6 Rails erweitern

- Wie kann Ruby on Rails an eigene Anforderungen angepasst werden?
- Was steckt hinter Railties?
- Wie können Generatoren entwickelt werden?
- Was sind die Neuerungen an den Engines in Rails 3.1?

Auch wenn Ruby on Rails eine große Anzahl an Hilfswerkzeugen mitbringt, so gibt es hin und wieder individuelle Anforderungen, an die ein Framework angepasst werden muss. Beispiele hierfür sind die Erstellung von eigenen Generatoren oder das Modifizieren des Boot-Prozesses.

Es gibt verschiedene Möglichkeiten, eine Rails-Anwendung anzupassen, die sich mit Erscheinen der Versionen 3.0 und 3.1 stark verbessert haben. Von der Nutzung einer speziellen Plug-in-Schnittstelle über die Integration von kompletten Drittapplikationen bis zur Implementierung eigener ORM-Frameworks: Durch Rails 3.1 stehen Entwicklern leistungsstarke Werkzeuge zur Seite, mit denen auch komplizierte Anwendungsfälle problemlos gemeistert werden können.

In diesem Kapitel führen wir in die Nutzung der neuen Komponenten ein und zeigen anhand des Beispielprojekts, wie diese in bestehende Anwendungen integriert werden können.

6.1 Eigene Models mit ActiveModel erstellen

Mit dem Release von Ruby on Rails 3.0 wurden Funktionalitäten wie Validierung, Serialisierung, Unterstützung des Observer-Patterns und Callbacks von ActiveRecord in das neue Gem ActiveModel ausgelagert. Damit stehen sie auch anderen Model-Frameworks zur Verfügung. Erste Frameworks, beispielsweise Mongoid oder Ripple, setzen auf ActiveModel.

Somit ist es mit Rails 3 leichter denn je, eigene Model(-Frameworks) zu erstellen. Gerade in Verbindung mit den vielen neuen NoSQL-Datenbanken ist es nützlich, auf die Dienste von ActiveModel

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

zurückgreifen zu können, wenn noch keine fertigen Ruby-Bibliotheken existieren oder bestehende Lösungen nicht verwendet werden können.

Aber auch für die Implementierung von Models, die nicht in einer Datenbank gespeichert werden und somit auf ein ORM-Framework verzichten können, bietet sich das Gem an.

Dieses Kapitel zeigt, wie mit ActiveModel ein selbst erstelltes Model validiert und serialisiert werden kann.

Als fortlaufendes Beispiel dient ein Model, das die Daten eines Kontaktformulars speichert. Das Kontaktformular soll validiert werden, schließlich möchte man nur vollständige Datensätze aufnehmen. Zusätzlich werden die Daten nicht in einer Datenbank, sondern in einer lokalen Datei im JSON-Format gespeichert, wozu eine Serialisierung benötigt wird.

Wir legen zunächst die Datei für das Model unter `app/models/contact_form.rb` an:

Listing 6.1 Initiale Definition des Models ContactForm

```
class ContactForm
end
```

Ein Kontaktformular besteht neben einer Nachricht aus weiteren Daten, die der Benutzer angeben muss: Name und E-Mail-Adresse. Im Model werden diese Attribute wie folgt definiert:

Listing 6.2 Definition der Attribute

```
class ContactForm
  attr_accessor :name, :email, :message
end
```

Nachdem das Grundgerüst steht, widmen wir uns im nächsten Schritt der Validierung.

6.1.1 Validierung

Selbstverständlich sollen keine leeren Kontaktformulare abgeschickt werden, um Spam und unvollständige Datensätze zu vermeiden. Es müssen stets alle Attribute des Models gesetzt sein.

Darüber hinaus muss eine korrekte E-Mail-Adresse angegeben werden und die Nachricht soll aus mindestens 50 Zeichen bestehen.

Für das Spezifizieren und Testen dieser Anforderungen verwenden wir RSpec. Da noch keine Testdatei für das Model existiert, legen wir sie zunächst unter `spec/models/contact_form_spec.rb` an und füllen sie mit Tests:

Listing 6.3
Tests des Models ContactForm

```
require "spec_helper"

describe ContactForm do
  before(:each) do
    @contact_form = ContactForm.new
  end

  context "given a valid contact form" do
    it "has a name" do
      @contact_form.name = "David"
      @contact_form.should have(:no).errors_on(:name)
    end

    it "has an email that is valid" do
      @contact_form.email = "david@37signals.com"
      @contact_form.should have(:no).errors_on(:email)
    end

    it "has a message that has a length of ↩
                              minimum 50 characters" do
      @contact_form.message = "Hello"*10
      @contact_form.should have(:no).errors_on(:message)
    end
  end

  context "given an invalid contact form" do
    it "has no name" do
      @contact_form.name = ""
      @contact_form.should have(1).error_on(:name)
    end

    it "has no email" do
      @contact_form.email = ""
      @contact_form.should have(2).errors_on(:email)
    end

    it "has an email that is invalid" do
      @contact_form.email = "david@.com"
      @contact_form.should have(1).error_on(:email)
    end

    it "has no message" do
      @contact_form.message = ""
      @contact_form.should have(2).errors_on(:message)
    end
```

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

```
    it "has a message that has a length of ⟵
                              less than 50 characters" do
      @contact_form.message = "Hello!"
      @contact_form.should have(1).error_on(:message)
    end
  end
end
```

Für die Definition der Tests können die bekannten Werkzeuge verwendet werden, die überprüfen, ob Fehler bezüglich eines bestimmten Attributs vorliegen. Lässt man die Tests jetzt durchlaufen, werden diese natürlich fehlschlagen, da die Validierungen noch gar nicht implementiert sind.

Hierfür erweitern wir das Model um das von ActiveModel mitgelieferte Modul Validations:

Listing 6.4 *Einbinden des Validations-Moduls*

```
class ContactForm
  include ActiveModel::Validations

  attr_accessor :name, :email, :message
end
```

Dadurch stehen die gewohnten Möglichkeiten zur Definition von Validatoren zur Verfügung. Zunächst definieren wir, dass alle Attribute des Models gesetzt sein müssen:

```
class ContactForm
  include ActiveModel::Validations

  attr_accessor :name, :email, :message

  validates_presence_of :name, :email, :message
end
```

Bei erneutem Aufruf der Tests sieht man bereits, dass einzelne Tests erfolgreich durchlaufen. Lediglich Testfälle, die die Länge der Nachricht und das Format der E-Mail-Adresse überprüfen, schlagen noch fehl, da wir dafür noch keine Validierungsregeln definiert haben.

