Datei `app/views/manage/_document.rhtml` gesucht. Wird diese nicht gefunden, wird die Oberklasse von Document ermittelt (Content) und dafür das passende Partial gesucht, also `app/views/manage/_content.rhtml`. Dies gilt auch für Formulare und Vorschauansichten, sowohl in Manage als auch im Öffeltichen Bereich. Somit ist es nicht nötig, bei einem neuen Content-Typ sofort alle Partials mitzuliefern, solange der neue Typ sinnvoll in der Objekthierarchie positioniert ist (zb. NewsFolder ¡ Folder ¡ Content).

Schwammig könnte man die Partials für die Inhalte als Templates bezeichnen. Da dieser Begriff allerdings stark vorbelastet und in anderen CMSn anders belegt ist, wird hier davon abgesehen.

4.2 Verzeichnisse

4.2.1 app/

Hier befindet sich der größte Teil des Programmcodes. Die Unterverzeichnisse sind entsprechend den Kompontenten des MVC-Prinzips benannt und aufgeteilt (siehe 1.1.1.

app/controllers/

app/helpers/

app/models/

app/views/

Kapitel 5

Wartung

5.1 Update

Wenn Tirade mithilfe von Subversion installiert wurde (empfohlen), kann die Installation wie folgt aktualisiert werden:

```
cd /var/www/tirade
svn up
rake db:migrate # Datenbank migrieren, falls nötig
sudo /etc/init.d/lighttpd restart
```

5.2 Troubleshooting

5.2.1 Fehler finden

Falls sich Tirade instabil verhält, abstürzt, unerklärliche Fehlermeldungen bringt oder einfach 'nicht geht', hilft meist ein Blick in die Logdateien weiter.

Diese befinden sich im RAILSROOT unter 'logs', wobei die 'production.log' die hilfreichste sein dürfte.

5.2.2 Fehler berichten

Falls man als Admin selbst keine Lösung sieht, kann im trac (dem Quellecode- und Bugverwaltungssystem) ein Ticket erstellt werden, welches so schnell wie möglich bearbeitet wird. Der Text ist kurz, aber genau zu formulieren. Dabei helfen folgende Punkte:

- Was wurde versucht zu tun?
- Was war zu erwarten?
- Was ist dagegen passiert?
- Läßt sich der Fehler reproduzieren?
- Entsprechende Logeinträge bitte anhängen.

Neues Ticket: `http://trac.lanpartei.de/trac/tirade/newticket`

Literaturverzeichnis

[RoRHome] Ruby on Rails Homepage `http://rubyonrails.org/` [03.02.2007]

[AgileRails] gile Web Development with Rails (Second Edtion)