

Datenbankapplikationen mit Ruby on Rails

Datenbanksysteme 2011
Universität Osnabrück
Gastvorlesung von Nicolas Neubauer

Ruby

- plattformunabhängige, höhere *Programmiersprache*
- 1995 erschienen
- Yukihiro Matsumoto, Entwickler
- ~ 2000 Bekanntheit außerhalb Japans
- Objektorientierung
- dynamische Typisierung
- Garbage Collection
- ...

Rails

- Open Source *Webapplication* Framework
- 2004 erschienen
- Anfang 2011 in Version 3
- David Heinemeier Hannsson
- Ziel: schnellere, „agilere“ Webentwicklung
- häufig benötigte Funktionalität bereits vorhanden

Kurzeinführung in Ruby

- „alles ist ein Objekt“
- 42 ist eine konstante Zahl (Fixnum)
- :symbol ist ein „konstanter String“
- „String“ ist ein String mit Inhalt String
- `meine_variable = 42` ist eine Variablenzuweisung
- Punktsyntax für Methodenaufrufe:
„Ruby ist klasse“.sub(„klasse“, „toll“)

Ruby: Collections

- `[]` ist ein leeres Array \Leftrightarrow `Array.new`
- `[42, 23]` ist ein Array mit Einträgen 42 und 23
- `mein_array[0]` gibt den ersten Eintrag zurück
- `{}` ist ein leerer Hash (etwa Hash-Map in Java)
- `{:key => „value“, :key2 => „value2“}`
- `mein_hash[:key]` gibt den Wert für den Schlüssel `:key` zurück

Ruby: Kontrollstrukturen

```
if i == 5
  return „i ist 5“
end
```

```
return „i ist 5“ if i == 5
```

```
unless i == 5                                # ~ if !(i == 5)
  return „i ist nicht 5“
end
```

```
return „i ist nicht 5“ unless i == 5
```

Ruby: Kontrollstrukturen

```
while i > 0  
  i -= 1  
end
```

```
until i == 0  
  i -= 1  
end
```

```
for element in mein_array  
  puts element.to_s  
end
```

Ruby: Methoden und Variablen

```
def hallo_welt
```

```
  lokale_variable1 = 23 #dynamisch typisiert
```

```
  lokale_variable2 = „Hallo Welt!“
```

```
  @instanz_variable = „In der Instanz sichtbar!“
```

```
  @@klassen_variable = „vgl. static in Java“
```

```
  $globale_variable = „überall sichtbar“
```

```
end
```

Impliziter Rückgabewert der Methode

`hallo_welt()` ist String: „überall sichtbar“

Ruby: Beispiel - Collatz

```
def collatz(x)

  anz_iterationen = 0

  until x == 1
    if x % 2 == 0
      x = x/2
    else
      x = 3*x+1
    end
    anz_iterationen += 1
  end

end
```

Ruby: Syntax-Besonderheiten

- Weglassen von Klammern bei Eindeutigkeit

```
[42,23].include? 42 <=> [42,23].include?(42)
```

```
methode_mit_hash_paramater :key => „value“
```

<=>

```
methode_mit_hash_paramater({:key => „value“})
```

- Übergeben von Codeblöcken an Methoden

```
10.times do |i|  
  puts i  
end
```

Ruby: Syntax-Besonderheiten

Demo

3 Konzepte von Rails

Don't Repeat Yourself



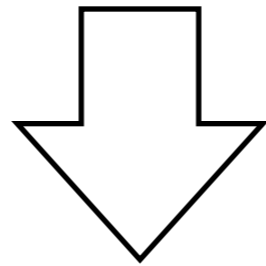
Model-View-Controller

Convention Over Configuration

Model-View-Controller Design Pattern

Idee:

Code nach Kategorien M,V,C strukturieren.



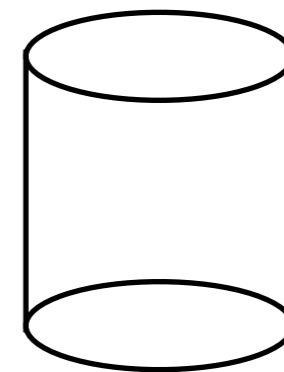
- bessere Programmstruktur
- Wiederverwertbarkeit von Komponenten
- Flexibilität

Model-View-Controller Design Pattern

I. Model und ORM

- Programmteile, die auf Datenhaltung arbeiten
- Abstraktion von der eigentlichen Datenhaltung

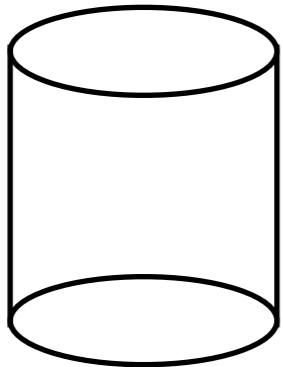
```
class Modelklasse  
  def besorge_nachrichten()  
  end
```



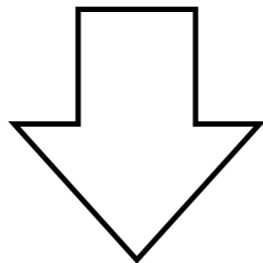
Nachrichten:
{[NachrichtID, Nachricht,
 Nutzer, ...]}

Model-View-Controller Design Pattern

I. Model und ORM: Objektrelationales Mapping



Nachricht: {[NachrichtID, Nachricht,
Nutzer, ...]}



```
class Nachricht  
  
end
```