Ausführen eines einzelnen Tests

Einzelne Tests können auch isoliert ausgeführt werden. Das ist dann hilfreich, wenn man nur an einer Teilfunktionalität arbeitet und nicht alle Tests des Projekts ausführen möchte. Dazu wird die Testdatei dem Helfer von RSpec auf der Kommandozeile übergeben, in unserem Fall:

```
bundle exec rspec spec/models/contact_form_spec.rb
```

```
2. Shell
Shell  ⌘1
stefansmac:rails3buchapp stefan$ bundle exec rspec spec/models/contact_form_spec.rb
....FFFF

Failures:

  1) ContactForm given an invalid contact form has no email
     Failure/Error: @contact_form.should have(2).error_on(:email)
       expected 2 error on :email, got 1
     # ./spec/models/contact_form_spec.rb:33:in `block (3 levels) in <top (required)>'

  2) ContactForm given an invalid contact form has an email that is invalid
     Failure/Error: @contact_form.should have(1).error_on(:email)
       expected 1 error on :email, got 0
     # ./spec/models/contact_form_spec.rb:38:in `block (3 levels) in <top (required)>'

  3) ContactForm given an invalid contact form has no message
     Failure/Error: @contact_form.should have(2).errors_on(:message)
       expected 2 errors on :message, got 1
     # ./spec/models/contact_form_spec.rb:43:in `block (3 levels) in <top (required)>'

  4) ContactForm given an invalid contact form has a message that has a length of less than 100 characters
     Failure/Error: @contact_form.should have(1).error_on(:message)
       expected 1 error on :message, got 0
     # ./spec/models/contact_form_spec.rb:48:in `block (3 levels) in <top (required)>'

Finished in 0.05359 seconds
8 examples, 4 failures
stefansmac:rails3buchapp stefan$
```

Abbildung 6-1
Zwischenstand der Implementierung

Somit verbleiben noch die restlichen Validierungsregeln. Das Format der E-Mail-Adresse soll mit einem regulären Ausdruck auf Korrektheit überprüft werden, hierzu ist die Methode `validates_format_of` zu verwenden:

```
class ContactForm
  include ActiveModel::Validations

  attr_accessor :name, :email, :message

  validates_presence_of :name, :email, :message
  validates_format_of :email,
      :with => /\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b/i
end
```

Der reguläre Ausdruck überprüft, ob die E-Mail-Adresse:

- im lokalen Teil (der Bereich vor dem @) und im Host keine Leerzeichen oder @ beinhaltet,
- ein @ zur Trennung von lokalem Teil und Host enthält,
- einen . als Trennzeichen zwischen Host und Top-Level-Domain besitzt sowie
- eine Top-Level-Domain aus zwei bis vier Zeichen enthält.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Die Regel, dass die Nachricht aus mindestens 50 Zeichen bestehen muss, wird durch `validates_length_of` spezifiziert:

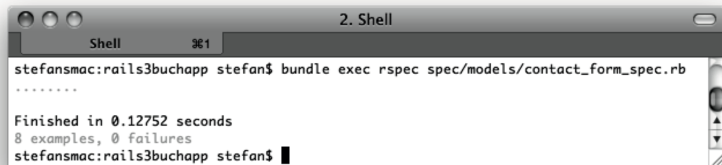
```
class ContactForm
  include ActiveModel::Validations

  attr_accessor :name, :email, :message

  validates_presence_of :name, :email, :message
  validates_format_of :email,
     :with => /\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b/i
  validates_length_of :message, :minimum => 50
end
```

Nun werden alle initial spezifizierten Anforderungen erfüllt, wie beim erneuten Ausführen der Tests zu sehen ist. Wir konnten mit wenigen Zeilen Ruby-Code und ohne die Validatoren manuell implementieren zu müssen, das selbst erstellte Model mit der gewünschten Funktionalität ausstatten.

Abbildung 6-2
Alle Tests werden erfüllt



6.1.2 Serialisierung

In der Einleitung des Kapitels haben wir geschrieben, dass die Daten eines Kontaktformulars, insofern es korrekt ausgefüllt wurde, im JSON-Format in einer Datei gespeichert werden. Dazu muss das Model in das vorgeschriebene Format überführt werden.

Hierfür kann man zum einen die Serialisierung selbst implementieren, was sehr aufwendig ist. Zum anderen kann man aber auch auf die Dienste von ActiveModel zurückgreifen, das ein entsprechendes Modul bereitstellt.

In den Tests spezifizieren wir lediglich, dass das Model die Methode `to_json` besitzt, die nach Konvention das Objekt im JSON-Format zurückliefert. Die eigentliche Logik wird von ActiveModel implementiert und auch dort getestet.

Listing 6.5
Erweiterung der Tests

```
require "spec_helper"

describe ContactForm do
  before(:each) do
    @contact_form = ContactForm.new
  end

  ...

  describe "#to_json" do
    it "exists" do
      @contact_form.should respond_to(:to_json)
    end
  end
end
```

Um das Model zu JSON serialisieren zu können, musss das Modul `ActiveModel::Serializers::JSON` eingebunden werden. Zusätzlich muss eine Methode mit dem Namen `attributes` implementiert werden, die alle Attribute, die serialisiert werden sollen, als Ruby-Hash zurückliefert:

Listing 6.6
Konfiguration der Serialisierung

```
class ContactForm
  include ActiveModel::Serializers::JSON
  include ActiveModel::Validations

  attr_accessor :name, :email, :message

  validates_presence_of :name, :email, :message
  validates_format_of :email,
     :with => /\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b/i
  validates_length_of :message, :minimum => 50

  def attributes
    { :name => @name, :email => @email, :message => @message }
  end
end
```

Durch das Einbinden des Moduls wird die Methode `to_json` bereitgestellt, die eine entsprechende Repräsentation der Attribute zurückliefert.

