

Rake – das moderne Build-Tool

Nikolaus Schüler

Rake ist ein modernes Build-Tool, das Ihnen die Arbeit erleichtert und beim Erstellen sowohl einfacher als auch komplexer Softwarepakete hilft. Es besticht durch seine Einfachheit, Übersichtlichkeit und Konsistenz, ist aber trotzdem allen Aufgaben gewachsen, die Ihnen im Entwickleralltag begegnen.

In diesem TecFeed lernen Sie, wie man Rake einsetzt, von den Grundlagen bis zu ausgefalleneren Tipps, Tricks und Kniffen. Nach der Lektüre dieses TecFeed wissen Sie, wie Sie mit Rake Ihren Build-Prozess besser in den Griff bekommen und Zeit gewinnen, sich ums eigentliche Entwickeln zu kümmern.

O'REILLY®

INHALT

Einleitung | 2

Installation | 3

Grundlagen | 6

Die Elemente eines Rakefile | 10

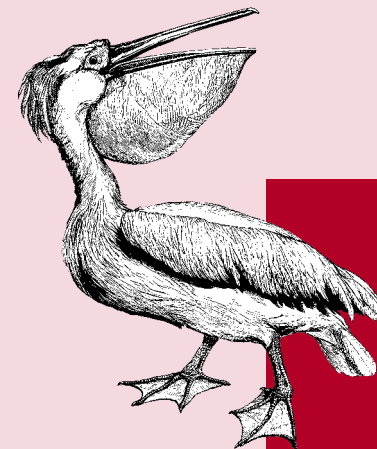
Fortgeschrittenes | 21

Rake und der Rest der Welt | 24

Fibonacci als Beispiel | 26

Ressourcen | 47

Über den Autor | 49



TecFeeds

www.tecfeeds.de

Einleitung

Was ist Rake?

Rake ist ein Build-Tool, also ein Programm, das das automatisierte Erstellen und Installieren von Software unterstützt. Rake ist eine Erweiterung der Skriptsprache Ruby. Das älteste und am weitesten verbreitete Werkzeug zum Handhaben des Build-Prozesses ist das altherwürdige Make, an das sich Rake auch von der Namensgebung anlehnt. (Das »R« hat Rake von Ruby: So wie es Mode ist, die Namen von in oder für Java geschriebenen Anwendungen mit »J« beginnen zu lassen – wie etwa *JAlbum* oder *JBuilder* –, lässt man Namen von Ruby-Software gern mit »R« anfangen, neben Rake z.B. auch *Rant*, das wir später noch kurz kennenlernen).

Die grundlegenden Konzepte sind bei all diesen Programmen die gleichen: Sie arbeiten eine Reihe von Aufgaben (auch *Tasks* oder *Targets* genannt) ab, die ausgeführt werden, um eine Gesamtaufgabe zu erledigen, etwa das Erstellen eines Programms aus seinen Quelltexten oder das Zusammenbauen aller HTML-Seiten für einen Internetauftritt. Teilaufgaben sind dabei zum Beispiel das Übersetzen der Quelltexte in Objektdateien und das Linken dieser Objektdateien mit anderen Bibliotheken, die das Programm benötigt.

Dabei wird überprüft, ob eine (Teil-)Aufgabe bereits erledigt ist, und nur wenn das noch nicht der Fall ist, wird sie überhaupt ausgeführt. Um festzustellen, ob eine Aufgabe erledigt ist, vergleicht man in den meisten Fällen die Zeit und das Datum der entsprechenden Dateien. (Manche Programme

nehmen auch andere Methoden zu Hilfe, zum Beispiel MD5-Signaturen, die sich bei jeder Änderung einer Datei mitändern, so dass sie als »Fingerabdruck« dienen können.) Sind die Dateien, aus denen etwas erstellt werden soll, neuer als das Ergebnis, muss man etwas tun; sind sie das nicht, hat es keinen Sinn, die entsprechende Aufgabe auszuführen, da das Ergebnis auf dem neuesten Stand ist und ein erneutes Ausführen nichts ändern würde.

Rake entnimmt seine Anweisungen genau wie Make einer Datei, sie heißt *Rakefile*, in Anlehnung an Makes *Makefile*. Diese Datei kann auch noch andere Namen haben, wie wir später sehen werden, aber die Form *Rakefile* ist die gebräuchlichste.