Model-View-Controller Design Pattern

I. Model und ORM: Objektrelationales Mapping

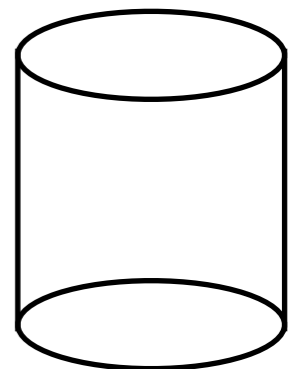


Nachricht.all

übersetze in SQL

übersetze Tupel-Ergebnisse
in Objekte

SELECT * FROM Nachricht

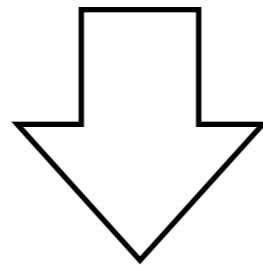


Model-View-Controller Design Pattern

2.View

- (grafische) Datenrepräsentation
- Templates

```
<li><%= @nachricht %></li>
```



```
<li>Hallo Welt!</li>
```

Model-View-Controller Design Pattern

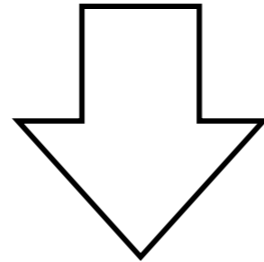
3. Controller

- Applikationssteuerung
- Vermitteln zwischen Daten und Repräsentation

Model-View-Controller Design Pattern

3. Controller

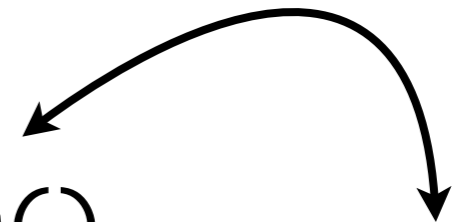
```
zeige_alle_nachrichten()
```



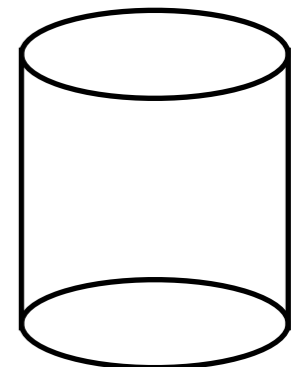
Controller-Klasse

```
Model.besorge_alle_Nachrichten()
```

```
View.zeige_Template_mit_  
Nachrichten(alle_Nachrichten)
```

