7363 - Web-basierte Anwendungen
4750 – Web-Engineering

Eine Vertiefungsveranstaltung

Routing & Controller

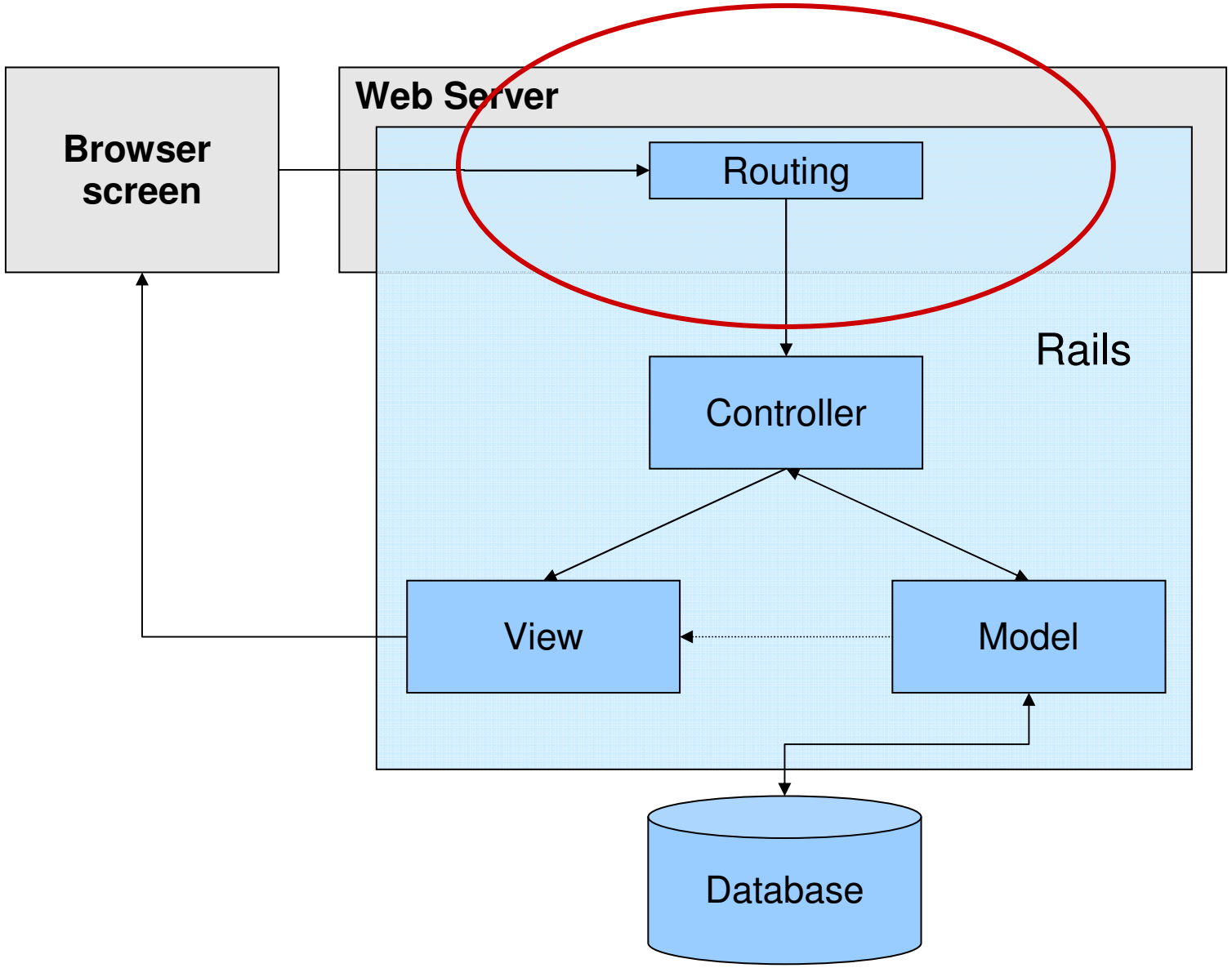
Routing

- *Routing*
  - URLs von HTTP *requests* müssen (zunächst von einem vorgeschalteten Web Server, dann) von Rails in Methodenaufrufe umgesetzt werden
  - Apache-Analogon: *Rewrite rules*, **`mod_rewrite`**
  - *Routing*-Regeln können komplex werden und benötigen daher Hilfsmittel zur Fehleranalyse (s.u.)

