7363 - Web-basierte Anwendungen
4750 – Web-Engineering

Eine Vertiefungsveranstaltung

Modellbildung

Grundlagen: Das Modul **ActiveSupport**

ActiveRecord – Das ORM-Modul von Rails

Migrationen

ActiveRecord

Das ORM-Modul von Rails

ORM: *Object Relational Mapping*

- **Konzept**
  - Klasse ←→ Tabelle
  - Exemplar ←→ Zeile
  - Methode (Getter/Setter) ←→ Spalte
  - Zuordnungen erfolgen durch ActiveRecord
    - **Automatisch dank sinnvoller Defaults**
    - Und/oder: explizit zu konfigurieren

- **ActiveRecord** – eine ORM-Implementierung für Ruby
  - Quelle: http://rubyforge.org/projects/activerecord/
  - Autor: David Heinemeier Hansson
  - Version: 2.1.1 (2008-09-04)
  - Konzept: ORM, 2-Schicht-Ansatz, DB-neutral
  - Bemerkungen: Ein Ruby Gem als „**Rails**“-Spinoff
  - Einführung: „Active Web Development with Rails“, 2nd ed., Kap. 17ff

ActiveRecord

- Standalone-Nutzung
  - Einbindung

```
require "rubygems"
require "active_record"
```

  - DB-Verbindung
    - Parametersatz hängt von der verwendeten DB ab.
    - Hier zwei Beispiele für die Verbindung zu SqLite3 und MySql:

```
ActiveRecord::Base.establish_connection(
  :adapter => "sqlite3",
  :dbfile => "crud_demo.sql3"
)
ActiveRecord::Base.establish_connection(
  :adapter => "mysql", :host => "localhost",
  :database => "rubydemo",
  :username => "werntges", :password => "rubypw"
)
```

ActiveRecord

- Standalone-Nutzung
  - Anlegen und Entfernen einer Tabelle mittels „migrations“, Bsp. „books“

```
# Zunächst Gems laden und DB-Verbindung regeln, s.o.

class CreateBooks < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.string :title
      t.string :author1, :author2, :author3, :editor
      t.string :publisher, :isbn
      t.integer :pubyear
      t.integer :edition
    end
  end
  def self.down
    drop_table :books
  end
end
```

ActiveRecord

- Tabellenoptionen

```
# Tabelle ggf. überschreiben
:force => true
# Tab. wird bei disconnect gelöscht
:temporary => true
# String an „create table“ der DB durchreichen
:options => " … "
```

- Anwendungsbeispiel:

```
create_table :books, :force => true do |t|
    t.string :title
    t.string :author1, :author2, :author3, :editor
    t.string :publisher, :isbn
    t.integer :pubyear
    t.integer :edition
  end
end
```

ActiveRecord

- Spaltenoptionen

```
# Mussfeld (not null-Bedingung setzen)
:null => false
# Grenze für Feldgröße festlegen
:limit => size
# Defaultwert für Spalte angeben
:default => value
# Nur für decimal: Anzahl gespeicherter Stellen/Nachkommastellen
:precision => 8
:scale => 2
```

- Bespiel

```
create_table :books do |t|
    t.string :title, :null => false, :limit => 128
    t.string :author1, :author2, :author3, :editor
    t.string :publisher, :isbn
    t.integer :pubyear
    t.integer :edition, :default => 1
  end
end
```

ActiveRecord

- Unterstützte Datentypen
  - Abstrahiert von verwendeter Datenbank!

```
:binary
:boolean
:date
:datetime
:decimal
:float
:integer
:string
:text
:time
:timestamp
```

- Abbildungen
  - In Ruby: meist als „string“, sonst sinngemäß
  - In konkreten Datenbanken: **Produktabhängig!**

ActiveRecord

- CRUD-Beispiel (Online-Demo)
  - ***C** reate a record*
    - Neues *Book*-Objekt anlegen
    - Felder via *setter* belegen
    - In DB sichern („create“) mit Methode „save“
    - Beim erstmaligen Sichern wird „id“ vergeben!
  - ***R** ead record(s)*
    - Verschiedene Methoden, um Objekte aus DB zu selektieren/lesen
    - In Demo:

```
find(id),
find(:all, :conditions => …),
find_by_sql('SELECT …')
```