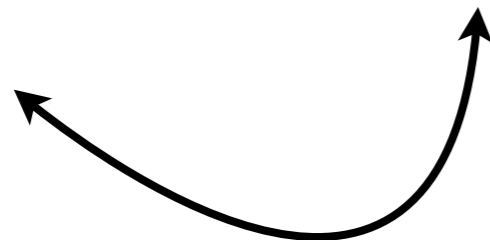


Modell Klasse

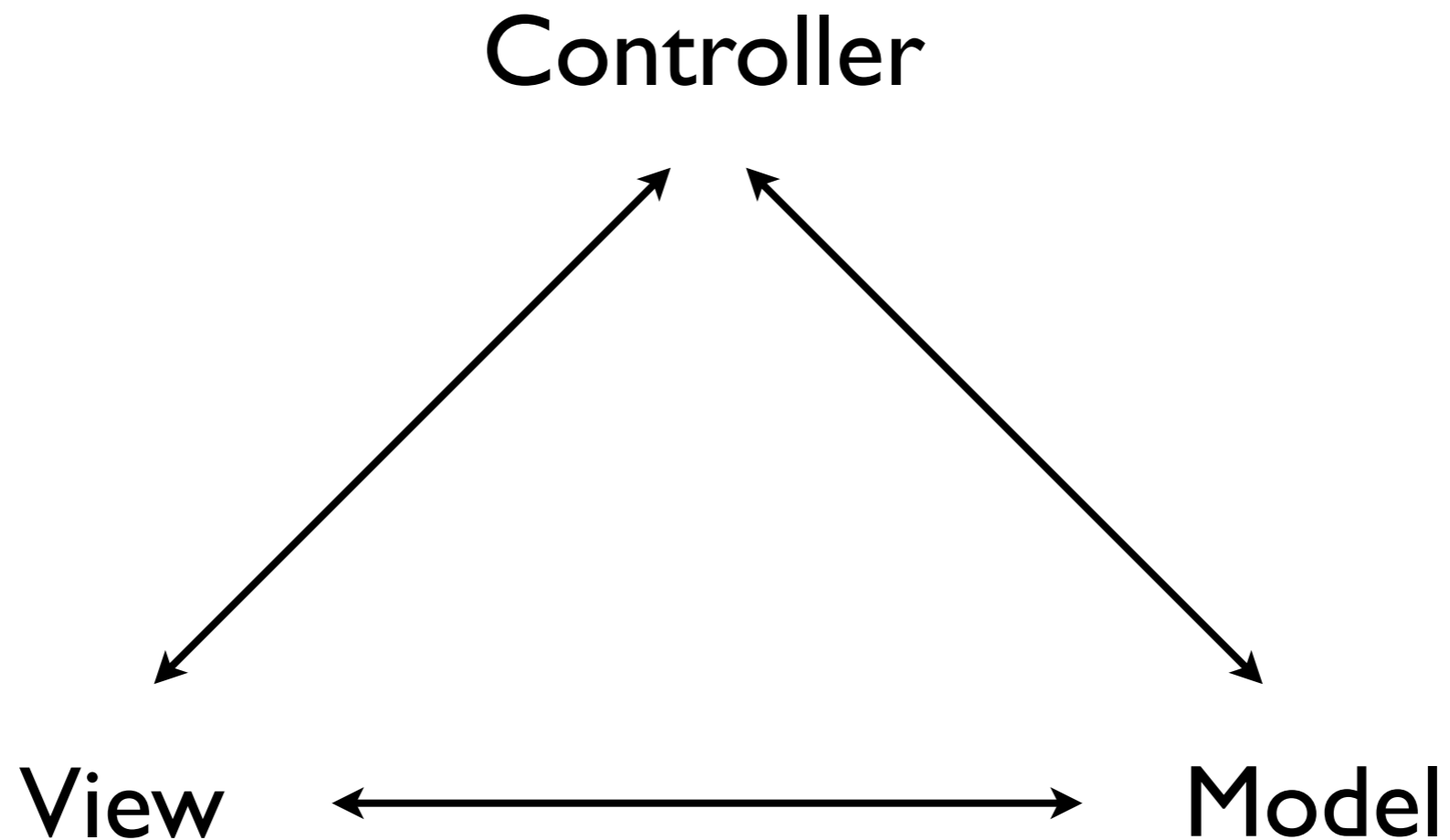


Template

```
<html>  
...  
</html>
```



Model-View-Controller Design Pattern



Eine Beispielapplikation

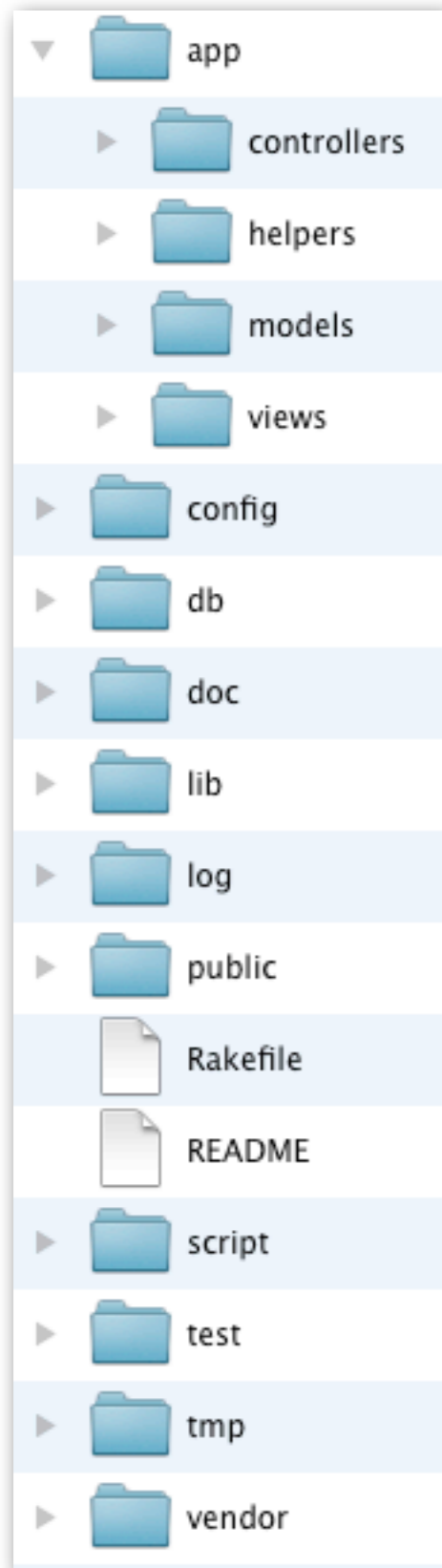
neue Rails Applikation erzeugen:

```
$ rails new meine_applikation -d mysql
```

Datenbanktyp im Backend,
SQLite: Standard

Applikationsname

Eine Beispielapplikation



← **eigentliche Applikationsdaten**

← **Konfigurationsdateien,**
← **Datenschemadefinitionen, „Migrations“**

← **Testumgebung**

Eine Beispielapplikation: Datenbankverbindung

- Rails mit Datenbank verbinden
- YAML-Konfigurationsdatei:
`config/database.yml`

```
development:
```

```
  adapter: mysql
```

```
  encoding: utf8
```

```
  reconnect: false
```

```
  database: DATENBANKNAME
```

