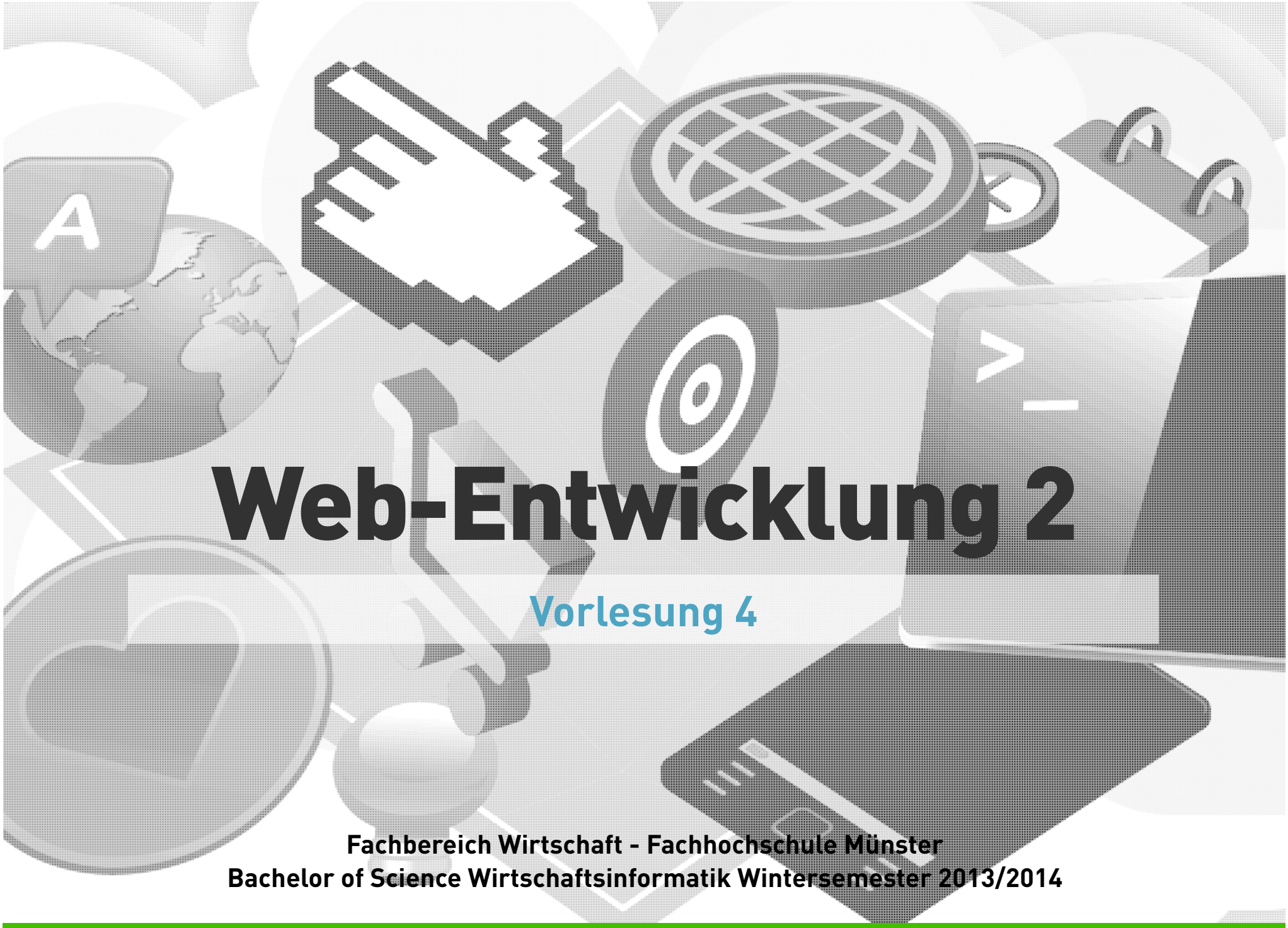


Web-Entwicklung 2

Vorlesung 4

Fachbereich Wirtschaft - Fachhochschule Münster
Bachelor of Science Wirtschaftsinformatik Wintersemester 2013/2014

Models: ActiveRecord

RECAP

- ORM (Object-Relational Mapper)
  - Übersetzt Operationen auf Objekten in Datenbankabfragen
- Grundfunktionen
  - Suchen, speichern und löschen in der Datenbank
  - Erzeugung von Gettern und Settern
- Erweiterte Funktionen
  - Assoziationen
  - Validierungen
  - Scopes

  ...

  → Mehr dazu im Verlauf der Veranstaltung!

ActiveRecord: Grundfunktionen

RECAP

- Rails Console: > rails console
- Erstellen eines Objekts:
  > Post.create :name => 'Neuer Eintrag'
- Suchen eines Objekts anhand der ID:
  > Post.find(1)
- Löschen eines Objekts:
  > post = Post.find(2)
  > post.destroy

Models: Grundfunktionen

RECAP

Erzeugen von Gettern und Settern

```
> post = Post.find(1)

> post.name

# => "Hello Rails"

> post.name = "Hallo Welt"

# => "Hallo Welt"

> post.save

# => true
```