Soll ein Model in das Format XML serialisiert werden, kann dies analog, durch Einbinden von `ActiveModel::Serializers::Xml`, erreicht werden.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

6.1.3 Verwendung innerhalb der Anwendung

Innerhalb der Anwendung, beispielsweise in der Controller-Schicht, kann das Model ContactForm wie jedes andere Model verwendet werden. Was jetzt noch fehlt, ist eine Möglichkeit, um Attribute beim Instanziieren des Models zu füllen. Zudem sollte eine Methode existieren, die das Model in einer Datei, die als Namen den aktuellen Zeitstempel trägt, serialisiert abspeichert.
Wir spezifizieren zunächst die verbleibenden Anforderungen:

```
require "spec_helper" do

describe ContactForm do
  before(:each) do
    @contact_form = ContactForm.new(:name => "David",
                              :email => "david@37signals.com",
                              :message => "Hello"*20)
  end

  ...

  describe "#initialize" do
    it "sets the name" do
      @contact_form.name.should eql("David")
    end

    it "sets the email" do
      @contact_form.email.should eql("david@37signals.com")
    end

    it "sets the message" do
      @contact_form.message.should eql("Hello"*20)
    end
  end

  describe "#save" do
    it "invokes the serialization" do
      @contact_form.should_receive(:to_json)
      @contact_form.save
    end

    it "creates a file" do
      File.should_receive(:open)
      @contact_form.save
    end
  end
end
```

Für die komfortable Initialisierung der Attribute erweitern wir die Klasse um einen Konstruktor, der die Attributswerte als Hash übergeben bekommt und verarbeitet.

Die Instanzmethode save stößt die Serialisierung an und speichert das Ergebnis im Verzeichnis contact_forms. Wir legen den Ordner auf der höchsten Ebene unserer Anwendung an und erweitern das Model:

```
class ContactForm
  include ActiveModel::Serializers::JSON
  include ActiveModel::Validations

  attr_accessor :name, :email, :message

  validates_presence_of :name, :email, :message
  validates_format_of :email,
     :with => /\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b/i
  validates_length_of :message, :minimum => 50

  def initialize(attributes)
    @name = attributes[:name]
    @email = attributes[:email]
    @message = attributes[:message]
  end

  def attributes
    { :name => @name, :email => @email, :message => @message }
  end

  def save
    file_name = File.join(Rails.root,
                       "contact_forms/#{Time.now.to_i}.json")
    File.open(file_name, "w") { |f| f.write(to_json) }
  end
end
```

Von ActiveRecord ist man gewohnt, dass nur valide Datensätze abgespeichert werden – so implementieren wir auch unser Model. Schließlich sollen sich andere Entwickler möglichst schnell einarbeiten können.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Falls das Model valide ist, liefert die Methode `save` den Wert `true` zurück, andernfalls `false`. Spezifizieren wir das zunächst in den Tests:

Listing 6.7
Erweiterung der Tests der save-Methode

```
require "spec_helper" do

describe ContactForm do
  before(:each) do
    @contact_form = ContactForm.new(:name => "David",
                                    :email => "david@37signals.com",
                                    :message => "Hello"*20)
  end

  ...

  describe "#save" do
    context "given a valid contact form" do
      before(:each) do
        @contact_form.stub!(:valid?).and_return(true)
      end

      it "invokes the serialization" do
        @contact_form.should_receive(:to_json)
        @contact_form.save
      end

      it "creates a file" do
        File.should_receive(:open)
        @contact_form.save
      end

      it "returns true" do
        @contact_form.save.should be_true
      end
    end

    context "given an invalid contact form" do
      before(:each) do
        @contact_form.stub!(:valid?).and_return(false)
      end

      it "returns false" do
        @contact_form.save.should be_false
      end
    end
  end
end
```

Die Implementierung dieses Verhaltens sieht wie folgt aus: Innerhalb der Methode `save` wird die Methode `valid?` aufgerufen, die von ActiveModel bereitgestellt wird und zurückliefert, ob die Attribute korrekt gesetzt sind. Je nach Ergebnis wird nun das serialisierte Model abgespeichert, zudem wird das Ergebnis der Validierung entsprechend weitergeleitet.

```
class ContactForm
  ...

  def save
    if valid?
      file_name = File.join(Rails.root,
                   "contact_forms/#{Time.now.to_i}.json")
      File.open(file_name, "w") { |f| f.write(to_json) }
      true
    else
      false
    end
  end
end
```

6.1.4 In Formularen verwenden

Nun stehen wir noch vor einer letzten Hürde: Der Rails-Form-Helper `form_for` stellt weitere Anforderungen an unser Model. So muss es beispielsweise über die Methoden `to_key` und `persisted?` verfügen, um mit ihm verwendet werden zu können.

Da die abgesendeten Kontaktanfragen nicht in einer Datenbank gespeichert werden, sind sie auch nicht persistent und besitzen kein Schlüsselattribut.

Wir spezifizieren die Anforderungen mit RSpec:

```
require "spec_helper" do

describe ContactForm do
  before(:each) do
    @contact_form = ContactForm.new(:name => "David",
                            :email => "david@37signals.com",
                            :message => "Hello"*20)
  end

  ...
```

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

```
  describe "#to_key" do
    it "returns nil" do
      @contact_form.to_key.should be_nil
    end
  end

  describe "#persisted?" do
    it "returns false" do
      @contact_form.should_not be_persisted
    end
  end
end
```

Wir müssen lediglich die Methode `persisted?` implementieren, den Rest übernehmen die Module `ActiveModel::Conversion` und `ActiveModel::Naming`:

```
class ContactForm
  include ActiveModel::Conversion
  extend ActiveModel::Naming

  ...

  def persisted?
    false
  end
end
```

Anschließend steht einer Verwendung des ContactForm-Models in Verbindung mit dem Rails-Helper nichts mehr im Weg.

Zusammenfassend kann gesagt werden, dass mithilfe von ActiveModel die Implementierung von eigenen Model(framework)s einfacher denn je ist. Problemlos können einzelne Funktionalitäten, wie die Validierung oder Serialisierung von Models, eingebunden werden. Neben den bereits beschriebenen Komponenten stellt ActiveModel weitere zur Verfügung – mehr Informationen dazu gibt es in der Dokumentation des Gems.

Weitere Informationen

- Dokumentation: `http://rubydoc.info/docs/rails/ActiveModel/`

6.2 Railties

Wie in Kapitel 2.3.4 beschrieben, übernimmt Railties die Verknüpfung der Frameworkkomponenten und bietet Schnittstellen für eigene Anpassungen und Erweiterungen. Mit Version 3.0 erstmals im Einsatz, wird es heute von vielen Bibliotheken verwendet, um beispielsweise eigene Generatoren zur Verfügung zu stellen.