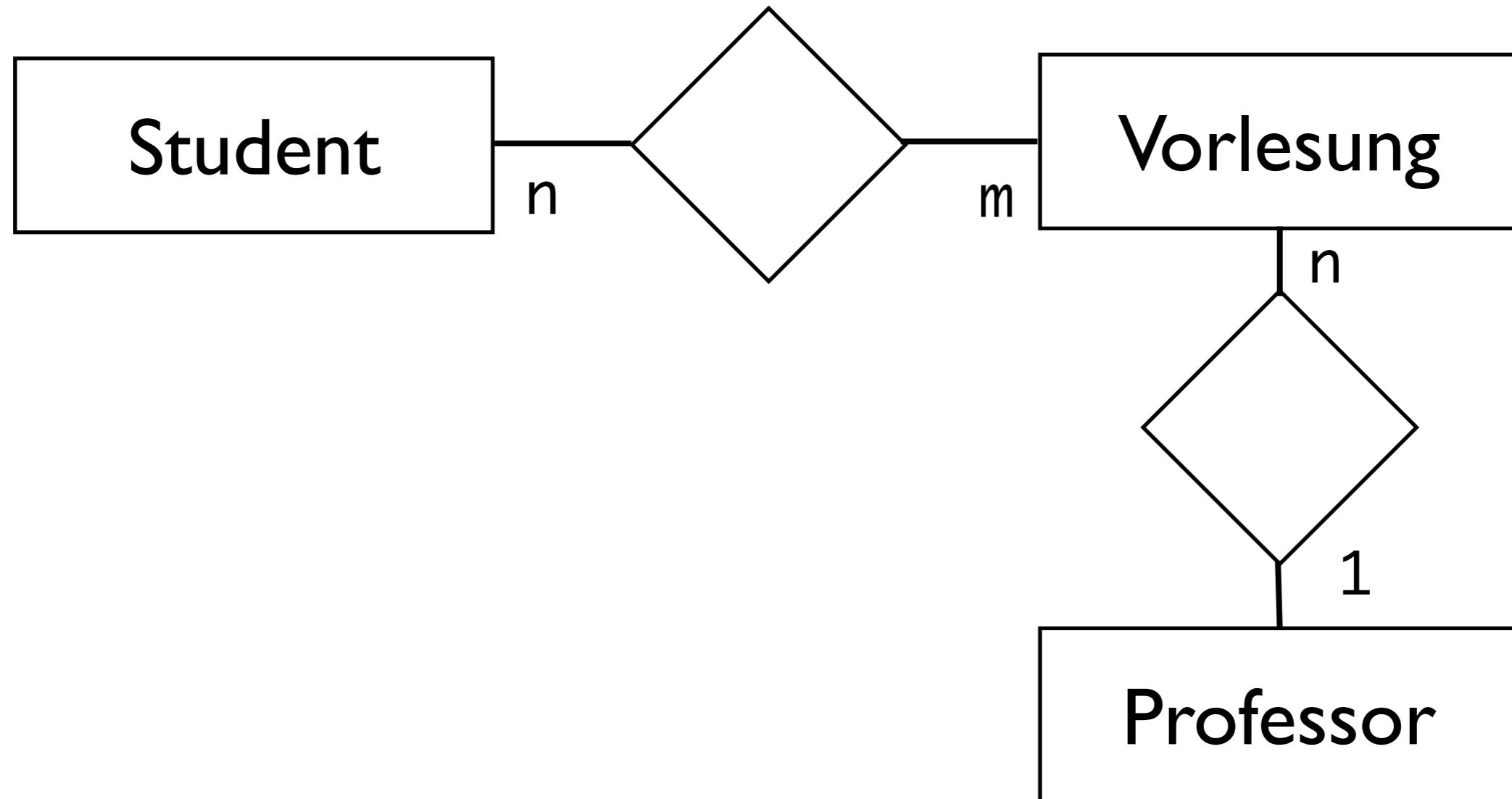
```
  pool: 5
```

```
  username: BENUTZERNAME
```

```
  password: PASSWORT
```

```
  host: HOSTNAME
```

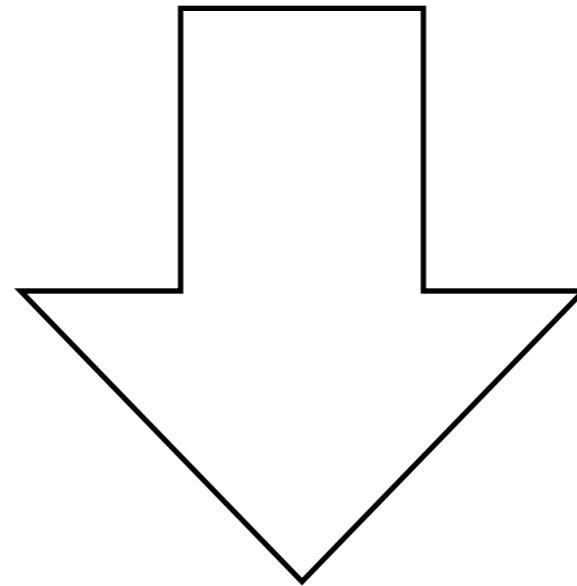
Eine Beispielapplikation



Convention over Configuration

Idee:

Halte dich an bestimmte Bedingungen ...



... erhalte Funktionalität ohne Konfigurationsaufwand.

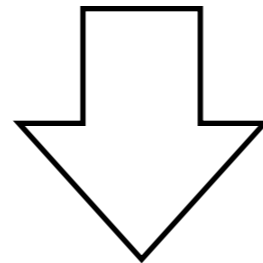
Namenskonventionen

- Beispiel: Entity-Typ „Vorlesung“
- Tabellename *underscored* im Plural:
vorlesungen (informatik_vorlesungen)
- *Model-Klasse* im *Mixed Case*:
Vorlesung (InformatikVorlesung)
- Obacht: englische Pluralbildung:
Vorlesung wird Vorlesungs
(vgl. Lecture, Lectures)
- Also: Entitäten in Englisch benennen

Schlüsselkonventionen

- Schlüsselkonvention:
synthetischer Schlüssel „id“ für jede Entität

Vorlesung: {[VorlNr, Titel, PersNr, ...]}

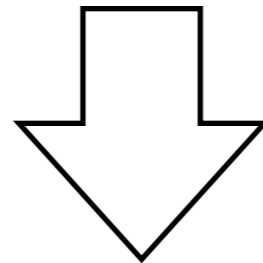


lectures: {[id, vorl_nr, titel, pers_nr, ...]}

Schlüsselkonventionen

- Konvention für Fremdschlüsselbeziehungen

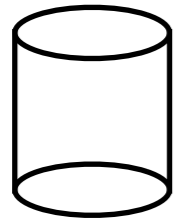
Vorlesung: {[VorlNr, Titel, PersNr, ...]}



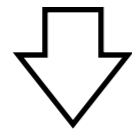
lectures: {[id, vorl_nr, titel, professor_id, ...]}

Model-Generierung

- Model-Klassen aus Datenbankschema ableiten



```
lectures: {[id, titel, ...]}
```



```
class Lecture < ActiveRecord::Base  
  ...  
end
```

```
$ rails generate model lecture
```