ActiveRecord

Erweiterte Funktionen

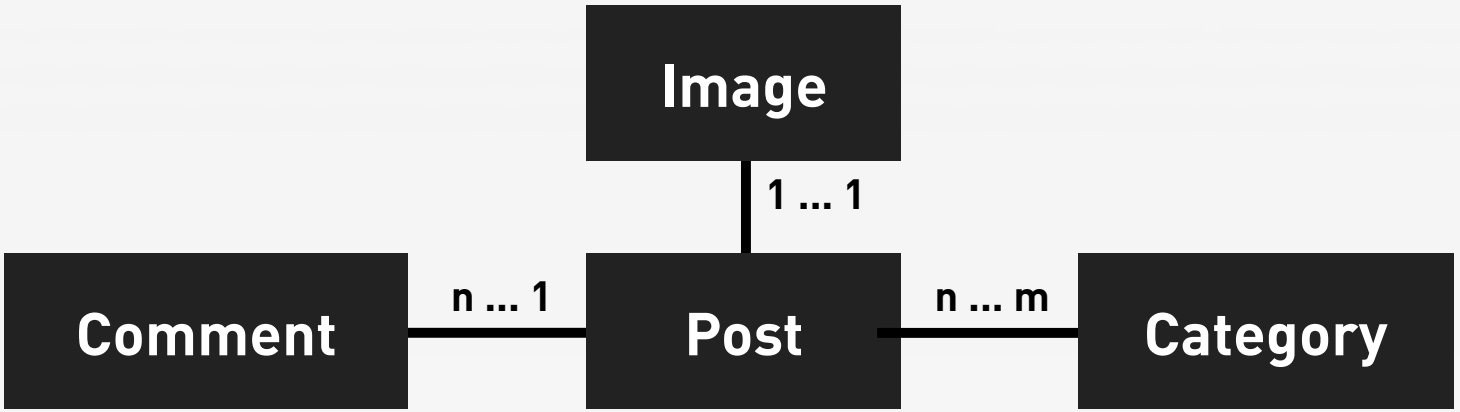
Erweiterte Funktionen

- Assoziationen
- Validierungen
- Callbacks
- Scopes
- „Model-Magic“

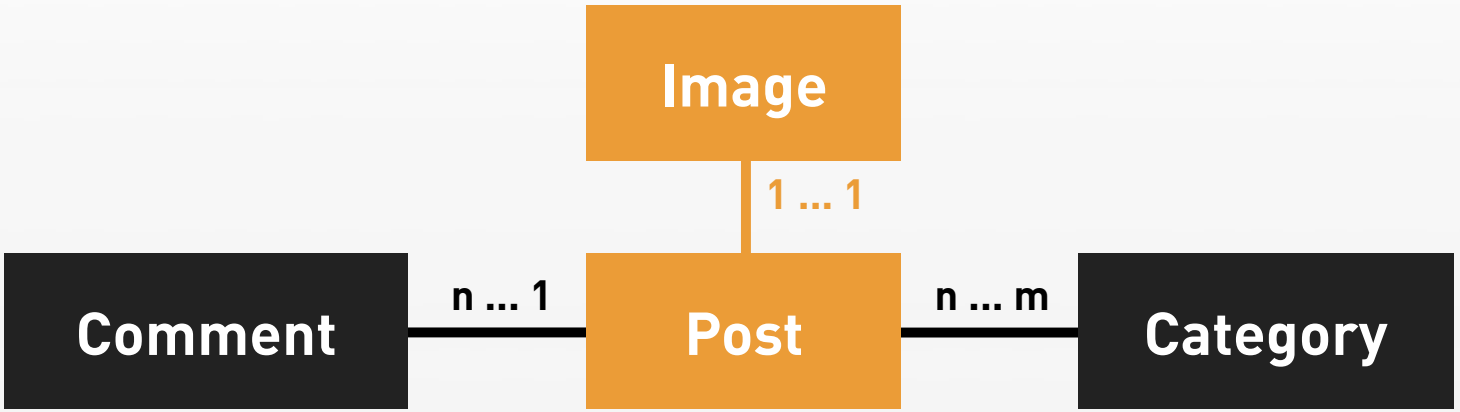
ActiveRecord: Assoziationen

ActiveRecord: Assoziationen

Assoziation:
Eine Assoziation ist eine Verbindung zwischen zwei ActiveRecord-Models. Assoziationen in Rails werden mit Hilfe einer **Domain-specific Language** (DSL) definiert.



ActiveRecord: 1:1-Assoziation



- > rails generate model image

ActiveRecord: 1:1-Assoziation

db/migrate/20121118210459_create_images.rb

```ruby
class CreateImages < ActiveRecord::Migration
  def change
    create_table :images do |t|
      t.string :url
      t.timestamps
    end
  end
end
```

app/models/post.rb

```ruby
class Post < ActiveRecord::Base
  has_one :image
end
```

ActiveRecord: 1:1-Assoziation

app/models/image.rb

```
class Image < ActiveRecord::Base
  belongs_to :post
end
```

- rails generate migration add_post_id_to_images

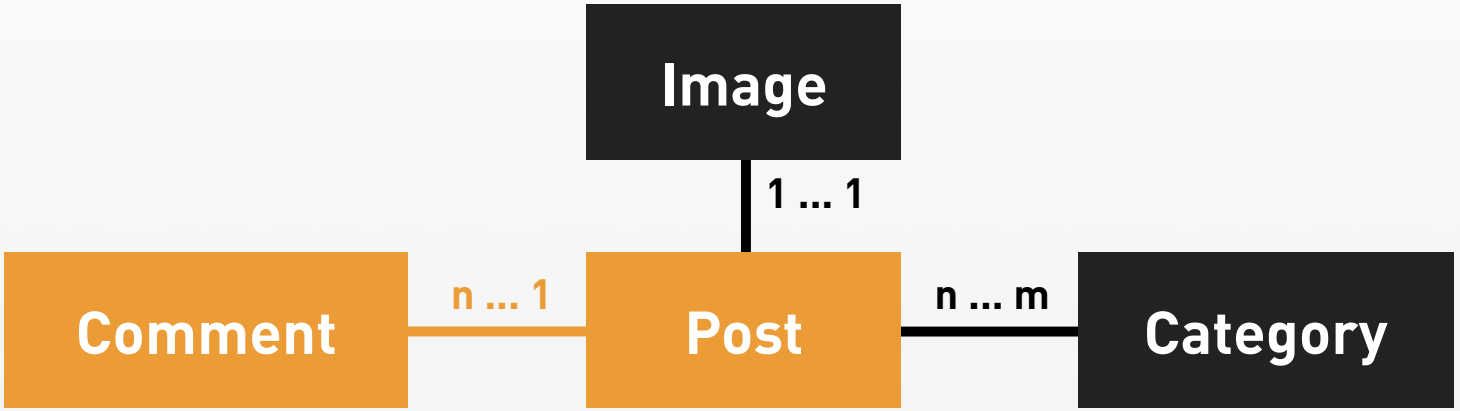
db/migrate/20121031120412_add_post_id_to_images.rb

```
class AddPostIdToImages < ActiveRecord::Migration
  def change
    add_column :images, :post_id, :integer
  end
end
```

ActiveRecord: 1:1-Assoziation

```
> rails console
> p = Post.find(1)
> p.image
# nil
> p.image = Image.create :url => "http:// ..."
> p.image.url
```