- *Routing*-Regeln
  - Rails verwaltet *Routing*-Regeln in **`config/routes.rb`**
  - Sie stellen einen Zusammenhang her zwischen Mustern in eingehenden URLs und daraus abgeleiteten Controller- sowie Methodennamen

- Informationsquellen
  - Kommentarblock & Beispiele in jeder Datei **`config/routes.rb`**
  - http://api.rubyonrails.org/ **ActionController:Routing**
  - http://guides.rubyonrails.org/routing_outside_in.html

Routing



Routing

- *Routing*-Priorisierung
  - Die erste „passende“ *Routing*-Regel wird genommen
  - Default-Routen, z.B. zur Reaktion auf nicht vorhandene Seiten, stehen daher am Dateiende („*catch-all*“-Einträge)
  - Vorsicht: Neue Einträge können bisher funktionierende verdecken, wenn sie zu früh eingefügt werden!

- Traditionelles *Routing*
  - Aus dem URL wird auf den zuständigen Controller sowie eine seiner Action-Methoden geschlossen.
  - Weitere URL-Angaben selektieren die gewünschten Daten.

- *RESTful routing*
  - Ein Blick in den BooksController

Routing

- *Routing*-Diagnose
  - Konsole starten

```
$ script/console
Loading development environment (Rails 2.2.2)
>>
```

  - Routen-Objekt erzeugen

```
>> rs = ActionController::Routing::Routes
#<…> …  (viel Text)
```

  - Routen anzeigen lassen

```
>> puts rs.routes
GET     /books/              {:action=>"index", :controller=>"books"}
GET     /books/.:format      {:action=>"index", :controller=>"books"}
POST    /books/              {:action=>"create", :controller=>"books"}
POST    /books/.:format      {:action=>"create", :controller=>"books"}
GET     /books/new           {:action=>"new", :controller=>"books"}
(usw.)
ANY     /:controller/:action/:id.:format/ {}
```

Routing

- *Routing*-Diagnose
  - Routen testen, *RESTful style*

```
>> rs.recognize_path "/books"
⇒{:action=>"index", :controller=>"books"}
>> rs.recognize_path "/books", :method => :post
⇒{:action=>"create", :controller=>"books"}
>> rs.recognize_path "/books/1", :method => :get
=> {:action=>"show", :controller=>"books", :id=>"1"}
>> rs.recognize_path "/books/show/1", :method => :put
=> {:action=>"update", :controller=>"books", :id=>"1"}
```

  - Hinweise
    - Die Dokumentation lässt noch einiges zu wünschen übrig. Auf den Zusatz „:method => xxx“ stieß ich weder im Rails-Buch noch in der Online-Doku, sondern durch Raten in Diskussionsforen…