Warum Rake?

Rake ist einfacher und übersichtlicher als Make. Manche sagen, Make sei »broken by Design«, weil hier zwei Konzepte in einer Datei gemischt sind: Angaben über Abhängigkeiten einerseits und die Ausführung von Kommandos durch eingebaute Shell-Skripten andererseits. Das macht die eingebetteten Shell-Kommandos schwer zu testen, weil sie nur im Kontext des Makefile und nicht alleine laufen. Ein weiteres Problem von Make (zumindest der ursprünglichen Versionen) ist die Einrückung. In einem Makefile müssen die Targets linksbündig stehen, die Kommandos sind demgegenüber um einen Tab eingerückt. Schon viele Entwickler sind verzweifelt, weil ihre Makefiles ganz normal aussahen, sie (oder jemand anders) aber statt Tabulatoren Leerzeichen zum Einrücken verwendet hatten. Obwohl so ein Makefile aussieht, als sei es gültig,

ergibt sein Aufruf nur Fehlermeldungen, weil Make auf dem Tabulator besteht. Syntaxfehler, die auf unsichtbaren Zeichen beruhen, können auch den hartgesottensten Entwickler in den Wahnsinn treiben.

Ein weiterer Vorteil von Rake: Man hat immer eine komplette Skriptsprache zur Hand, innerhalb eines Rakefile kann man beliebige Ruby-Anweisungen verwenden. Das ermöglicht es, Konstrukte zu verwenden, die in Make so nicht zur Verfügung stehen, etwa Schleifen, die eine ganze Reihe ähnlicher Task-Anweisungen automatisch erstellen. Beispiele dafür werden wir später kennenlernen.

Der Autor und Projektverantwortliche von Rake ist Jim Weirich. Rake ist beliebt in der Extreme Programming-Gemeinde, und so ist es kein Wunder, dass auch Martin Fowler ein Fan von Rake ist (siehe Martin Fowlers Artikel über Rake: <http://martinfowler.com/articles/rake.html>).

Für wen ist dieser Text gedacht?

Dieses TecFeed ist an alle gerichtet, die sich für moderne Skriptsprachen interessieren und ein Build-Werkzeug für ihre täglichen Programmier- und Verwaltungsaufgaben suchen (und das vielleicht bis jetzt mit Make erledigen, aber damit nicht richtig glücklich werden). Vorausgesetzt werden grundlegende Programmierkenntnisse und ein bisschen Erfahrung mit Skriptsprachen – wenn Sie beispielsweise Perl oder Python kennen, tun Sie sich mit Ruby bestimmt nicht schwer. Ruby brauchen sie aber nicht zu kennen, denn alles, was wir an Ruby brauchen, wird in diesem Text erklärt.

Voraussetzungen

Die Beispiele in diesem Text habe ich unter Linux ausprobiert (unter Ubuntu-Linux, um genau zu sein). Sie sollten genauso unter Windows und Mac OS X laufen, wobei man natürlich unter Windows Aufrufe auf der Kommandozeile etwas anpassen muss. Unter Mac OS X hat man eine Unix-Umgebung zur Verfügung, so dass keine Änderungen nötig sein sollten.

Installation

Ruby installieren

Natürlich sollte auf dem System auch Ruby vorhanden sein, das die Grundvoraussetzung für Rake ist. Ob Ruby installiert ist, können Sie feststellen, in dem Sie Folgendes auf einer Kommandozeile eingeben:

```
ruby -v
```

Ist Ruby installiert, bekommen sie eine Ausgabe wie diese:

```
ruby 1.8.6 (2007-06-07 patchlevel 36) [i486-linux]
```

Auf diese Weise wissen Sie auch gleich, welche Version von Ruby auf Ihrem System installiert ist. Als dieser Text geschrieben wurde, war das Ruby in der Basisversion 1.8; diese Version sollte man für die tägliche Arbeit einsetzen. Ruby 1.9 war noch in Vorbereitung und nur als Betaversion für risikofreudige Pioniere zu empfehlen.

Bei den meisten Linux-Installationen ist es inzwischen automatisch dabei, wenn nicht, muss man über den Paketmanager (z.B. *Synaptic*, *aptitude* oder *apt-get* bei Debian Linux und Ubuntu, *Yast* bei Suse, *rpm* bei Redhat Linux oder *yum* bei Fedora Linux) das Ruby-Paket nachinstallieren.