Objektrelationales Mapping

- Komfortabler Umgang mit Daten:

```
#Erzeuge neue Vorlesung:
```

```
new_vl = Lecture.new
```

```
#Fülle Daten
```

```
new_vl.titel = „Ruby on Rails“
```

```
new_vl.sws = 4
```

```
new_vl.professor_id = 1
```

```
#Speichern
```

```
new_vl.save
```

Objektrelationales Mapping

- Komfortabler Umgang mit Daten:

```
SHOW COLUMNS FROM lectures,  
synthetisiere Setter und Getter für alle  
Attribute
```

```
nutze Setter für gefundenes Attr. titel
```

```
nutze Setter für gefundenes Attr. sws
```

```
nutze Setter für gefundenes Attr. professor_id
```

```
INSERT INTO lectures (...) VALUES (...)
```

Don't Repeat Yourself

- Woher weiß Rails, welche Attribute ein bestimmtes Objekt hat?
- Attribute stehen bereits in der Datenbank

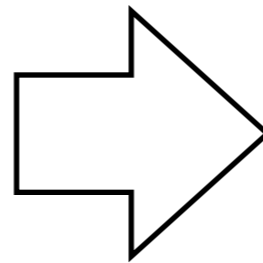
	Feld	Typ	Kollation	Attribute	Null	Standard
<input type="checkbox"/>	<u>id</u>	int(11)			Nein	<i>Kein</i>
<input type="checkbox"/>	vorl_nr	int(11)			Nein	<i>Kein</i>
<input type="checkbox"/>	titel	varchar(20)	latin1_swedish_ci		Ja	<i>NULL</i>
<input type="checkbox"/>	sws	int(11)			Ja	<i>NULL</i>
<input type="checkbox"/>	professor_id	int(11)			Nein	<i>Kein</i>

- „Wiederhole dich nicht selbst!“

Datenbankabstraktion, ORM

- CRUD-Operationen mit Klassen/Objekten kapseln
- CREATE (INSERT), READ (SELECT), UPDATE, DELETE

```
my_vl = Lecture.new  
my_vl = Lecture.first  
my_vl.save  
my_vl.delete
```



```
INSERT INTO lectures ...  
SELECT * FROM lectures  
UPDATE lectures SET ...  
DELETE FROM lectures ...
```

- Zur Erinnerung:
Funktionalität „umsonst“ wegen Konventionen

Datenbankabstraktion, ORM

- Komfortabler Umgang mit Daten:

```
#Besorge eine Vorlesung:
```

```
vl = Lecture.first
```

```
#Welcher Professor hält die Vorlesung
```

```
vl.professor
```

```
#=> <Objekt vom Typ Professor>
```

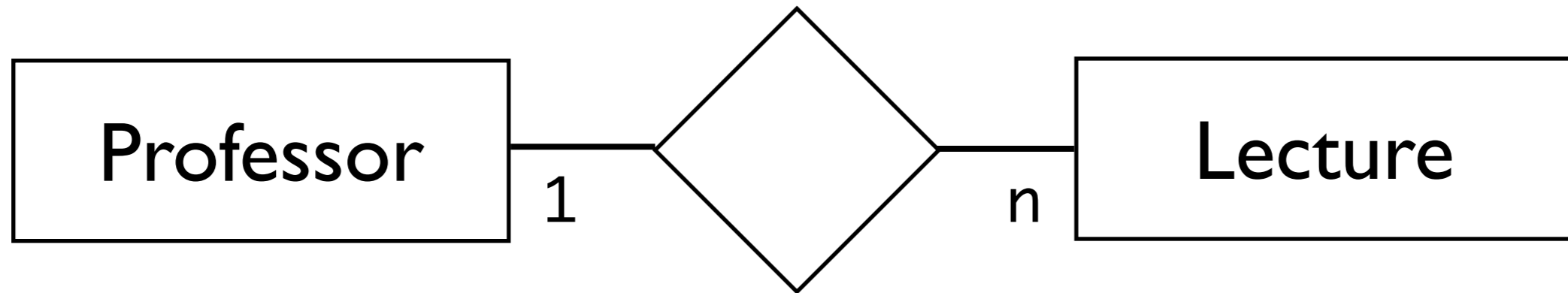
Datenbankabstraktion, ORM

- Komfortabler Umgang mit Daten:

```
SELECT * FROM professors WHERE  
id = [v1.professor_id],
```

Erzeuge neues Professor-Objekt aus dem
Ergebnis-Tupel

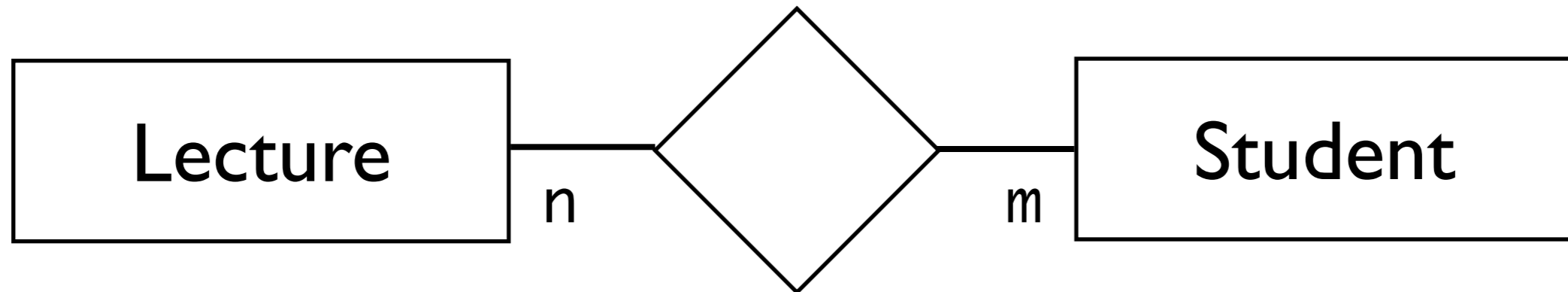
Model und Beziehungen



```
class Professor < ActiveRecord::Base
  has_many :lectures
end
```

```
class Lecture < ActiveRecord::Base
  belongs_to :professor
end
```