  - ***U** pdate a record*
    - Gelesenes Objekt ändern und erneut per „save“ sichern
  - ***D** elete a record*
    - Mit Klassenmethode „delete(id)“

ActiveRecord

- Namenskonventionen
  - Name der Tabelle:
    - Klein, Plural, Teile mit ‚_‘ getrennt
  - Klassenname:
    - Groß, Singular, CamelCase
  - Spaltenname:
    - Klein
  - Getter/Setter-Name:
    - Klein
- Beispiel
  - Name der Tabelle
    books, line_items
  - Klassenname
    Book, LineItem
  - Spaltenname
    title
  - Getter/Setter dazu
    title

ActiveRecord

- ActiveRecord in Rails
  - Einbindung
    - Geschieht automatisch
  - DB-Konfiguration
    - Hinterlegt in `proj/config/database.yml` (Beispiel zeigen)
    - (YAML = Persistenz-Mechanismus, leichtgewichtiger als XML)
    - 3 DB-Versionen, unterschieden mittels ENVIRONMENT-Variable „RAILS_ENV“: „development“, „test“, „production“
  - Einhaltung der Namenskonvention
    - Automatisch, mittels Hilfsmethoden aus Gem „active_support“!

ActiveRecord

- Das Migrations-Konzept von Rails
  - Das Datenbankschema eines Projekts wird schrittweise entwickelt
  - Jeder Schritt (z.B. Anlegen einer Tabelle, Umbenennen einer Spalte, Laden von Stammdaten, Ändern von Spaltenattributen) wird reversibel definiert
  - Jeder Schritt wird als „migration“-Datei in **`db/migration`** realisiert und mit einer Versionsnummer (Rails 2: i.w. ein Zeitstempel) versehen
  - Die automatisch verwaltete Tabelle „schema_migrations“ verwaltet in ihrer einzigen Spalte „version“ die aktuelle Versionsnummer
  - Neue Migrationsdateien werden eingearbeitet mittels

    ```
    $ rake db:migrate
    ```

  - Bestimmte (insb. zurückliegende) Versionsstände erreichbar mit (Bsp:)

    ```
    $ rake db:migrate VERSION=20081118220547
    ```

  - Den Überblick zum erreichten Datenbank-Schema liefert Rails in Datei

    `db/schema.rb`

ActiveRecord

- Daten-Migrationen
  - Stammdaten lassen sich analog zu Testdaten aus „fixtures“ laden
  - *Bitte beschränken auf Daten, die so auch in die Produktions-DB gehören!*
- Code-Beispiel

```
$ mkdir db/migrate/master_data
```

Datei „books.yml“ dort mit folgendem Inhalt anlegen:

```
Rails2:
  title: Agile Web Development with Rails
  author1:  Dave Thomas
  author2:  David Heinemeier Hansson
  publisher:  The Pragmatic Programmers LLC.
  isbn:  0-9776166-3-0
  pubyear:  2006
  edition: 1

# Weitere Bücher …
```

ActiveRecord

- Code-Beispiel (Forts.)

```
$ script/generate migration load_books_data
```

Migration-Datei „…_load_books_data.rb“:

```
require "active_record/fixtures"
class LoadBooksData < ActiveRecord::Migration
  def self.up
    down
    directory = File.join(File.dirname(__FILE__), "master_data")
    Fixtures.create_fixtures(directory, "books")
  end

  def self.down
    Books.delete_all       # Alle Datensätze löschen
  end
end
```