Für den unwahrscheinlichen Fall, dass die Distribution Ruby nicht enthält, oder wenn man unbedingt die allerneueste, superschnieke Version von Ruby haben will, muss man es aus den Quellen selbst kompilieren. Holen Sie sich diese einfach von der Ruby-Homepage <http://www.ruby-lang.org>. Wer es noch aktueller mag, kann sich auch die tagesaktuellen Ruby-Quellen aus dem Subversion-Repository holen und sie selbst übersetzen.

Rake installieren

INSTALLATION ALS GEM

Der bevorzugte Weg der Installation von Rake ist – wie bei den meisten Ruby-Erweiterungen – die Installation als *Gem*. Das Gem-Programm verwaltet Ruby-Libraries und -Erweiterungen; es lädt die Software von einem Repository (einem Rechner im Netz, der die Software zur Verfügung stellt), prüft dann, ob das entsprechende Paket von anderen Paketen abhängt, installiert diese gegebenenfalls mit und richtet schließlich alles auf dem System ein. Unter Linux steht *gem* bei den gängigen Distributionen als Paket bereit, unter Ubuntu Linux etwa als *rubygems*, hier installieren Sie es mit

```
sudo apt-get install rubygems
```

(Warum Sie hier am Anfang des Befehls ein `sudo` angeben müssen, erfahren Sie gleich, das hat bei *apt-get* dieselben Gründe wie bei *gem*.)

Gem ist das Ruby-Analogon zu CPAN, dem Perl-Softwarearchiv, das einer der Gründe für die Popularität von Perl ist. Libraries müssen nicht manuell aus dem Netz geholt, ausgepackt und dann auch noch installiert werden; stattdessen erledigt ein einziger Befehl all diese Aufgaben vollautomatisch. In unserem Fall sollte es reichen,

```
gem install rake
```

einzugeben. Es kann nötig sein, das Programm als Superuser aufzurufen, weil nur er Schreibzugriff auf den Pfad hat, in dem das Gem installiert wird. Dazu können Sie entweder mit `su` in die Rolle des Superusers schlüpfen und das Programm dann ausführen, oder Sie stellen dem Programm den Befehl `sudo` voran, dann wird aus dem obigen Kommando beispielsweise

```
sudo gem install rake
```

Handelt es sich um Ihren eigenen Rechner, sollten Sie das Passwort für den Superuser eigentlich kennen, handelt es sich um einen Rechner, der von anderen verwaltet wird, müssen Sie den Superuser bitten, das Paket für Sie zu installieren.

Der Webserver, von dem das Gem heruntergeladen werden soll, ist dem *gem*-Programm hier schon bekannt. Bei weniger bekannten Gems, die der Autor nur auf seiner eigenen Homepage zur Verfügung stellt, muss man den Server von Hand

angeben. Das geschieht mit der Option `--source`, mit der eine URL angegeben wird, was zum Beispiel so aussehen könnte:

```
gem --source http://gems.example.com install rake
```

Es ist auch möglich, mit `gem` ein Paket zu installieren, das bereits auf den Rechner heruntergeladen wurde, dazu benutzt man

```
gem install <Paketname>
```

Der Paketname ist hier der vollständige Dateiname, einschließlich der Endung `.gem`

Mehr Informationen zu `gem` bietet die Manpage, die Sie mit `man gem`

aufrufen. Zudem ist das Gem-Programm selbstdokumentierend. Sie bekommen Hilfestellung, indem Sie das Programm so aufrufen:

```
gem help
```

Hilfestellung zu speziellen Themen gibt es beispielsweise mit

```
gem help install
```

Damit erfahren Sie (fast) alles, was Sie zum Thema Installation wissen müssen.

Es kann sein, dass der Pfad, unter dem Gems abgelegt werden, nicht im Library-Pfad von Ruby enthalten ist. In diesem Fall muss man ihn entweder hinzufügen; unter Unix geschieht das üblicherweise, indem man ihn an die Umgebungsvariable `RUBYLIB` anhängt oder am Anfang eines jeden Programms, in dem das Gem verwendet wird, ein zusätzliches `require 'ruby-gems'` angibt.