Durch Railties ist es für Rails-Entwickler möglich, das Framework an die eigenen Anforderungen anzupassen:

- Die gesamte Konfiguration einer Rails-Anwendung, die üblicherweise in der Datei `config/application.rb` stattfindet, kann über die Plug-in-Schnittstelle verändert werden.
- Selbst erstellte Rake-Tasks können in das Plug-in ausgelagert und dadurch wiederverwendbar gemacht werden. Bisher mussten diese im Verzeichnis `lib/tasks` verwaltet werden.
- Soll Ruby on Rails durch eigene Generatoren erweitert werden, können diese mit der Railties-Schnittstelle dem Framework bekannt gemacht werden.
- Die Verwaltung von Initializers, also Programmen, die beim Starten einer Rails-Applikation ausgeführt werden, ist mithilfe von Railties möglich. Problemlos können eigene Initializers hinzugefügt werden, zudem kann auch die Ausführreihenfolge bestimmt werden.
- Die Rails-Konsole ist ein mächtiges Werkzeug, das täglich in der Entwicklung eingesetzt wird. Über Railties kann Code hinzugefügt werden, der beim Starten der Konsole ausgeführt wird.

6.2.1 Verwenden der Railties-Schnittstelle

Im Grunde genommen reicht für die Verwendung von Railties die Erstellung einer neuen Klasse, die von `Rails::Railtie` erbt, aus. Durch die Vererbung wird die Schnittstelle in Form von Klassenmethoden bereitgestellt.

Erzeugen wir zunächst die Klasse inklusive der Vererbungsbeziehung:

```
class MyRailtie < ::Rails::Engine
end
```

Innerhalb der Klasse steht nun die Methode `config` zur Verfügung, über die Einstellungen vorgenommen werden können. So lässt sich beispielsweise die Standardsprache für Lokalisierungen ändern:

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Listing 6.8
Änderungen an der Konfiguration

```
class MyRailtie < ::Rails::Engine
  config.i18n.default_locale = :de
end
```

Auch das Hinzufügen eigener Rake-Tasks verläuft ähnlich. Hierfür existiert die Methode `rake_tasks`, die als Parameter einen Ruby-Codeblock erhält, der die Rake-Tasks einbindet:

Listing 6.9
Einbinden eigener Rake-Tasks

```
class MyRailtie < ::Rails::Engine
  rake_tasks do
    require "my_task.rake"
  end
end
```

Analog dazu verläuft das Erweitern der Rails-Anwendung um eigene Generatoren. Der Methode `generators` wird ein Codeblock übergeben, in dem Generatoren eingebunden werden:

Listing 6.10
Einbinden eigener Generatoren

```
class MyRailtie < ::Rails::Engine
  generators do
    require "my_generator"
  end
end
```

Initializers werden beim Start einer Rails-Anwendung ausgeführt. Üblicherweise können hier Konfigurationen vorgenommen werden. Bisher musste für das Anlegen eines Initializers eine Datei im Ordner `config/initializers` erstellt werden, die automatisch vom Framework aufgerufen wird. Durch die Railties-Schnittstelle steht nun eine weitere Möglichkeit zur Verfügung.

Die Methode `initializer` erwartet den Namen des Initializers sowie den auszuführenden Codeblock:

Listing 6.11
Anlegen eines Initializers

```
class MyRailtie < ::Rails::Engine
  initializer "my_engine.initial_configuration" do
    MyRailtie::Configuration.init
  end
end
```

Um die Ausführreihenfolge der Initializers zu beeinflussen, werden die Optionen `:before` und `:after` verwendet:

Listing 6.12
Ausführreihenfolge der Initializers

```
class MyRailtie < ::Rails::Engine
  initializer "my_engine.initial_configuration" do
    MyRailtie::Configuration.init
  end
```

```
    initializer "my_engine.boot",
      :after => "my_engine.initial_configuration" do
        MyRailtie.boot
    end
  end
```

Die Methode `console` hilft beim Andocken an die Rails-Konsole. Ihr wird ein Codeblock übergeben, der beim Starten der Konsole ausgeführt wird. Die folgende Implementierung gibt beim Start der Konsole die aktuelle Uhrzeit aus:

```
class MyRailtie < ::Rails::Engine
  console do
    puts Time.now
  end
end
```

Listing 6.13 *Erweitern der Rails-Konsole*

Die verschiedenen Möglichkeiten können auch miteinander innerhalb einer Klasse kombiniert werden.

6.2.2 Auslagern in ein eigenes RubyGem

Nachdem wir die Verwendung der Railties-Schnittstelle kennengelernt haben, stehen wir vor der nächsten Frage: Wie und wo sollen eigentlich die Klassen in Rails abgelegt werden? Üblicherweise wird der Ordner `lib` als Ort für eigene Klassen und Bibliotheken verwendet, doch das würde in diesem Fall die Vorteile und Stärken von Railties schmälern. Schließlich soll damit eine wiederverwendbare Lösung geschaffen werden, was bei Ablage im `lib`-Verzeichnis nicht der Fall ist.

Nicht erst seit dem Erscheinen von Bundler ist RubyGems eine leistungsstarke Verwaltung von Bibliotheken und häufig benötigten Funktionalitäten in Rails-Applikationen. Plug-ins sollten in ein eigenes Gem ausgelagert werden, um sie in andere Anwendungen einbinden zu können.

Zunächst wechseln wir in das Verzeichnis, in dem auch der Ordner unserer Beispielanwendung liegt. Wir verwenden den Generator von Bundler, um das Gem my_railtie anzulegen:

```
bundle gem my_railtie
```

Listing 6.14 *Anlegen des Gems*

Dadurch wird der Ordner `my_railtie` angelegt, der eine Basisstruktur enthält. Für uns ist vor allem die Datei `lib/my_railtie.rb` interessant, die beim Einbinden des Gems geladen wird. In dieser Datei können wir auf die Railties-Schnittstelle zugreifen. Es muss darauf geachtet werden, dass alle Klassen innerhalb eines eigenen Namensraums (in unserem Fall MyRailtie) abgelegt werden, um Probleme mit anderen Bibliotheken zu vermeiden:

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

```
module MyRailtie
  class Base < ::Rails::Railtie
    console do
      puts Time.now
    end
  end
end
```

Der Einfachheit halber binden wir das erstellte Gem über das Dateisystem ein, sinnvollerweise sollte aber ein Versionskontrollsystem als Quelle dienen. Dazu muss folgender Eintrag im `Gemfile` vorgenommen werden:

Listing 6.15 Einbinden in die Beispielanwendung

```
gem "my_railtie", :path => "../my_railtie"
```

Startet man jetzt die Rails-Konsole mit

```
rails console
```

wird zu Beginn die aktuelle Uhrzeit ausgegeben.
Nachdem das entwickelte Gem in das Versionskontrollsystem aufgenommen wurde, kann es auch von dort eingebunden werden. Dazu ist lediglich die Adresse im `Gemfile` anzugeben, zum Beispiel:

Listing 6.16 Einbinden des Gems über Git

```
gem "my_railtie",↩
    :git => "git@rails-expertenwissen.de:my_railtie.git"
```

Weitere Informationen

- Dokumentation: `http://edgeapi.rubyonrails.org/classes/Rails/Railtie.html`

6.3 Generatoren selbst entwickeln

Im vorangegangenen Kapitel haben wir gesehen, dass durch die Railties-Schnittstelle Generatoren einer Rails-Anwendung hinzugefügt werden können – aber wie entwickelt man diese eigentlich? In diesem Kapitel dreht sich alles darum, wie man mit Rails 3.1 einen eigenen Generator implementiert und verwendet.

Als einführendes Beispiel in das Testen mit RSpec haben wir eine Möglichkeit für applikationsweite Konfigurationen geschaffen. Natürlich möchte niemand eine solche YAML-Datei von Hand anlegen, weshalb wir dafür einen Generator implementieren werden.

6.3.1 Das Grundgerüst erstellen

Zunächst muss das Grundgerüst erstellt werden. Für diese Aufgabe bietet Ruby on Rails einen eigenen Generator an, dem der gewünschte Name, in unserem Fall settings, übergeben wird:

```
bundle exec rails generate generator settings
```

Listing 6.17
Anlegen des Generators

Dieser Befehl legt im Verzeichnis `lib/generators` den Ordner `settings` an, in dem alle den Generator betreffenden Dateien abgelegt werden. Wir öffnen zunächst die Datei `settings_generator.rb`:

```
class SettingsGenerator < Rails::Generators::NamedBase
  source_root File.expand_path('../templates', __FILE__)
end
```

Hier wird der neue Generator definiert, der sein Verhalten von `Rails::Generators::NamedBase` erbt. Zudem wird das Verzeichnis `templates` als Basisverzeichnis für Templates spezifiziert.

6.3.2 Templates

Templates sind Schablonen für Dateien, die durch den Generator angelegt werden. Da wir eine YAML-Datei generieren, legen wir die Datei `settings.yml` im Verzeichnis `lib/generators/settings/templates` an und füllen sie mit folgendem Inhalt:

```
default: &default
  title: "Website title"

development: &development
  <<: *default

test: &test
  <<: *default

production: &production
  <<: *default
```

Listing 6.18
Template für den Generator

Innerhalb der Templates wird die aus Rails-Views bekannte ERb-Syntax verwendet, um Ruby-Code einzufügen. So bleibt das Template generisch und wird je nach Aufruf mit einem anderen Inhalt gefüllt.

Im Generator muss spezifiziert werden, welches Template verwendet werden soll. Zudem muss auch der spätere Ablageort des ausgefüllten Templates, nämlich das Verzeichnis der Konfiguration, `config`, angegeben werden.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Dazu erstellen wir die Methode `copy_config_file`, die die Methode `template` der Railties-Schnittstelle verwendet, um die Konfigurationsdatei an den gewünschten Platz zu kopieren.

Listing 6.19 Kopieren der Konfigurationsdatei

```
class SettingsGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)

  def copy_config_file
    template("settings.yml",↩
             "config/#{file_name}.yml")
  end
end
```

Wir greifen auf die Methode `file_name` zurück, die einen der Rails-Konvention entsprechenden Dateinamen für den übergebenen Namen erstellt. Schließlich soll der Benutzer selbst entscheiden können, wie die Konfigurationsdatei heißt.

Für den Initializer muss ein weiteres Template unter `lib/generators/settings/templates/initializer.rb` abgelegt werden:

Listing 6.20 Initializer der Konfiguration

```
<%= class_name -%>.source_file = ↩
                              "<%= "config/#{file_name}" %>.yml"
<%= class_name -%>.read_configuration
```

Die Methode `copy_initializer` kopiert den Initializer an seinen Platz:

Listing 6.21 Kopieren des Initializers

```
class SettingsGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)

  def copy_config_file
    template("settings.yml",↩
             "config/#{file_name}.yml")
  end

  def copy_initializer
    template("initializer.rb",↩
             "config/initializers/settings.rb")
  end
end
```

Um die Templates abzuschließen fehlt nur noch die Implementierung der Settings-Klasse, die in `lib/generators/settings/templates/settings.rb` abgelegt wird:

Listing 6.22
Implementierung der Settings

```
require "ostruct"
require "yaml"

class <%= class_name %>
  cattr_accessor :source_file
  @@config ||= OpenStruct.new

  class << self
    def read_configuration
      @@config = OpenStruct.new(YAML.load_file(source_file))
      # get only the current environment's settings
      @@config = OpenStruct.new(@@config.__send__(Rails.env))
    end

    def method_missing(method_name, *args, &block)
      @@config.__send__(method_name, *args)
    end
  end
end
```

Das Template wird von der Methode `copy_settings_file` bereitgestellt:

Listing 6.23
Kopieren der Settings-Implementierung

```
class SettingsGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)

  def copy_config_file
    template("settings.yml",↩
             "config/#{ file_name }.yml")
  end

  def copy_initializer
    template("initializer.rb",↩
             "config/initializers/settings.rb")
  end

  def copy_settings_file
    template("settings.rb",↩
             "app/models/#{ file_name }.rb")
  end
end
```

Analog zu `file_name` generiert `class_name` einen der Rails-Konvention entsprechenden Klassennamen.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Benennung der Methoden eines Generators

Sie werden sich vielleicht fragen, nach welcher Konvention die Methoden von Generatoren benannt werden müssen, damit sie durch Rails aufgerufen werden? Im Beispiel wurde die Methode, die das Template ausfüllt und an die korrekte Stelle kopiert, `copy_initializer` genannt.

Beim Aufruf eines Generators ruft Ruby on Rails alle öffentlichen Methoden der Klasse auf, somit besteht keine direkte Namenskonvention. Es sollten lediglich selbstbeschreibende Methodennamen gewählt werden, sodass die dahinterstehenden Aufgaben eindeutig erkennbar sind.