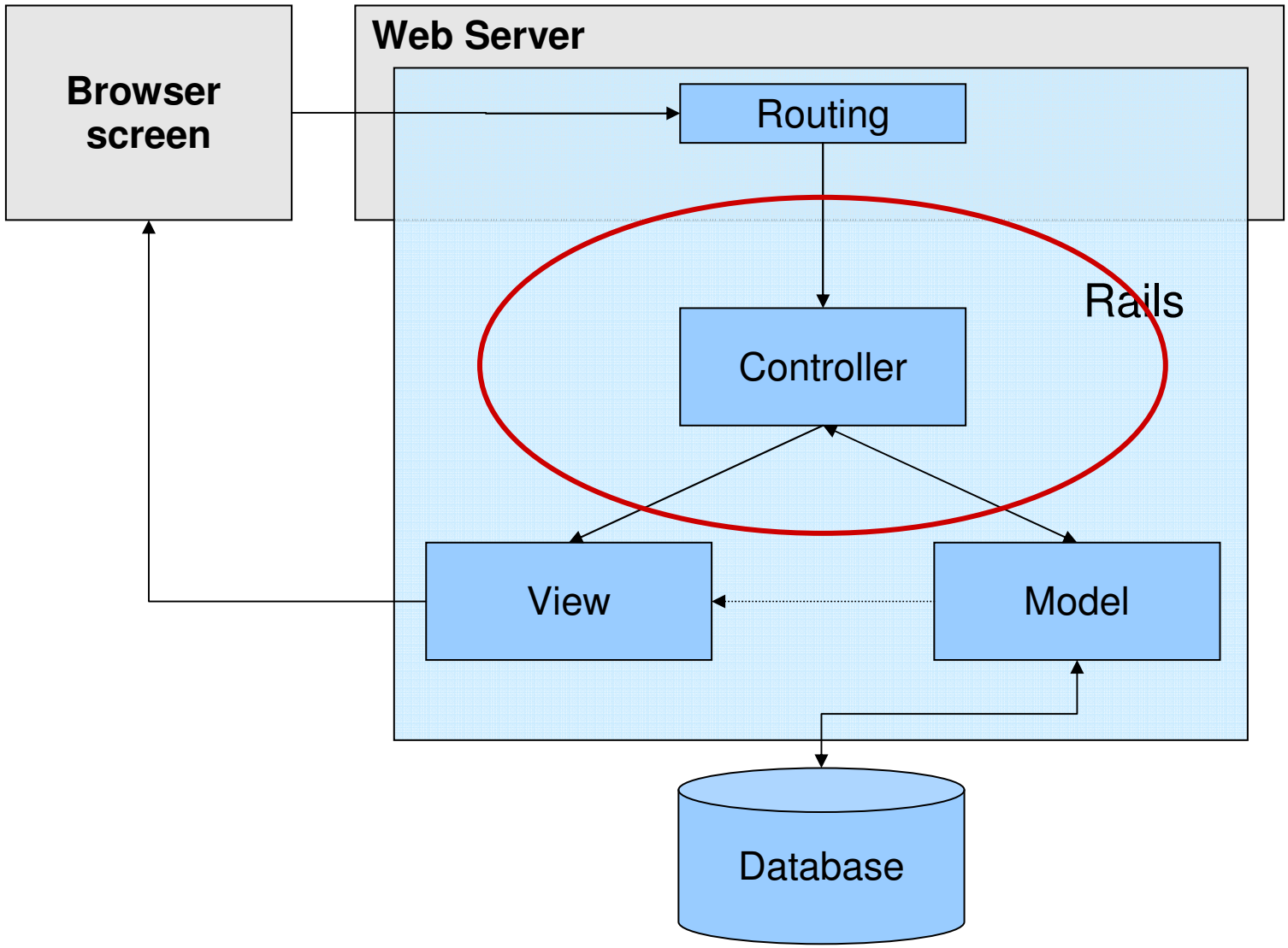
Routing

- *Routing*-Diagnose
  - Umkehrung: Routen generieren

```
>> rs.generate :controller=> :books
⇒ "/books"
>> rs.generate :controller=> :books, :action=> :show, :id=>1
⇒ "/books/1"
>> rs.generate :controller=> :books, :action=> :update,:id=>1
⇒ "/books/1"
```

  - Hinweise
    - Die Methode „generate“ dient nur zu derartigen Testzwecken! Verwenden Sie im Anwendungsteil darauf aufbauende Methoden wie
      - **url_for**
      - **link_to**
  - Online-Demo

Controller



Controller

- Aufgaben eines Controllers
  - Organisation der Model-Daten
    - Bereitstellung von Daten für den View
    - Speichern von Daten, z.B. aus User input
  - **Reaktion auf Anfrage einleiten**
    - **Rendering-Details (View)**
  - Entscheidungen bez. Weiterleitung
  - Filter
    - Z.B. Zugriffskontrolle
  - Umgang mit HTTP Header-Daten
  - …

Controller

- Der einfachste Controller

```
class FooController < ApplicationController
end
```

  - Wirkung nur durch Default-Regeln!
    - Ein „GET /foo/bar/“ bewirkt Suche nach einer public-Methode „bar“ in Datei app/controllers/foo_controller.rb, Controller FooController
    - Falls keine solche Methode vorhanden, wird implizit und ohne *model*-Daten gerendert und dazu die view-Datei app/views/foo/bar.html.erb gesucht.
    - Demo mit Projekt „lib“ und einer geeigneten Route (s.o.)