INSTALLATION ALS LINUX-PAKET ODER AUS DEN QUELLEN

Die Installation als Gem bietet die Gewähr, stets ein aktuelles Paket zu bekommen. Falls das nicht so wichtig ist, bleibt natürlich die Installation als Paket der jeweiligen Linux-Distribution. Alternativ kann man Rake auch von Hand installieren, indem man es herunterlädt (etwa unter <https://rubyforge.org/projects/rake/>) und dann von Hand installiert, indem man das Paket auspackt, in das ausgepackte Verzeichnis wechselt und dann `ruby install.rb` aufruft. Lesen Sie aber auf jeden Fall die Datei `README` im entpackten Rake-Paket.

Erweiterungen für Editoren und IDEs

Arbeitet man mit Emacs als Editor, empfiehlt es sich, noch das Paket `Elisp-Ruby` zu installieren, das einen Ruby-Mode für Emacs bietet. Alternativ kann man Ruby- und Rake-Skripten natürlich mit jedem beliebigen Texteditor bearbeiten. Wer es ganz luxuriös mag, sollte Eclipse mitsamt den entsprechenden Plugins verwenden oder `NRubyIDE`, eine IDE, die auf Netbeans basiert. Ab Netbeans 6.0 ist der Ruby-Support eingebaut, so dass man sich die separate Entwicklungsumgebung sparen kann und direkt Netbeans einsetzen kann.

Webadressen zu Ruby, Gems und den IDEs finden Sie am Schluss dieses Textes, aber *eine* sollte man auf jeden Fall kennen: Die Rake-Homepage unter <http://rake.rubyforge.org>.

Grundlagen

Ein bisschen Ruby

Bevor wir zu Rake selbst kommen, müssen wir kurz über Ruby sprechen. Ruby ist leicht verdaulich, aber es gibt ein paar Dinge, die Sie wissen sollten, damit es nicht zu unliebsamen Überraschungen kommt. Alles, was Sie an Ruby können müssen, um ein grundlegendes Rakefile zu schreiben, lernen Sie in diesem Text. Sollten Sie gewagtere Dinge in Ihren Rakefiles machen wollen oder gar Gefallen an Ruby gefunden haben und ein ernsthafter Ruby-Programmierer werden wollen, empfehle ich Ihnen die weiterführenden Texte und Links im Abschnitt »Ressourcen«, besonders aber das »Pickaxe Book«, mit dem Sie ziemlich weit kommen dürften.

VARIABLEN

Variablen bekommen in Ruby – wie in den meisten anderen Programmiersprachen auch – einen Wert zugewiesen, indem man links von einem Gleichheitszeichen den Namen der Variablen angibt, rechts davon den Wert. Da Rakefiles ja Ruby-Code sind, findet dort die Zuweisung von Werten an Variablen auf dieselbe Weise statt.

BLOCKS UND CLOSURES

Blocks sind ein in Ruby häufig verwendetes Stilmittel. Technisch gesehen sind sie *Closures* (wir werden gleich sehen, was das ist), und die gab es schon bei Lisp und Smalltalk, die erklärtermaßen zu den Vorbildern für Ruby zählen.

In Ruby ist ein Block alles, was sich zwischen einem `do` und einem `end` beziehungsweise einem `{` und einem `}` befindet. Solch ein Block ist im Prinzip eine Funktion, die als Argument an eine andere Funktion übergeben werden kann. Meist geschieht das in Form eines Blocks, weil man die Funktion nur ein Mal braucht, es handelt sich also um eine anonyme Funktion. Man kann aber auch eine echte Funktion als Argument an eine andere Funktion übergeben, dazu verwendet man ein Proc-Object. Wenn man aus einem Block etwas an die aufrufende Umgebung zurückgeben will, verwendet man die Funktion `yield()`. Im Beispiel, das im Abschnitt »Fibonacci« vorgestellt wird, machen wir von dieser Funktion Gebrauch – wer sich für Blocks und `yield()` interessiert, kann ja schon mal vorblättern.

Blocks sieht man in Rakefiles dauernd, da das, was in einem Task ausgeführt wird, durch einen Block beschrieben wird.

Das Interessante daran ist, dass ein Block eine Closure darstellt. In einer Closure kann auf alle Variablen zugegriffen werden kann, die gültig waren, als die Closure definiert wurde, in unserem Fall also auf alle Variablen, die vorher im Rakefile definiert wurden (und das sind ja ungefähr die Variablen, die wir so brauchen).