6.3.3 Hooks zu anderen Generatoren

Wenn wir uns an das Standardverhalten von Rails erinnern, stellen wir fest, dass beim Generieren eines Models automatisch die passende Testdatei angelegt wird. Um diese Funktionalität auch in eigenen Generatoren nutzen zu können, stellt Ruby on Rails Hooks zur Verfügung. So können beim Aufruf eines Generators auch andere Generatoren angestoßen werden. Dieses Verhalten wird durch die Methode `hook_for`, die als Parameter die aufzurufenden Generatoren erhält, konfiguriert.

In unserem Beispiel soll automatisch der Generator des Testframeworks aufgerufen werden, um eine Testdatei anzulegen, in der eigene Erweiterungen der Settings-Klasse getestet werden. Die Testdatei wird bei den Models abgelegt:

Listing 6.24 *Verwenden der Generator-Hooks*

```
class SettingsGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)

  def copy_config_file
    template("settings.yml",↩
             "config/#{ file_name }.yml")
  end

  def copy_initializer
    template("initializer.rb",↩
             "config/initializers/settings.rb")
  end

  def copy_settings_file
    template("settings.rb",↩
             "app/models/#{ file_name }.rb")
  end

  hook_for :test_framework, :as => :model
end
```

Übrigens werden die Hooks auch verschachtelt ausgeführt. Wenn ein eingebundener Generator selbst Abhängigkeiten zu anderen Generatoren besitzt, werden diese natürlich auch beachtet.

6.3.4 Verwendung auf der Kommandozeile

Nun steht der Verwendung des Generators nichts mehr im Wege. Um ihn zu testen, wird er wie folgt ausgeführt:

Listing 6.25 *Verwendung des Generators*

```
bundle exec rails generate settings Settings
```

Das Ergebnis steht anschließend in `app/models/settings.rb` und `config/settings.yml` bereit.

Die Benutzung des Generators scheint für uns recht einleuchtend zu sein, doch woher wissen andere Entwickler, wie sie ihn verwenden können? Hierfür steht die Datei `USAGE` im Ordner des Generators bereit: In ihr kann die Verwendung inklusive eines Beispiels beschrieben werden.

Wir öffnen die Datei und fügen die Informationen ein:

Listing 6.26 *Beschreibung des Generators*

```
Description:
    Creates a possibility for defining application-specific
    configurations.

    This generates a model in app/models, an initializer in
    config/initializers, and a configuration file in
    config/name.yml. It invokes the test framework generator.

Example:
    'rails generate settings Settings'

    This will create:
        Model: app/models/settings.rb
        Configuration file: config/settings.yml
        Initializer: config/initializers/settings.rb
```

Die Hilfe zur Verwendung des Generators steht mittels

```
bundle exec rails generate settings
```

oder

```
bundle exec rails generate settings --help
```

den Benutzern zur Verfügung.

6.3.5 In ein Gem auslagern

Der eigentliche Generator ist fertig, wird aber aktuell noch innerhalb der Rails-Anwendung verwaltet. Im vorangegangenen Kapitel konnten

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

wir sehen, dass die Railties-Schnittstelle erlaubt, Ruby on Rails um eigene Generatoren zu erweitern. Es bietet sich an, den Generator in das erstellte Gem auszulagern, um ihn so auch anderen Applikationen zur Verfügung zu stellen.

Zunächst erstellen wir einen Ordner `generators` im Verzeichnis `lib` unseres Gems. In dieses Verzeichnis kopieren wir den Generator mitsamt aller benötigten Dateien, also den Ordner `lib/generators/settings` aus der Rails-Anwendung. Anschließend sollte die Struktur innerhalb des Gems wie in Abbildung 6-3 aussehen.

Nun muss der Generator nur noch im Gem eingebunden werden. Das passiert in der Datei `lib/my_railtie.rb`:

Listing 6.27 *Einbinden des Generators über Railties*

```
module MyRailtie
  class Base < ::Rails::Railtie
    console do
      puts Time.now
    end

    generators do
      require "generators/settings/settings_generator"
    end
  end
end
```

Da der Generator durch das Gem geladen wird, kann das Verzeichnis `lib/generators/settings` aus der Anwendung gelöscht werden.

Abbildung 6-3 *Struktur des Gems*

Name	Änderungsdatum	Größe	Art
Gemfile	20. April 2011 09:57	4 KB	Dokument
lib	7. Mai 2011 13:36	--	Ordner
generators	Heute, 14:34	--	Ordner
settings	Heute, 14:34	--	Ordner
settings_generator.rb	7. Mai 2011 13:24	4 KB	Ruby ...rce File
templates	7. Mai 2011 13:03	--	Ordner
USAGE	6. Mai 2011 13:26	4 KB	Dokument
my_railtie	Heute, 15:40	--	Ordner
settings.rb	20. April 2011 10:03	4 KB	Ruby ...rce File
version.rb	20. April 2011 09:57	4 KB	Ruby ...rce File
my_railtie.rb	20. April 2011 10:03	4 KB	Ruby ...rce File
my_railtie.gemspec	20. April 2011 09:57	4 KB	Dokument
Rakefile	20. April 2011 09:57	4 KB	Dokument

Natürlich ist es keine wartbare Lösung, die gesamte Implementierung der Klasse Settings innerhalb der Anwendung abzulegen. Da der Generator durch ein Gem bereitgestellt wird, wäre es sinnvoller, die Klasse auch dort abzulegen, was wir zum Abschluss vornehmen.