ActiveRecord

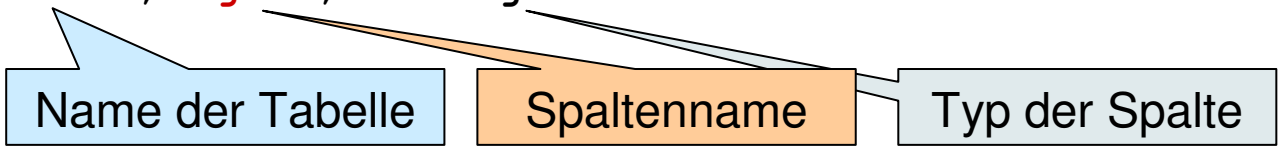
- Code-Beispiel: **Ergänzen einer Spalte**

```
script/generate migration add_gtin
```

Migration-Datei „…_add_gtin.rb“:

```
require "active_record/fixtures"
class AddGtin < ActiveRecord::Migration
  def self.up
    # GTIN = ISBN-13 (just digits), hence Integer
    add_column :books, :gtin, :integer
  end

  def self.down
    remove_column :books, :gtin
                  # Spalte „gtin“ in Tabelle „books“ löschen
  end
end
```



ActiveSupport

(Ein kleiner Abstecher)

ActiveSupport

- Standalone-Nutzung des ActiveSupport-Gems
  - Einbindung

```
require "rubygems"
require "active_support"
```

- Maßnahmen
  - Erweitert (verändert?) eingebaute Klassen
  - Fügt neue Klassen hinzu

- Neu in Modul „Enumerable“
  - Methode „group_by“: Wandelt eine „collection“ in ein gruppierendes Hash

```
groups = books.group_by {|book| book.publisher_id}
```

  - Methode „index_by“: Wandelt eine „collection“ in ein indizierendes Hash

```
isbn_lookup = books.index_by {|book| book.isbn}
```

  - Methode „sum“: Berechnet eine Summe aus einer „collection“

```
total_sales = Sale.find(:all).sum {|sale| sale.value}
```

ActiveSupport

- Neu in Array, Hash, String, NilClass: Methode „blank?“
  - Vereinfachung, z.B. **… `if a.blank?`** statt **… `if a.nil? || a.empty?`**

```
[ ].blank?            # true
{ }.blank?            # true
{:a => 1 }.blank?     # false
"".blank?             # true
"   ".blank?          # true  (auch mit space chars)
" x ".blank?          # false
nil.blank?            # true
```

- Neu in Array
  - Methode „to_sentence“

```
%w/Milano Naples Rome/.to_sentence
                      # → "Milano, Naples, and Rome"
```

  - Methode „in_groups_of(n)“ Zerlegung in Teil-Arrays der Länge n

ActiveSupport

- Neu in String
  - „Sprechende“ Methoden für Teilstrings: „at“, „from“, „to“, „first“, „last“
  - „Sprechende“ Tests: „starts_with?“, „ends_with?“

  - Grundlagen für unsere Namenskonventionen
    - Man beachte die Beherrschung einiger Ausnahmen in der Pluralbildung

```
%w/book child erratum/.map{|item| item.pluralize}
  # → ["books", "children", "errata"]

"children".singularize      # "child"

"given_name".humanize       # "Given name"

"my booktitle here".titleize      # "My Booktitle Here"
```

ActiveSupport

- Neu in String (Forts.)
  - Beugungsregeln
    - Ausnahmen einpflegen

```
"formula".pluralize          # "formulas" (falsch!)

Inflector.inflections do |inflect|
  inflect.irregular "formula", "formulae"
end
```

  - Weiterführendes
    - Wörter ohne Plural werden mit „uncountable“ eingepflegt
    - Es gibt auch musterbasierte Bildungsregeln

- Andere Sprachen?
  - Rails hat (noch) keine Beugungsregeln für Deutsch hinterlegt. Das dürfte auch viel schwerer fallen als im Englischen, deshalb:
    Verwenden Sie nur englische Namen in Ihren Rails-Projekten!