ActiveRecord: 1:n-Assoziation



- rails generate model comment body:text

ActiveRecord: 1:n-Assoziation

app/models/post.rb

```ruby
class Post < ActiveRecord::Base
  has_many :comments
end
```

app/models/comment.rb

```ruby
class Post < ActiveRecord::Base
  belongs_to :post
end
```

ActiveRecord: 1:n-Assoziation

- rails generate migration add_post_id_to_comments

db/migrate/20121118221232_add_post_id_to_comments.rb

```
class AddPostIdToComments < ActiveRecord::Migration
  def change
   add_column :comments, :post_id, :integer
  end
end
```

ActiveRecord: 1:n-Assoziation

```
> rails console
> p = Post.find(1)
> p.comments
=> []
> p.comments << Comment.new :body => "Toller Beitrag"
> p.comments
 => [#<Comment id: 1, body: "Toller Beitrag", created_at:
"2012-11-18 22:18:47", updated_at: "2012-11-18
22:18:47", post_id: 1>]
```

ActiveRecord: n:m-Assoziation

- http://railscasts.com/episodes/47-two-many-to-many
- http://guides.rubyonrails.org/association_basics.html

ActiveRecord: Validierungen

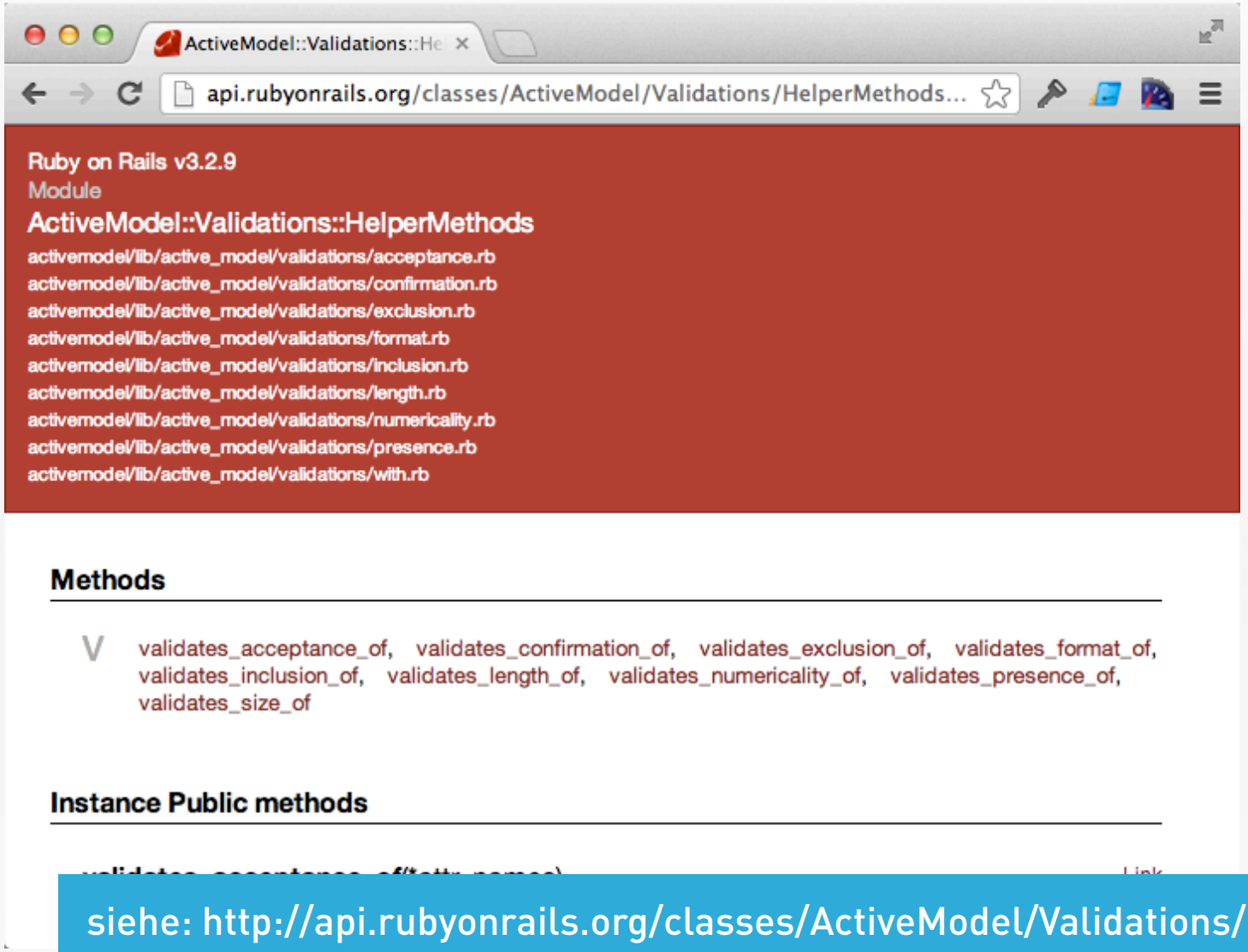
ActiveRecord: Validierungen

Validierungen:
Validierungen stellen sicher, dass nur gültige Objekte in der Datenbank gespeichert werden. Ebenso wie Assoziationen werden Validierungen in Rails mit Hilfe einer **Domain-specific Language** (DSL) definiert.