Model und Beziehungen



```
class Lecture < ActiveRecord::Base
  has_many_and_belongs_to_many :students
end
```

```
class Student < ActiveRecord::Base
  has_many_and_belongs_to_many :lectures
end
```

Model und Beziehungen

- n:m-Relationen-Modellierung in der Datenbank
- nicht zu „verfeinern“

```
lectures: {[id,titel,...]}
```

```
students: {[id,matr_nr,...]}
```

- „Join“-Table nötig, Konvention:

```
entität1_entität2: {[entität1_id, entität2_id]}
```

```
lectures_students: {[lecture_id, student_id]}
```

- Obacht: $L < S$, daher *nicht* students_lectures

Model und Beziehungen

- Erweiterungen von Model-Klassen um modelbezogene Logik:

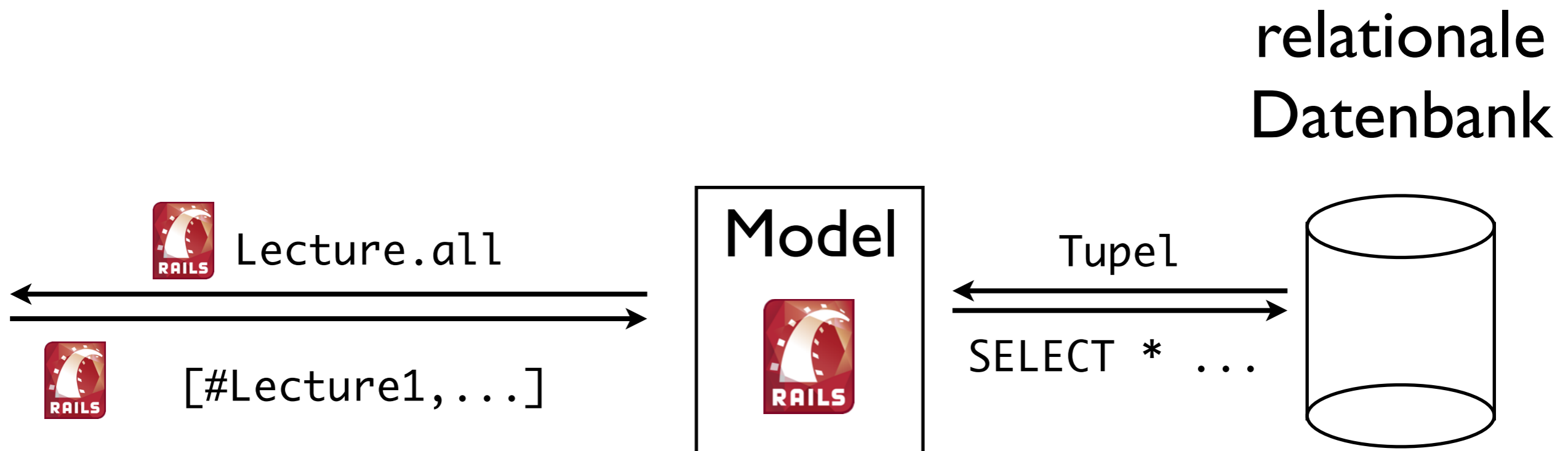
```
#liefert alle Professoren eines Studenten  
def professors
```

```
  self.lectures.collect { |l| l.professor }
```

```
end
```

Datenbankabstraktion - Zusammenfassung

- M in MVC - Rails Models
- generiert (DRY, Konventionen)
- Kapseln von Datenbankabfragen
- grundlegende CRUD-Operationen „kostenlos“

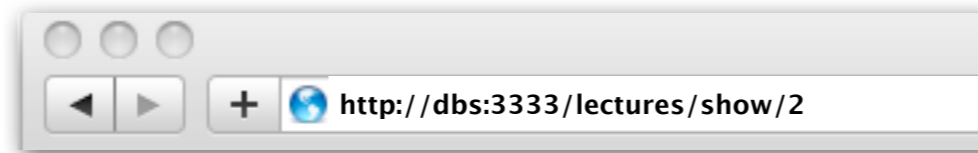


Datenbankabstraktion - Zusammenfassung

Demo

Controller

- C in MCV - Rails Controller
- Rails Applikation aus dem Browser aufrufen

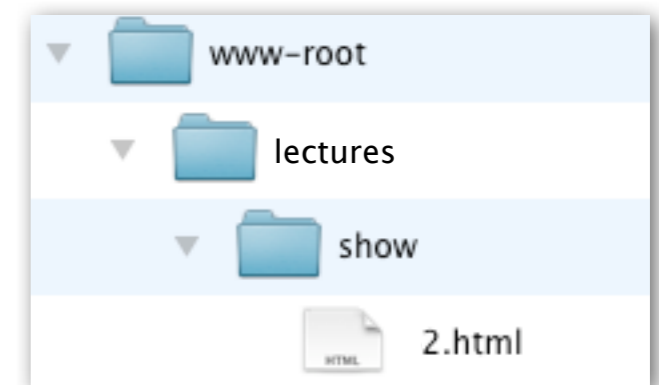


- Klassisch:

Server: dbs, Port: 3333

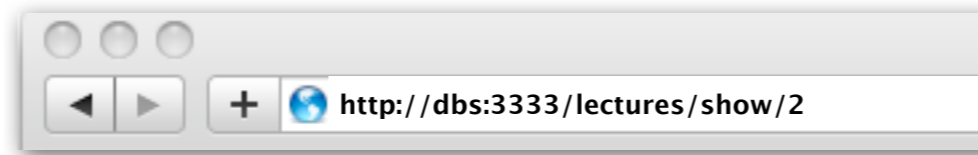
Aufgerufene Seite:

`www-root/lectures/show/2.html`

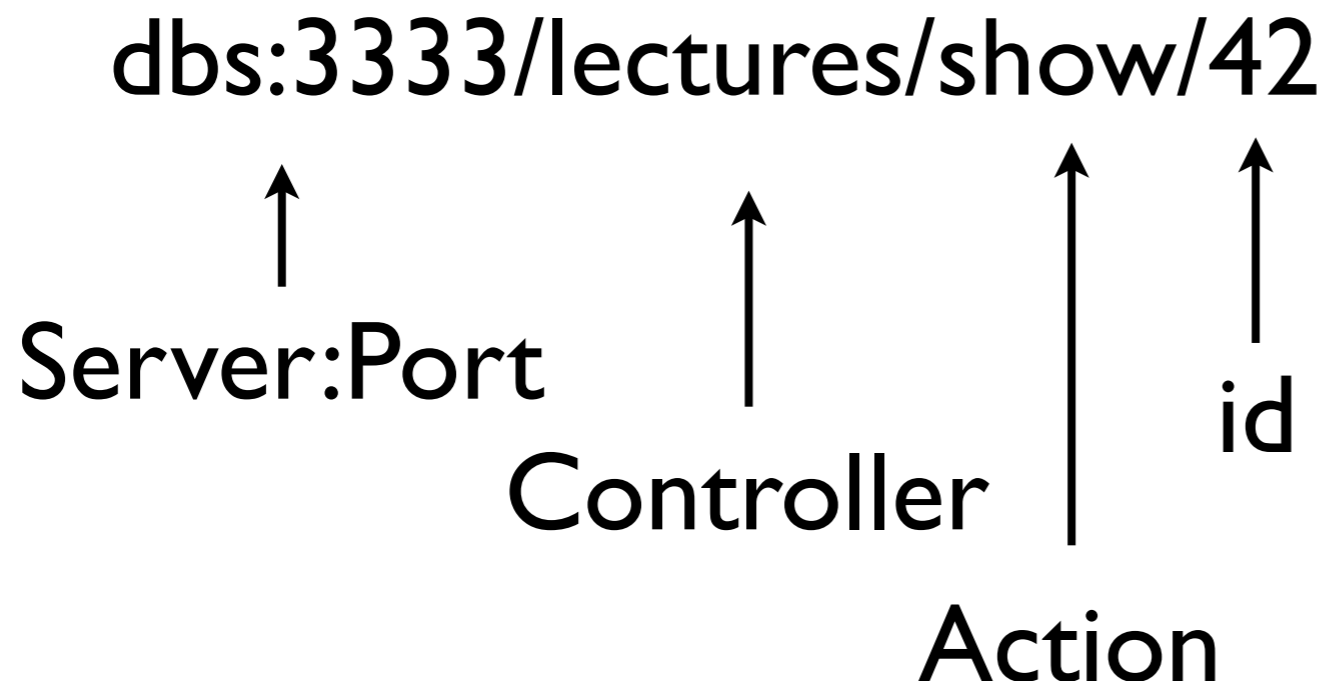


Controller

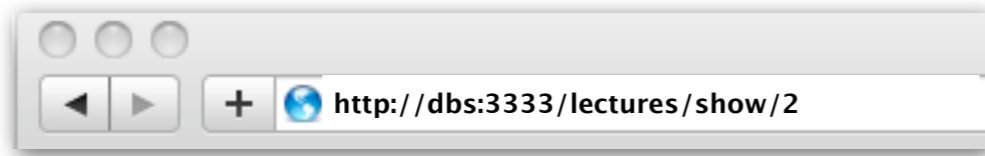
- C in MCV - Rails Controller
- Rails Applikation aus dem Browser aufrufen



- Rails (Konvention):

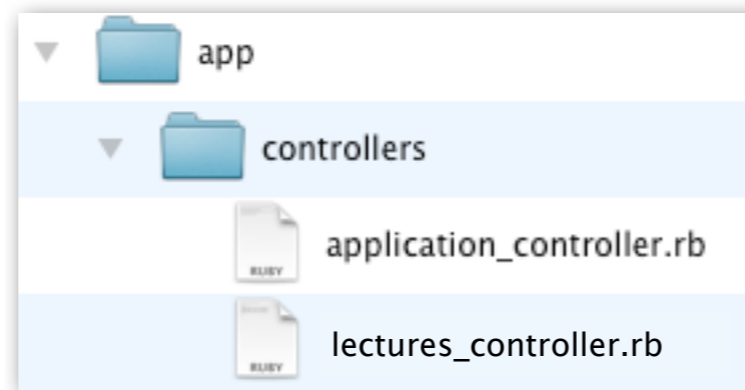


Controller

- 
- Instantiiere die Klasse `LecturesController`
- Rufe die Methode („Action“) `show` auf
- Übergebe der Methode im *Parameter-Hash* für die Variable `id` den Wert `42`