ActiveSupport

- Neu in Integer

```
[1.even?, 1.odd?]    # [false, true]
1.ordinalize         # "1st"
20.megabytes         # 20971520
# analog: bytes, kilo/mega/giga/tera/peta/exabytes

10.seconds           # 10
10.minutes           # 600 (Umrechnung in Sekunden!)
# analog: hours, days, weeks, fortnights, months, years
```

- Zeit-Arithmetik

```
10.minutes.ago       # Mon Nov xx … (ein Time-Objekt)
10.minutes.from_now  # Mon Nov xx … (ein Time-Objekt)
# ferner: until, since (mit Time-Argument als Bezug)
```

- Neu in Time: Zahlreiche nützliche Methoden…
- Neue Klasse: ActiveSupport::TimeZone

… zurück zu ActiveRecord

ActiveRecord

- ActiveRecord in Rails
  - Migrations
    - … für Anlegen neuer Spalten
    - … für Ändern von Spaltentypen oder -optionen
    - … für Initial load
    - Verwaltung: VERSION, Tabelle „schema_info“ mit Spalte „version“
      - Ansteuern eines bestimmten Versionsstands
    - Spaltentypen
    - Optionen: null, limit, default; precision, scale (decimal)
    - Tabellen-Optionen: force, temporary, options; primary_key, :id=>false (join tables)
    - Tabellen umbenennen – oder besser nicht…
    - Indizes anlegen

ActiveRecord

- ActiveRecord in Rails
  - Suchen – die Methode „find“
    - find(n) Suche nach Record mit Id „n“
    - find(:first, …) Suche nach
    - find(:last, …)
    - find(:all [, …])
  - conditions, sichere Parameter-Übergabe
  - order, limit, offset, select
  - find_by_sql
  - Beispiele in **ActiveRecord::Base** (http://api.rubyonrails.org)

ActiveRecord: Validierung

- Situation
  - Objekt-Repräsentationen in der Persistenzschicht (hier: Datenbank) sollten stets gültig sein.
  - Abspeichern von Benutzereingaben beim Ändern vorhandener oder Erzeugen neuer Objekte gefährdet dies: Mussfelder könnten fehlen, Formatvorgaben missachtet werden usw.
  - Falsche Eingaben können auch programmatisch verursacht werden
  - ActiveRecord stellt die gemeinsame Verbindung zwischen Eingaben und der Datenbank-Schicht her und ist daher der ideale Ort für Gültigkeitsprüfungen!
- Konsequenz
  - Gültigkeitsprüfungen formalisieren und einbauen!
  - Geeigneter Ort: Vor dem Speichern in die DB, d.h. zu Beginn von „save“
    - Daraus folgt: Erzeugen und Modifizieren von ActiveRecord::Base-Objekten ist bis dahin möglich, auch mit fehlerhaften Daten.

ActiveRecord: Validierung

- Rails-Stil
  - Deklarativ
    - Teilen Sie Ihre Testkriterien mit
    - Überlassen Sie Rails die Ausführung & Details!
  - Redundanzarm
    - dank zahlreicher eingebauter Validierungshelfer
  - Offen für Abweichungen vom Standardverhalten
    - Auch *beliebige* Validierungsregeln lassen sich explizit codieren

ActiveRecord: Validierung

- Beispiel: Pflicht-Felder von „Book“ überwachen
  - Einbindung in die Datei app/models/book.rb

```
class Book < ActiveRecord::Base
  validates_presence_of :title, :author1 # usw.
end
```

  - Weitere Validierungshelfer:
    - validates_acceptance_of
    - validates_associated
    - validates_confirmation_of
    - validates_each
    - validates_exclusion_of
    - validates_format_of
    - validates_inclusion_of
    - validates_length_of
    - validates_numericality_of
    - validates_size_of
    - validates_uniqueness_of
  - Siehe Rails-Doku, Modul „ActiveRecord::Validations::ClassMethods“

ActiveRecord: Validierung

- Eigene Validierungen
  - Validierungsmethoden werden von Rails per Callback-Technik eingebunden
  - Genügen die eingebauten Validierungshelfer nicht, überschreiben Sie folgende Default-Methoden:

```
validate                  # Aufruf vor jedem "save"
validate_on_create        # … nur beim ersten "save" …
validate_on_update        # … bzw. nicht beim ersten "save"
```