Legen wir zunächst eine Datei mit dem Namen `settings.rb` im Verzeichnis `lib/my_railtie` des Gems an und füllen sie mit der Implementierung:

Listing 6.28
Ablegen der Implementierung

```
require "ostruct"
require "yaml"

module MyRailtie
  class Settings
    cattr_accessor :source_file
    @@config ||= OpenStruct.new

    class << self
      def read_configuration
        @@config = OpenStruct.new(YAML.load_file(source_file))
        # get only the current environment's settings
        @@config = OpenStruct.new(@@config.__send__(Rails.env))
      end

      def method_missing(method_name, *args, &block)
        @@config.__send__(method_name, *args)
      end
    end
  end
end
```

Anschließend muss definiert werden, dass die Implementierung automatisch durch das Gem eingebunden werden soll, was in der Datei `lib/my_railtie.rb` geschieht:

Listing 6.29
Einbinden der Klasse Settings

```
module MyRailtie
  autoload :Settings, "my_railtie/settings"

  class Base < ::Rails::Railtie
    console do
      puts Time.now
    end

    generators do
      require "generators/settings/settings_generator"
    end
  end
end
```

Schon steht die `Settings`-Klasse über das Gem innerhalb der Applikation zur Verfügung. Eine Kleinigkeit müssen wir noch ändern: Schließlich soll nur eine Klasse, die von der Settings-Klasse erbt, und nicht die gesamte Implementierung generiert werden.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Das passen wir im Template `lib/generators/settings/templates/settings.rb` an:

Listing 6.30 Aufräumen des Templates

```
class <%= class_name -%> < ::MyRailtie::Settings
end
```

Damit haben wir eine übersichtliche, wartbare Implementierung erhalten, die auch in anderen Rails-Anwendungen eingesetzt werden kann.

Weitere Informationen

- Rails Guides: `http://edgeguides.rubyonrails.org/generators.html`
- Railscasts: `http://railscasts.com/episodes/216-generators-in-rails-3`

6.4 Rails-Engines

Eine der größten Errungenschaften von Ruby on Rails 3.1 ist die Neuimplementierung der Rails-Engines. Erstmals ist es möglich, ganze Rails-Anwendungen in andere Applikationen zu integrieren, was auch als `Mounten` bekannt ist. Der Hauptteil der Arbeit an den Engines wurde im Rahmen des "Ruby Summer of Code"-Projekts im Jahr 2010 von Piotr Sarnacki erledigt.

Innerhalb der Engines stehen alle Funktionalitäten von Rails zur Verfügung: Datenbankmigrationen, Routen, Assets und viele mehr. Sie können durch einen eigenen Namensraum von der Hauptapplikation getrennt werden.

In diesem Kapitel werden wir näher auf das Arbeiten mit und den Aufbau von Rails-Engines eingehen.

Wir erstellen eine Engine, die den aktuellen Betriebsstatus der Beispielanwendung ausliefert. Falls beispielsweise Probleme am Server vorliegen, können sie auf dieser Seite den Benutzern mitgeteilt werden. Dabei werden die Statusmeldungen in Form einer Liste dargestellt.

Solch eine Statusseite findet sich auf vielen großen Webseiten wie Twitter (`http://status.twitter.com/`) oder GitHub (`http://status.github.com/`). Durch Auslagerung in eine Engine kann die Funktionalität auch in anderen Rails-Anwendungen wiederverwendet werden und muss nicht mehrfach entwickelt werden. Wir werden aus Platzgründen nicht auf die Administration, sondern lediglich auf die Ausgabe der Statusmeldungen eingehen. Zudem legen wir den Fokus auf das Arbeiten mit Engines – wie man Datensätze erstellt, bearbeitet oder löscht, wurde schon bei der Beispielanwendung gezeigt.

6.4.1 Engine anlegen

Im ersten Schritt muss zunächst die Engine im Dateisystem angelegt werden. Für diese Aufgabe existiert ein eigener Generator, dem der gewünschte Engine-Name übergeben wird. Wir rufen ihn auf der höchsten Ebene der Rails-Anwendung auf:

```
rails plugin new ../application_status --full --mountable
```

Der Generator legt eine neue Engine mit dem Namen application_status auf der Ebene der Beispielanwendung und des Railties-Gems an.

Die Option `--full` signalisiert, dass eine komplette Applikationsstruktur inklusive `app`-Verzeichnis angelegt werden soll.

Mit `--mountable` wird erreicht, dass die Engine in einem eigenen Namensraum ausgeführt wird, wodurch sie ohne Seiteneffekte in Rails-Anwendungen eingebunden wird.

Testen von Engines

An sich werden Engines genauso wie gewöhnliche Rails-Anwendungen getestet. Interessant ist allerdings das Verhalten, wenn sie in eine andere Applikation eingebunden werden. Hierfür wird beim Generieren der Engine eine Rails-Dummy-Anwendung angelegt, die die Engine testweise verwendet. Standardmäßig wird die Dummy-Anwendung im Ordner `test/dummy` abgelegt. Wer ein anderes Testframework benutzt oder sie in einem selbstgewählten Verzeichnis ablegen möchte, kann dies über die Option `--dummy-path` beeinflussen. Soll beispielsweise die Dummy-Anwendung im Verzeichnis `spec/dummy` abgelegt werden, wird der Generator wie folgt verwendet:

```
rails plugin new engine_name --dummy-path spec/dummy
```

6.4.2 Aufbau

Wir wechseln in das neu erstellte Verzeichnis und öffnen die generierte Engine. Im Grunde ähnelt der Aufbau sehr stark einer gängigen Rails-Applikation, auch wenn er an einigen Stellen leicht minimiert ist.

Die Datei `application_status.gemspec` ist das Herzstück der Engine: Da Rails-Engines als RubyGems ausgeliefert und in Anwendungen eingebunden werden, werden hier alle Informationen über das Gem und somit die Engine gepflegt. Hinterlegen wir zunächst die wichtigsten Informationen in dieser Datei:

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Listing 6.31 Gemspec der erstellten Engine

```
$:.push File.expand_path("../lib", _ _FILE_ _)

# Maintain your gem's version:
require "application_status/version"

# Describe your gem and declare its dependencies:
Gem::Specification.new do |s|
  s.name = "application_status"
  s.version = ApplicationStatus::VERSION
  s.authors = ["Stefan Sprenger", "Kieran Hayes"]
  s.email = ["info@stefan-sprenger.com"]
  s.homepage = "http://www.rails-expertenwissen.de"
  s.summary = "Provides a status page for your website."
  s.description = "ApplicationStatus is a Rails engine that " +
                  "provides a status page for your website."

  s.files = Dir["{app,config,db,lib}/**/*"] +
            ["MIT-LICENSE", "Rakefile", "README.rdoc"]
  s.test_files = Dir["test/**/*"]

  s.add_dependency "rails", "~> 3.1.0"

  s.add_development_dependency "sqlite3"
end
```

6.4.3 Anwendungsschichten einer Engine

Im nächsten Schritt legen wir das Model, das die Statusmeldungen speichert, über den Rails-Generator an. Es besitzt die Attribute title und message:

Listing 6.32 Anlegen des Status-Models

```
bundle exec rails generate model Status↩
                title:string message:string
```

Komponenten werden automatisch im Namensraum der Engine, in diesem Fall `ApplicationStatus`, erstellt.