Validierungsoptionen

- Generell:
  - Datenbank-Constraints mit Stored Procedures
    Vorteile: „Historisch“ bewährt
    Nachteile: Datenbankabhängig, Intransparent, Schwer zu deployen und zu warten
  - Client-seitige Validierungen mit z.B. Javascript
    Vorteile: Unnötige Requests werden vermieden
    Nachteile: einfach zu umgehen
  - Controller-level Validierung
    Nachteile: Schwer zu warten, nicht DRY
  - Model-level Validierung
    Vorteile: nicht zu umgehen, DRY, Datenbankagnostisch

Validierungsoptionen



siehe: http://api.rubyonrails.org/classes/ActiveModel/Validations/HelperMethods.html

Validierungsoptionen

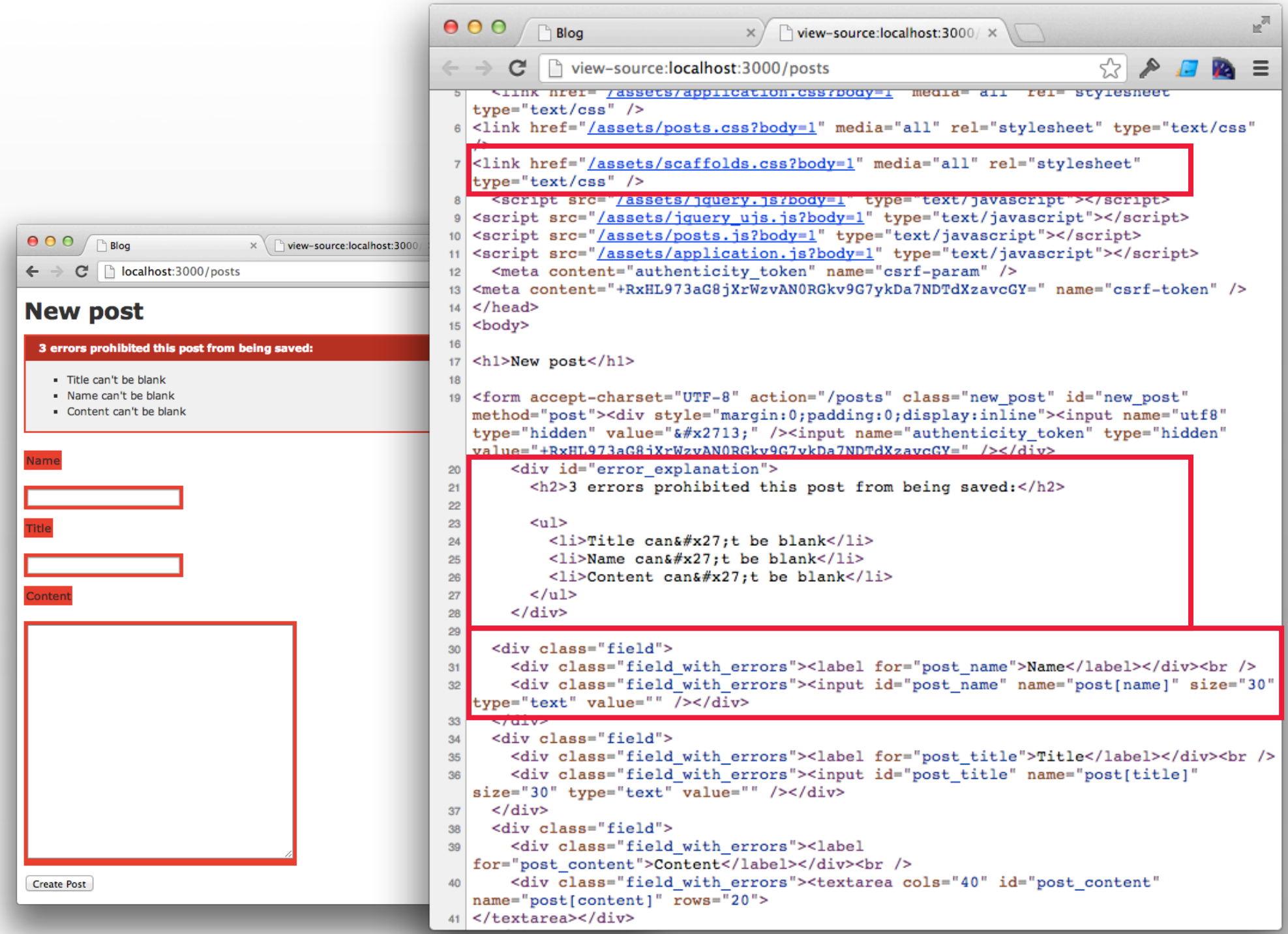
- Speziell in Rails (Beispiele):
  - Akzeptanz einer Bedingung:
    validates :terms_of_service, :acceptance => true
  - Überprüfung des Eingabeformats (mit Regular Expression):
    validates_format_of :email, :with => /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\Z/i, :on => :create
  - Überprüfung auf Länge einer Eingabe:
    - validates :name, :length => { :minimum => 2 }
    - validates :bio, :length => { :maximum => 500 }
    - validates :password, :length => { :in => 6..20 }
    - validates :registration_number, :length => { :is => 6 }
  - Prüfung auf Gleichheit von zwei Eingaben:
    validates :email, :uniqueness => true

Validierungsoptionen

- Speziell in Rails (Beispiele):

app/models/post.rb

```ruby
class Post < ActiveRecord::Base
  # Prüfung auf „Vorhandensein“
  validates_presence_of :title, :name, :content
end
```


Blog
view-source:localhost:3000
view-source:localhost:3000/posts
Blog
view-source:localhost:3000
localhost:3000/posts
New post
3 errors prohibited this post from being saved:
Title can't be blank
Name can't be blank
Content can't be blank
Name
Title
Content
Create Post
type="text/css" />
<link href="/assets/posts.css?body=1" media="all" rel="stylesheet" type="text/css" />
<link href="/assets/scaffolds.css?body=1" media="all" rel="stylesheet" type="text/css" />
<script src="/assets/jquery.js?body=1" type="text/javascript"></script>
<script src="/assets/jquery_ujs.js?body=1" type="text/javascript"></script>
<script src="/assets/posts.js?body=1" type="text/javascript"></script>
<script src="/assets/application.js?body=1" type="text/javascript"></script>
<meta content="authenticity_token" name="csrf-param" />
<meta content="+RxHL973aG8jXrWzvAN0RGkv9G7ykDa7NDTdXzavcGY=" name="csrf-token" />
</head>
<body>
<h1>New post</h1>
<form accept-charset="UTF-8" action="/posts" class="new_post" id="new_post" method="post"> <input name="utf8" type="hidden" value="✓" /><input name="authenticity_token" type="hidden" value="+RxHL973aG8jXrWzvAN0RGkv9G7ykDa7NDTdXzavcGY=" />

<h2>3 errors prohibited this post from being saved:</h2>

Title can't be blank
Name can't be blank
Content can't be blank

<label for="post_name">Name</label>
<input id="post_name" name="post[name]" size="30" type="text" value="" />