  - Beispiel:

```
class Book < ActiveRecord::Base
  protected
  def validate # usw.
    errors.add("ISBN", "Formatfehler") unless
       isbn =~ /\d+-\d+-\d-[\dX]/ && isbn.size == 13
  end
end
```

  - Siehe Rails-Doku, Modul „ActiveRecord::Validations“

ActiveRecord: Validierung

- Was passiert im Fehlerfall?
  - „Flash“, errors-Objekt
  - **Online-Demo** mit Projekt „lib“, Modell „Book“

ActiveRecord: Unit tests

- Assertions
  - Grundlage: Die Ruby-Bibliothek „test/unit“
  - Prinzip:
    - Rails bezieht Unit-Tests auf Modelle (Exemplare von Modell-Klassen)
    - Beispiel-Daten und –Situationen seien gegeben
    - Ergebnis der zu testenden Methode sei bekannt
    - Formulieren Sie Ihre Erwartungshaltung formal per „assertion“
  - Beispiel: Automatisch generierter Test unter test/unit/book_test.rb :

```
require 'test_helper'

class BookTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  def test_truth
    assert true      # Immer wahr
  end
end
```

ActiveRecord: Unit tests

- Assertions: Fixture-Daten
  - Testdaten, YAML-codiert, in test/unit/fixtures/books.yml:

```
one:
  title: MyString
  authors: MyString
  publisher: MyString
  pub_year: 1
  edition: 1
  isbn: 1-234-56789-X
  gtin: 9781234567897

two:
  title: MyString
  authors: MyString
  publisher: MyString
  pub_year: 1
  edition: 1
```

  - Anmerkungen
    - „one“ ist ok (außer bei Prüfziffern), „two“ enthält Mussfeld „isbn“ nicht, auch „gtin“ fehlt
    - Quellen: http://ar.rubyonrails.org/classes/Fixtures.html bzw. API-Doku, Class “Fixtures”

ActiveRecord: Unit tests

- Assertions: Konkrete Tests
  - Code in test/unit/book_test.rb modifizieren:

```
require 'test_helper'
class BookTest < ActiveSupport::TestCase
  # fixtures :books  # erfolgt automatisch
  def test_something
    a = books(:one)                 # Fixture-Daten
    b = books(:two)                 # via Namen laden
    assert_valid a
    assert !b.valid?                # ISBN fehlt hier
    assert_equal '0', b.isbn        # Default ok?
  end
end
```

  - Test-DB vorbereiten, Test ausführen

```
$ rake db:test:prepare
$ rake test:units
```

ActiveRecord: Unit tests

- Assertions: Eingebaute Varianten

```
assert(boolean, message)
  assert !b.valid?

assert_equal(expected, actual, message)
assert_not_equal(expected, actual, message)
  assert_equal '0', b.isbn

assert_nil(object, message)
assert_not_nil(object, message)
  assert_nil b.gtin, "GTIN doch vorhanden"

assert_in_delta(expected_float, actual_float, message)
```

ActiveRecord: Unit tests

- Assertions: Eingebaute Varianten (Forts.)

```
assert_raise(Exception, …, message) { block … }
assert_nothing_raised(Exception, …, message) { block … }
  assert_raise(ActiveRecord::RecordInvalid) { Book.new.save! }

assert_match(pattern, string, message)
assert_no_match(pattern, string, message)
  assert_match /978\d{10]/, a.gtin.to_s

assert_valid(activerecord_object) # Eine Rails-Ergänzung
  assert_valid a

flunk(message)
  flunk "Leider verloren beim Spiel '9:1`" if rand < 0.1
```

ActiveRecord: Unit tests

- Womit testen?
  - Auswahl von Testfällen sorgfältig vornehmen!

- Was testen?
  - Korrektheit des Normalbetriebs
  - Gezielt: Verhalten in Grenzfällen
  - Fehlerverhalten
  - Alle „möglichen“ User-Inputs
  - Geschäftslogik, insb. bei Abhängigkeiten zwischen Modellen
  - Technisch: Alle ergänzten Methoden!