Im Model `app/models/application_status/status.rb` definieren wir die Validierungsregeln:

Listing 6.33 Validierungen des Models

```
module ApplicationStatus
  class Status < ActiveRecord::Base
    validates_presence_of :title, :message
  end
end
```

Da hiermit das Datenmodell steht, beschäftigen wir uns nun mit der Verarbeitung von Anfragen. Dazu legen wir einen Controller für die

Auslieferung der Statusmeldungen an:

```
bundle exec rails generate controller Status index
```

Durch den Generator werden übrigens auch gleich die benötigten Routen in der Datei `config/routes.rb` abgelegt. Diese erweitern wir und pflegen eine Route für die Startseite ein:

```
ApplicationStatus::Engine.routes.draw do
  get "status/index"

  root :to => "status#index"
end
```

Listing 6.34
Routen der Engine

Wir öffnen den Controller `app/controllers/application_status/status_controller.rb` und lesen innerhalb der Action `index` alle Statusmeldungen, sortiert nach dem Datum, an dem sie erstellt wurden, aus:

```
module ApplicationStatus
  class StatusController < ApplicationController
    def index
      @status = Status.order("created_at DESC")
    end
  end
end
```

Anschließend wechseln wir in das zugehörige View-Template `app/views/application_status/status/index.html.erb` und greifen auf sie zu:

```
<% @status.each do |status| %>
<div class="status">
  <h4><%= status.title -%></h4>
  <h5><%= status.created_at.strftime("%d.%m.%Y - %H:%M") -%>↩
          </h5>
  <div>
    <%= status.message -%>
  </div>
</div>
<% end %>
```

Nun fehlt nur noch ein Layout. Wir öffnen `app/layouts/application_status/application.html.erb` und passen das Standardlayout an:

```
<!DOCTYPE html>
<html>
<head>
  <title>Current Status</title>
  <%= stylesheet_link_tag "application" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
```

Listing 6.35
Anpassen des Layouts

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

```
</head>
<body class="status">
<div id="header-wrapper">
  <div id="header">
    <h1><%= link_to "Status", root_path %></h1>
    <br class="clear" />
  </div>
</div>
<div id="main">
  <div id="content">
    <%= yield %>
  </div>
</div>
</body>
</html>
```

Beim Generieren des Controllers hat Ruby on Rails auch Assets, JavaScript- und CSS-Dateien, im Verzeichnis `app/assets` angelegt, die von der Engine mitgeliefert werden.

In diesem Fall ignorieren wir sie, da das Aussehen anwendungsspezifisch angepasst werden soll und wir keine Standardwerte benötigen. Passende Assets sind bereits in der Beispielanwendung vorhanden.

Jetzt steht einer Verwendung nichts mehr im Weg. Im kommenden Abschnitt zeigen wir, wie die Rails-Engine in eine Anwendung eingebunden wird.

6.4.4 Einbinden in die Rails-Anwendung

Da Rails-Engines als RubyGems ausgeliefert werden, können sie mit Bundler in Rails-Applikationen eingebunden werden. Um die erstellte Engine in das Beispielprojekt zu integrieren, genügt es, das `Gemfile` zu öffnen und dort die Verbindung zum lokalen Projektverzeichnis herzustellen:

Listing 6.36 *Einbinden der Engine*

```
gem "application_status", :path => "../application_status"
```

Danach wird die Abhängigkeit mittels

```
bundle install
```

installiert und steht zur Verwendung bereit.

Wir kopieren zunächst die Migrationen der Engine in die Anwendung:

```
bundle exec rake application_status:install:migrations
```

Anschließend führen wir die Migrationen aus:

```
bundle exec rake db:migrate
```

Da noch eine Administrationsoberfläche zum Verwalten der Statusmeldungen fehlt, öffnen wir stattdessen die Rails-Konsole mit

```
bundle exec rails console
```

und erstellen einen Datensatz:

```
ApplicationStatus::Status.create(:title => "Wartungsarbeiten",↩
    :message => "Am 31. November führen wir Wartungsarbeiten ↩
                durch.")
```

In den Routen wird spezifiziert, über welche URL die Engine verfügbar sein soll. Wir öffnen die Datei `config/routes.rb` und pflegen die folgende Route ein:

Listing 6.37
Mounten der Engine

```
Beispielanwendung::Application.routes.draw do
  ...

  mount ApplicationStatus::Engine => "/status"
end
```

Dadurch ist die Engine mitsamt ihren Routen unter dem Pfad `/status` verfügbar. Starten wir den Server neu, können wir das Ergebnis nach Aufruf von `http://localhost:3000/status` betrachten.

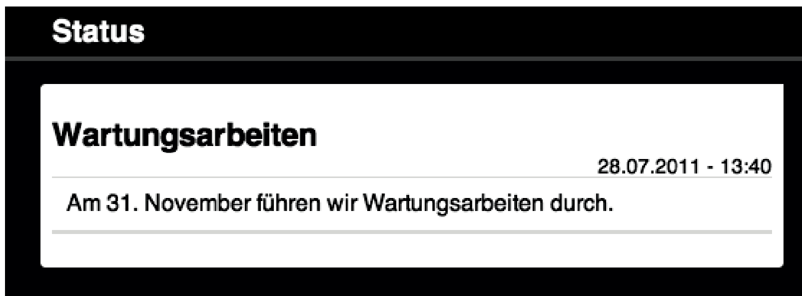


Abbildung 6-4
Die Statusmeldung in der Anwendung

6.4.5 Fazit

Mit der Überarbeitung der Engines schafft Ruby on Rails 3 völlig neue Möglichkeiten, um wiederverwendbare Komponenten zu erstellen. Durch die Einführung der Mountable Apps können komplette Rails-Anwendungen in andere integriert werden. Dabei stehen sämtliche Werkzeuge, die man bereits vom Webframework kennt, zur Verfügung.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

In diesem Kapitel haben wir nur ein kleines Beispiel betrachtet. Engines lassen sich aber auch in anderen Bereichen einsetzen: von Blogs über Content-Management-Systemen bis hin zu Administrationsoberflächen.

Weitere Informationen

- Geschichte der Rails-Engines:
  `http://piotrsarnacki.com/2010/09/14/mountable-engines/`