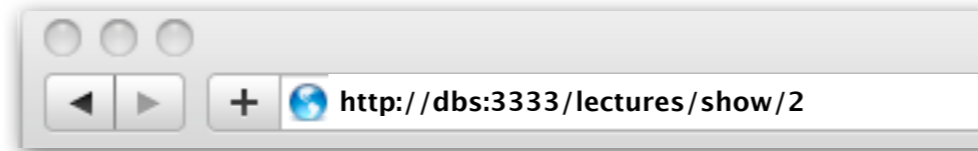
Controller

- Controller erzeugen
- `rails generate controller lectures`



```
class LecturesController < ApplicationController  
  
end
```

Controller



```
class LecturesController < ApplicationController

  def show

    id = params[:id] #enthält Wert 2
    my_vl = Lecture.find(id)
    #findet Lecture mit id 2

  end

end
```

Controller

```
class LecturesController < ApplicationController

  def show

    id = params[:id] #enthält Wert 2
    @my_vl = Lecture.find(id)
    #findet Lecture mit id 2

    render :controller => „lectures“,
           :action => „show“
  end

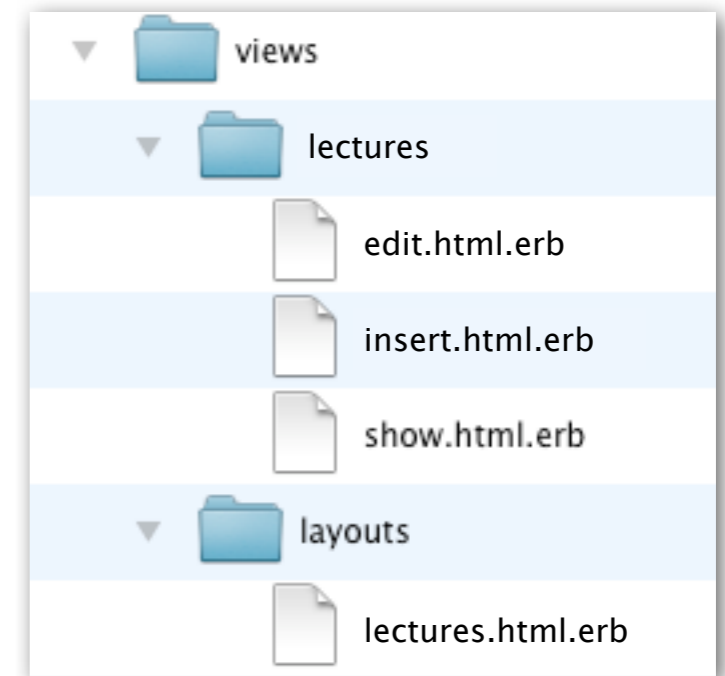
end
```

Views & Templates

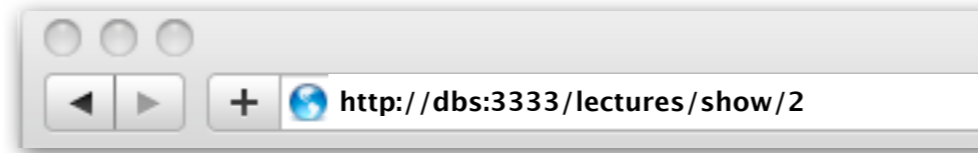
- V in MVC - Rails Views
- Konvention:

Ordner für Controller
Templates für Actions
Layouts für Controller

- Template-Sprache: (Embedded-)Ruby



Views & Templates



```
render :controller => „lectures“,  
       :action => „show“,
```

- Verarbeite Template für Action „show“
(Konvention: `show.html.erb` im Ordner `lectures`)
- Sende Ergebnis an Nutzer zurück

Views & Templates: Embedded Ruby

- Ausgebende und Nicht-Ausgebende Tags

```
<%= „Ich bin ein String!“ %>
```

```
<% „Ich bin auch ein String!“ %>
```

- Ausgabe

```
Ich bin ein String!
```

- Sinnvoll bei Blöcken

```
<% if(bedingung) %>
```

```
<%= gib_etwas_aus() %>
```

```
<% end %>
```

Views & Templates: Zugriff auf Variablen

- Zugriff auf Variablen

```
<%= lokal = @variable_aus_dem_controller %>  
<%= lokal %>
```

- Beispiel

```
<h1>Alle Vorlesungen</h1>  
<ul>  
<% @lectures.each do |lecture| %>  
  <li><%= lecture.title %></li>  
<% end %>  
</ul>
```

Views & Templates

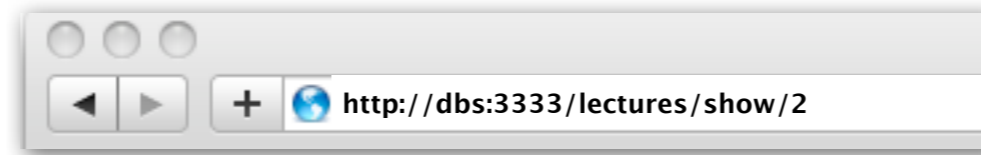
- Layout beinhaltet Rahmenstruktur
- Platzhalter für Inhalt des eigentlichen Templates

```
<html>
<head>
  <title>Lectures</title>
</head>
<body>

  <%= yield %>

</body>
</html>
```

Views & Templates



Request \updownarrow Response

Controller

Objekte zur
Anzeige

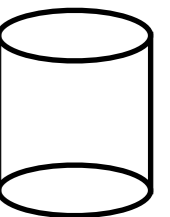
Anfrage

Rails-
Objekte

SQL

View(s)

Model(s)



DB



Views & Templates

Demo