ActiveRecord: Relationen zwischen Tabellen

- 1:n-Beziehungen
  - Beispiel: Ein Verlag vertreibt/verlegt viele Bücher
  - Rails repräsentiert diese Beziehung „deklarativ“ in den Modellen
    - Das untergeordnete Modell („Book“) deklariert seine Beziehung wie folgt:

```
class Book < ActiveRecord::Base
  belongs_to :publisher    # Singular
end
```

    - Wirkung: Dynamische Erzeugung von Methoden, insb. von „publisher“ und „publisher=()“, nicht nur von „publisher_id“
    - Das übergeordnete Modell („Publisher“) deklariert entsprechend:

```
class Publisher < ActiveRecord::Base
  has_many :books          # Plural!
end
```

    - Grundlage / DB-Hintergrund: Die Tabelle zum abhängigen Modell enthält eine Spalte mit dem Fremdschlüssel in die Tabelle des übergeordneten Modells
    - Rails-Namenskonvention: Modellname (singular, klein) + „_id“
      - Hier: „books“ enthält die Spalte „publisher_id“ vom Typ „integer“
    - ACHTUNG: Bereits beim Anlegen der Felder beachten (→ Migrations)

ActiveRecord: Relationen zwischen Tabellen

- 1:1-Beziehungen
  - Spezialfall der 1:n-Beziehung
    - Fiktives Beispiel (ohne DB-Ebene)

```
class Book < ActiveRecord::Base
  has_one :book_cover      # Singular!
end
class BookCover < ActiveRecord::Base
  belongs_to :book
end
```

ActiveRecord: Relationen zwischen Tabellen

- n:m-Beziehungen
  - Realisierung über zusätzlichen *join table*
  - Beide Klassen müssen (via *foreign key*) aufeinander verweisen können, daher sind entsprechende id-Felder vorzusehen
  - Anlegen des *join table* (ohne Spalte „id“ – ohne ActiveRecord-Klasse!)
    - Namenskonvention: xxx_yyy (Namen der beiden Tabellen, alphabetisch sortiert)

```
create_table authors_books, :id => false do |t|
 t.integer :author_id, :null => false
 t.integer :book_id, :null => false
end
```

  - Deklaration in den Modell-Klassen:

```
class Book < ActiveRecord::Base
  belongs_to :publisher    # Singular
  has_and_belongs_to_many :authors        # Plural!
end
class Author < ActiveRecord::Base
  has_and_belongs_to_many :books          # Singular
end
```

ActiveRecord: Relationen zwischen Tabellen

- n:m-Beziehungen
  - Herstellung einer konkreten Beziehung

```
a1 = Author.find 1
a2 = Author.find 2          # Zwei Autoren

b = Book.new                # Ein neues Buch dieser Autoren!
b.title = "Unser neues Buch"
# usw., schließlich:
b << a1                     # Autor 1 an die Autorenliste anfügen
b << a2                     # Autor 2 an die Autorenliste anfügen
b.save                      # Dies genügt!
```

  - Kontrolle (später)

```
a = Author.find 1
a.books.size                # Anzahl seiner Bücher
a.books.last.title          # → "Unser neues Buch"

Books.find_by_title("Unser neues Buch").first.authors.size # 2
```

ActiveRecord: Relationen zwischen Tabellen

- Anmerkungen
  - Einhaltung der Namenskonventionen sorgt für sehr lesbaren Code – geradezu umgangssprachlich!
  - Eine Zeile „Deklaration“ (tatsächlich: ein Methodenaufruf) bewirkt die dynamische Erzeugung mehrerer Methoden
  - Bei Bedarf lassen sich die Namenskonventionen überschreiben, indem man einen Optionen-Hash folgen lässt.
    Beispiel dazu aus dem Rails-Buch, Kap. 18:

```
class Customer < ActiveRecord::Base
  has_many :orders
  has_one  :most_recent_order,
           :class_name => 'Order',
           :order => 'created_at DESC'
end
```