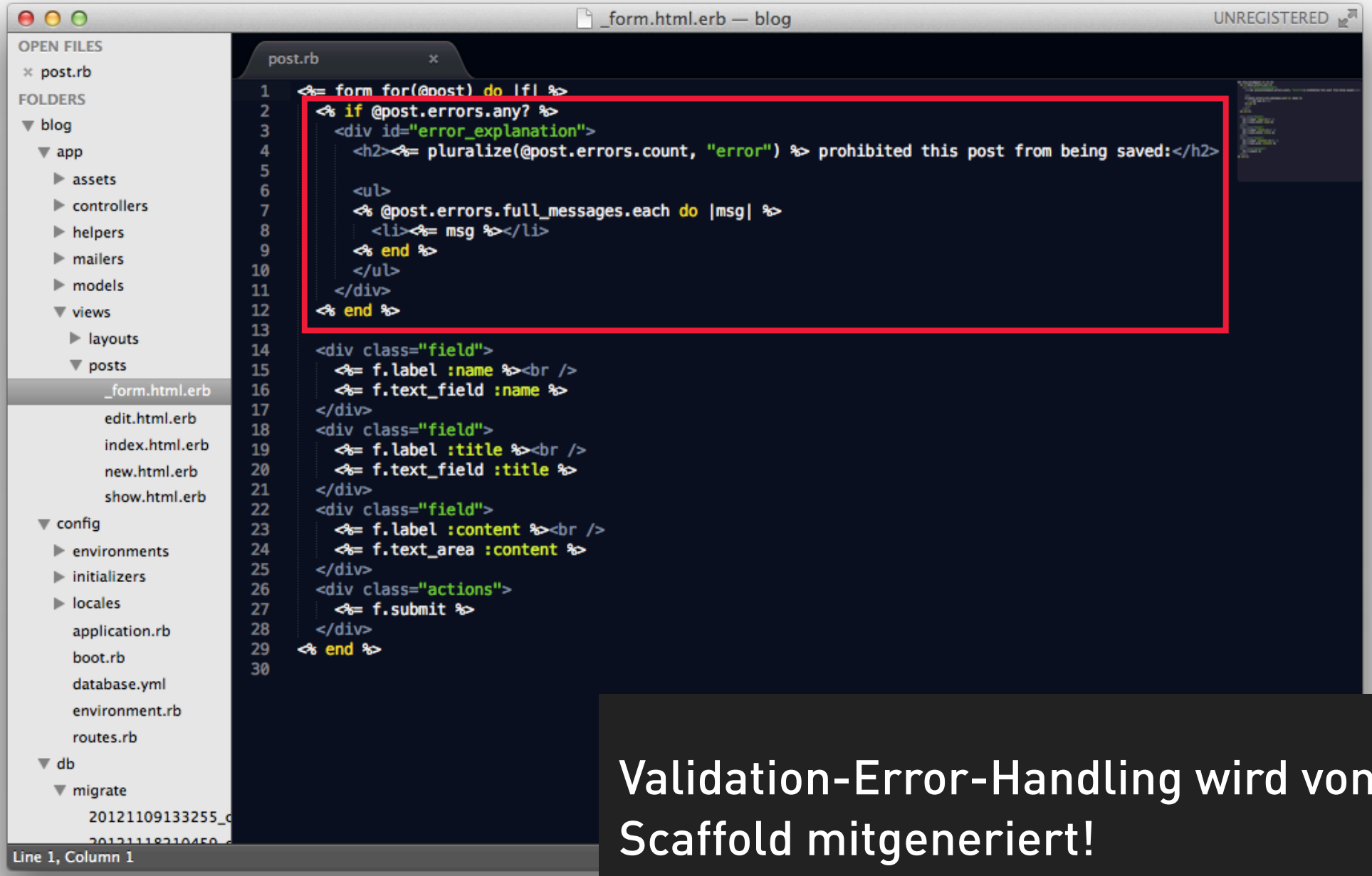
<label for="post_title">Title</label>
<input id="post_title" name="post[title]" size="30" type="text" value="" />

<label for="post_content">Content</label>
<textarea cols="40" id="post_content" name="post[content]" rows="20">
</textarea>

Funktionsweise von Validierungen

- Validierungen werden nach folgenden Methoden aufgerufen:
  create, create!, save, save!, update,
  update_attributes, update_attributes!
- Validierungsfehler können auf Model-Ebene abgefragt werden:
  > Post.new.errors.any? # => false
  > Post.create.errors.any? # => true

siehe: http://guides.rubyonrails.org/active_record_validations_callbacks.html



Validation-Error-Handling wird von Scaffold mitgeneriert!

ActiveRecord: Callbacks

Callbacks in Rails

Callbacks:
Methoden, die in bestimmten Situationen des Objekt-Lebenszyklus automatisch aufgerufen werden.

- Beispiele:
  - Nach Anlegen eines Objekts:
    before_validation, after_validation, before_save, around_save, before_create, around_create, after_create, after_save
  - Nach Aktualisierung eines Objekts:
    before_validation, after_validation, before_save, around_save, before_update, around_update, after_update, after_save
  - Nach Löschen eines Objekts:
    before_destroy, around_destroy, after_destroy

Funktionsweise von Callbacks

- rails generate migration add_comment_counter_column

db/migrate/20121119101134_add_comment_counter_column.rb

```
class AddCommentCounterColumn < ActiveRecord::Migration
  def up
    add_column :posts, :comment_count, :integer, :default => 0
  end

  def down
    remove_column :posts, :comment_count
  end
end
```

Funktionsweise von Callbacks

app/models/comment.rb

```
class Comment < ActiveRecord::Base
  attr_accessible :body, :post_id
  belongs_to :post

  after_create :increment_counter

  def increment_counter
    self.post.update_attribute(:comment_count,
self.post.comments.size)
  end
end
```

Funktionsweise von Callbacks

```
> p = Post.first

> Comment.create(:body => "Ein
Kommentar", :post_id => p.id)
```

siehe: http://guides.rubyonrails.org/active_record_validations_callbacks.html

ActiveRecord: Scopes

ActiveRecord: Scopes

- Conditions
- Joining
- Custom Scopes

siehe: http://guides.rubyonrails.org/active_record_querying.html

Conditions

- Einfache Abfragen:
  - > Post.first
  - > Post.last
  - > Post.all
  - > Post.find_all_by_name("Neuer Eintrag")

Conditions

- **Abfragemethoden:** where, select, group, order, reorder, reverse_order, limit, offset, joins, includes, lock, readonly, from, having
- > Post.where("comment_count > ?", 0)
- > Post.where("name like ?", "%eu%")
- > Comment.order("created_at desc").all
- > Post.select("id").all

Joining

- > Post.comments
- > Post.joins(:comments)

Custom Scopes

Custom Scope:
Abfrage, die sich über eine selbst definierte Methode aufrufen lässt.

app/models/post.rb

```
class Post < ActiveRecord::Base
  scope :published, ->() { where(:published => true) }
end
```

- Erst Spalte „published“ per Migration hinzufügen, dann:
  > Post.first.published

ActiveRecord: „Model-Magic“

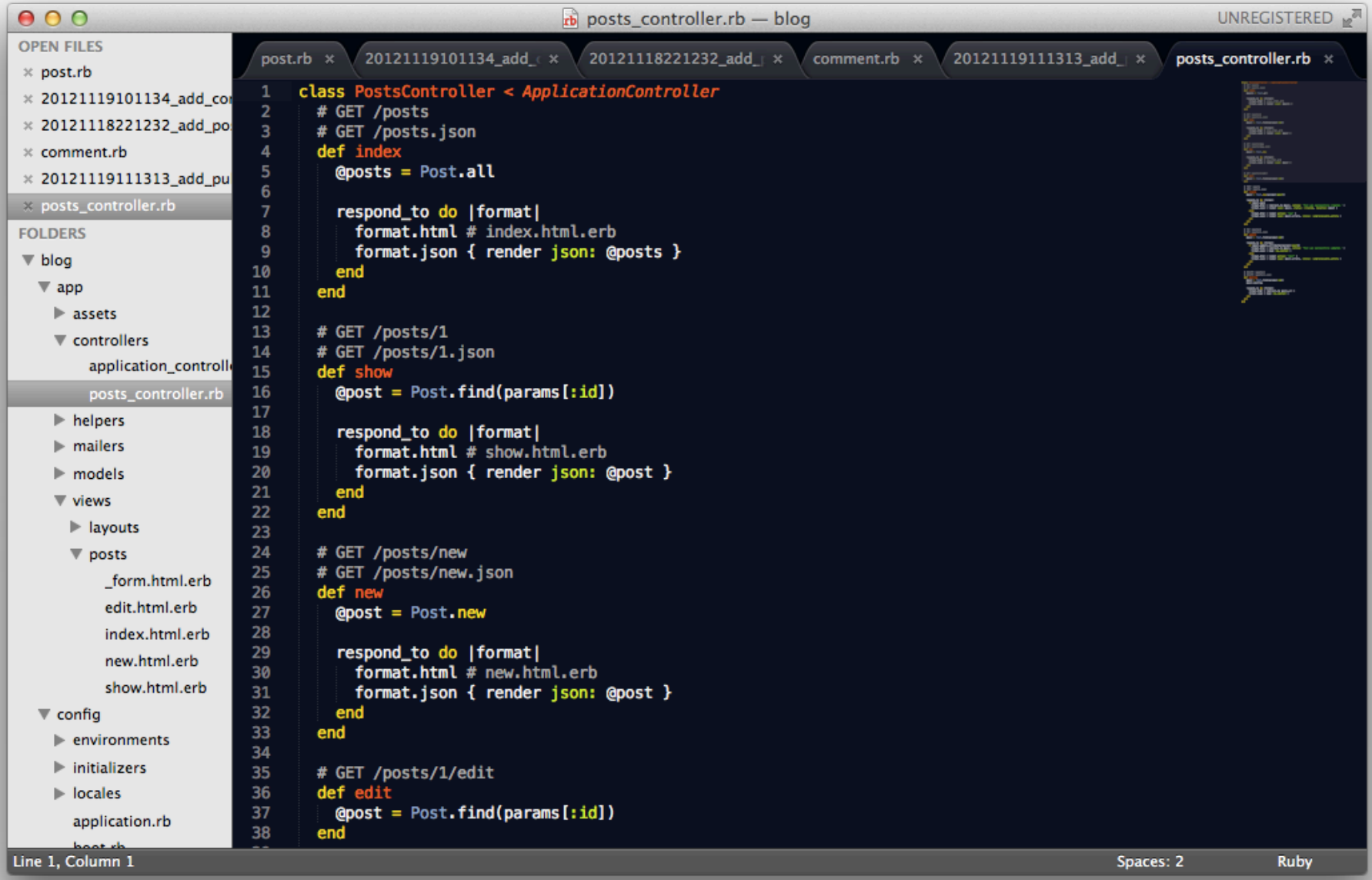
Counter Cache

app/models/comment.rb

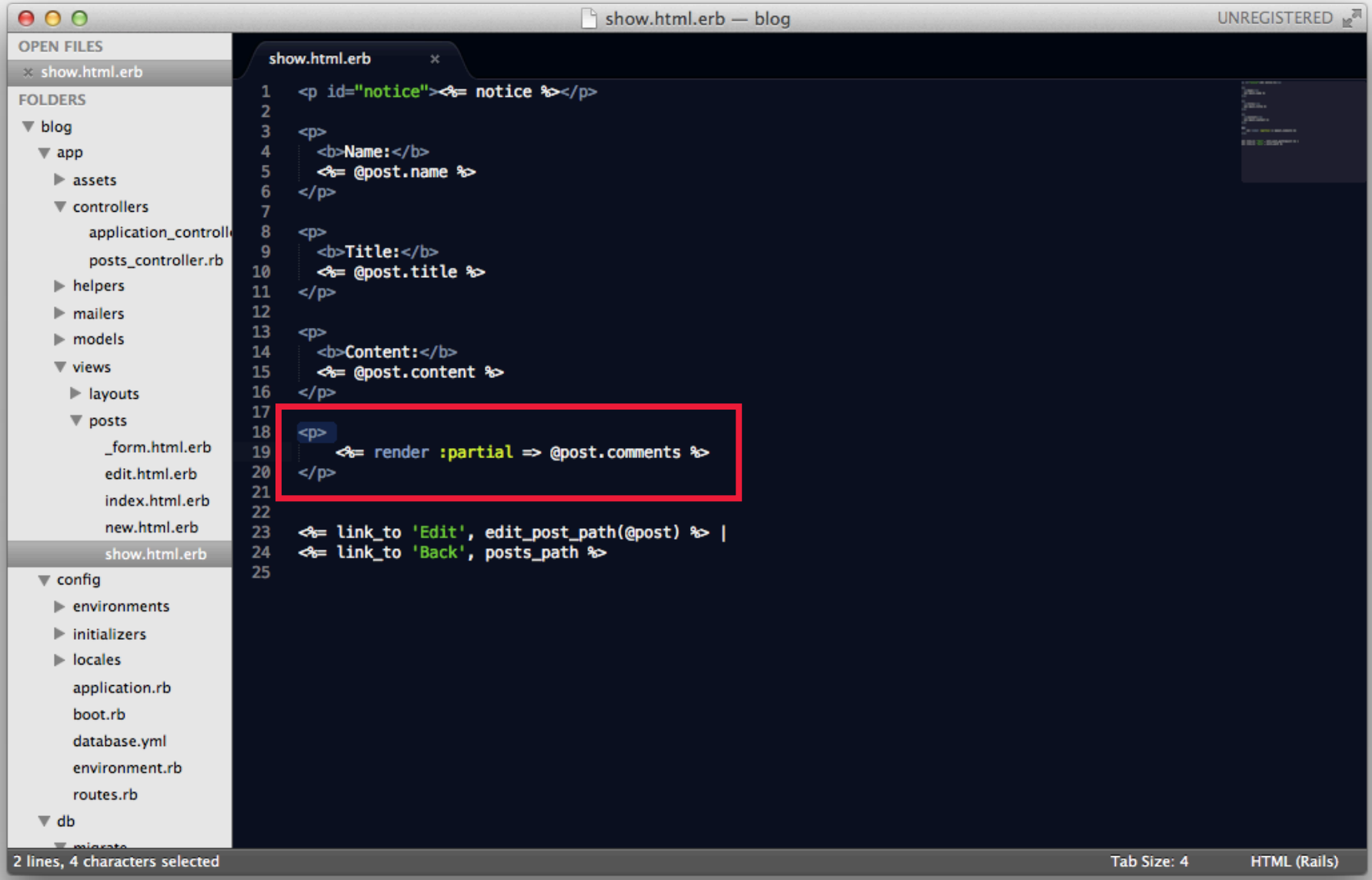
```
class Post < ActiveRecord::Base
    attr_accessible :body, :post_id
    # Counter Cache aktualisiert automatisch den Counter
    belongs_to :post, :counter_cache => :comment_count

    # after_create :increment_counter

    # def increment_counter
    #  self.post.update_attribute(:comment_count,
  self.post.comments.size)
    # end
end
```

ActionController und ActionView

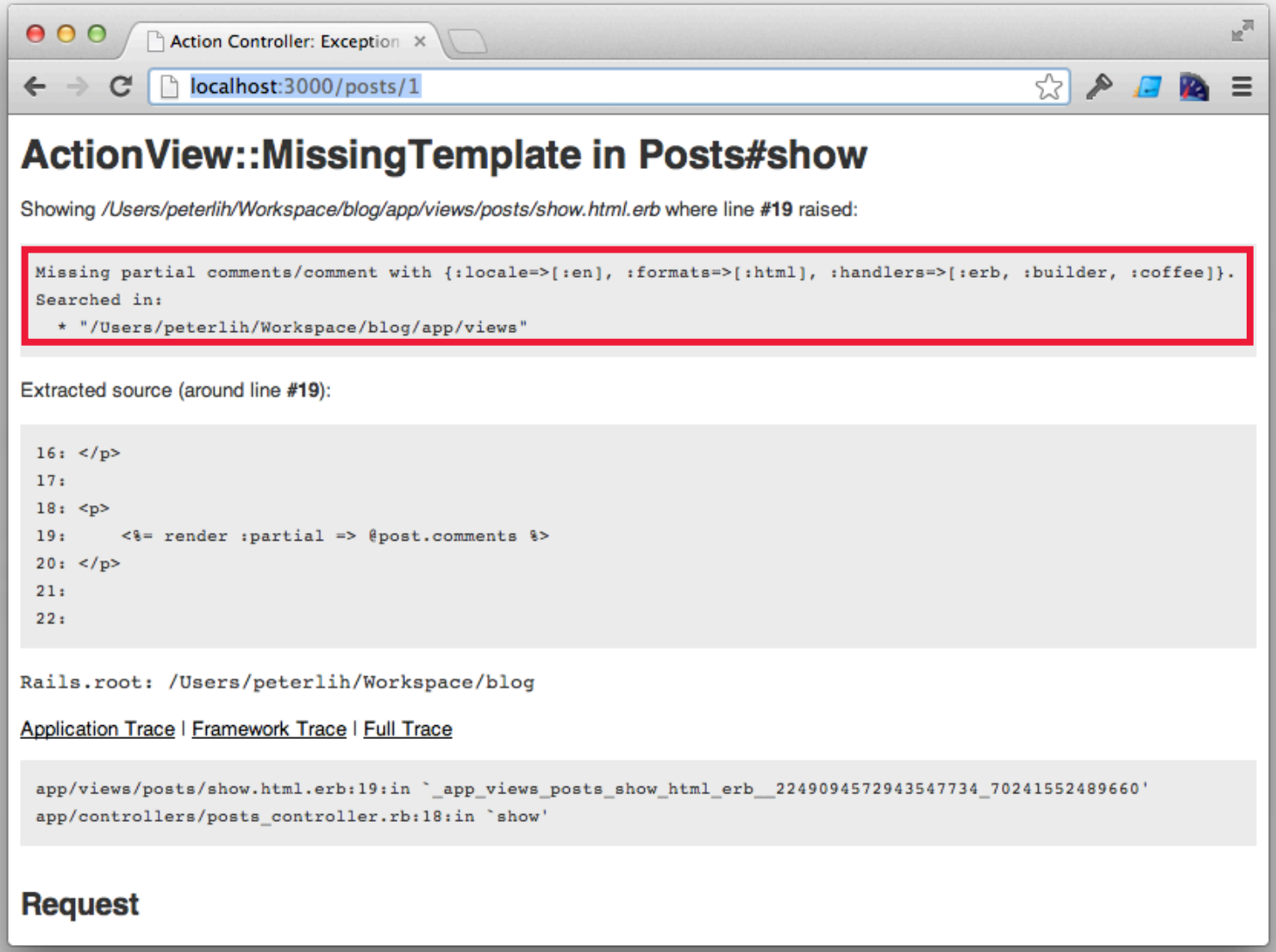
ActionController



View Partials



View Partials



View Partials

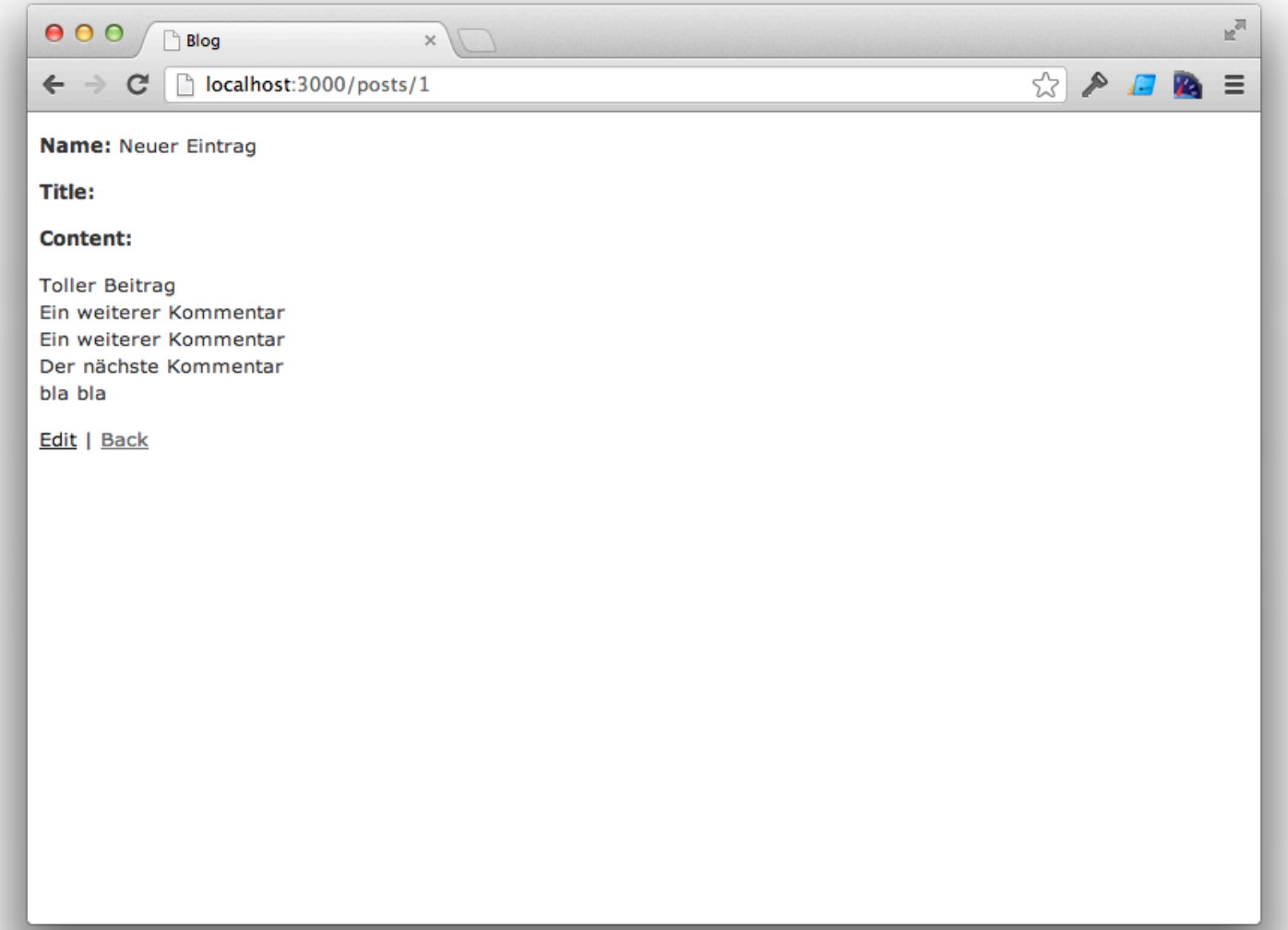
- > mkdir app/views/comments
- > touch app/views/comments/_comment.html.erb

app/views/comments/_comment.html.erb

```
<%= comment.body %><br/>
```

- Durch Übergabe einer Collection an die Partial, automatische Iteration über das Partial so lange noch weitere Einträge in der Collection sind!
- Varianten: (...)

View Partials



Warmup

- Aufsetzen des Blog-Projektes aus den letzten beiden Vorlesungen:
  - Gemset (https://rvm.io/gemsets/) anlegen:
    rvm gemset create we2_blog
  - Gemset anwenden: rvm gemset use we2_blog
  - git clone git@github.com:we2/blog.git
  - cd blog

Warmup

EXKURS

- Änderung der Datenbank-Konfiguration

config/database.yml

```
development:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: we2_blog_dev
  pool: 5
  username: root
  password:
  socket: /tmp/mysql.sock
```

Warmup

EXKURS

- Änderung des „Datenbank-Treibers“

config/database.yml

```
source 'https://rubygems.org'
gem 'rails', '3.2.8'
rails/rails.git'
#gem 'sqlite3'
gem 'mysql2'
group :assets do
  gem 'sass-rails',   '~> 3.2.3'
  gem 'coffee-rails', '~> 3.2.1'
  gem 'uglifier', '>= 1.0.3'
end
gem 'jquery-rails'
```

Warmup

- Aufsetzen des Blog-Projektes aus den letzten beiden Vorlesungen (contd.):
  - Gems installieren: > bundle install
  - Starten der Datenbank: > mysqld
  - Rails Server starten: > rails s
  - Datenbank anlegen: > rake db:create
  - Datenbank migrieren: > rake db:migrate
  - URL aufrufen: > http://localhost:3000/posts