Controller

- Der elementare, implizite Controller

```
class FooController < ApplicationController
  def bar
  end
End
```

  - Wirkung nur Default-Rendering
    - Die Methode „bar“ wird gefunden.
    - Da sie kein Rendering auslöst, wird an ihrem Ende automatisch eins ausgelöst

Controller

- Der elementare, explizite Controller

```
class FooController < ApplicationController
  def bar
    render                  # Wählt app/views/foo/bar.html.erb
  end
end
```

– Wirkung:
  - Rendering wird explizit auslöst, auch ein anderes Template wäre möglich

```
class FooController < ApplicationController
  def bar
    render :template => "foo/bar"  # Analog, nur explizit
  end
end
```

Controller

- Controller für mehrere Formate / mit Web Service-Unterstützung

```
class BooksController < ApplicationController
  def index
    @books = Book.find :all
    respond_to do |format|
      format.html # render implizit, views/books/index.html.erb
      format.xml  { render :xml => @books.to_xml }
      format.js   # render implizit, views/books/index.js.erb
    end
  end
end
```

  - Wirkung:
    - Bei den Formaten html und js wird implizit die zuständige view-Datei gerendert
    - Das XML-Format wird direkt ausgeliefert über die Option „:xml“ von „render“
    - Offenbar gibt es eine generische Umwandlung von Objekten in XML-Darstellung.

Varianten der Methode `render`

- Hauptaufgabe einer Controller-Methode ist, auf eine Anfrage (*request*) zu reagieren (*response*), meist per „rendering“ einer Schablone

- Variante `:nothing`
  - Schließt einen HTTP request ab, ohne Daten zu liefern
  - Nützlich etwa im Kontext von AJAX *requests*
  - Vielleicht die bessere Alternative: `head` statt `render`

```
def xhr_reply
  mache_etwas_mit(params)
  render :nothing => true # Formaler Abschluss des HTTP requests
end
```

Controller

- Variante `:text`
  - Umgehen der Schablonen und Layouts, direktes Ausliefern des übergebenen Texts, z.B. für sehr kurze Antworten
  - Ohne Layout ist Content-type = „text/plain“

```
def show
  render :text => "Sorry - noch nicht implementiert!"
end

def boom
  render :text => "Boom - an error", :status => 500
end

def hello
  render :text => "<h1>Hello world!</h1>", :layout => true
end
```

Controller

- Variante **`:template`**
  - Umlenkung auf alternatives Template, falls Default nicht gewünscht
  - Relativ zum Pfad app/views und mit Auto-Vervollständigung (Bsp.: .html.erb)
  - Hinweis: Im Normalfall verwendet man **`render :action`**

```
def bar
  render :template => "foo/baz",
         :locals => {:book => Book.find 1} # Lokale Var. in view
end
```

- Variante **`:file`**
  - Angabe des absoluten Dateipfades eines Templates auf dem Server

```
def foo
  render :file => "/path/to/some/template.html.erb"
end
```

Controller

- Variante `:inline`
  - Nutzung der Template-Technik ohne Template-Datei
  - Der Template-Inhalt wird direkt im Controller („*inline*“) mitgegeben
  - Hinweis: Sparsam verwenden – *Inlining* widerspricht dem MVC-Muster!

```
def bar
  # Schabloneninhalt für ERb als String übergeben,
  # ggf. lokale Variablen setzen:
  @book = Book.find :first
  render :inline => "<%= 'Titel des ersten Buchs:', title %>",
                        :locals => { :title => @book.title }
end
```

Controller

- Variante `:action`
  - Umlenkung auf alternatives Template, falls Default nicht gewünscht
  - Relativ zum Pfad app/views und mit Auto-Vervollständigung (Bsp.: .html.erb)
  - Hinweis: Im Normalfall verwendet man **`render :action`**

```
def bar
  # Verwendet Template „baz“ des aktuellen Controllers,
  # Layout wird ersetzt durch „other.html.erb“ in „layouts“
  render :action => "baz", :layout => "other"
end
```

Controller

- Variante `:xml`
  - Zur Auslieferung beliebiger XML-Dokumente
  - Ebenfalls zur persistenten Darstellung von Rails-Objekten
  - HTTP Header Content-type = „application/xml“

```
def foo
  render :xml => generate_svg()
end

def bar
  render :xml => @books  # ruft implizit @books.to_xml auf
end
```

Controller

- Variante `:json`
  - Zur persistenten Darstellung von Rails-Objekten mit JSON (analog XML)
  - HTTP Header Content-type = „application/json“

```
def foo
  render :json => { :some_key => "some_value" }.to_json
end
```

- Variante `:js`
  - Direkte Auslieferung von JavaScript-Code. Ausführung auf Client-Seite?
  - HTTP Header Content-type = „text/javascript“

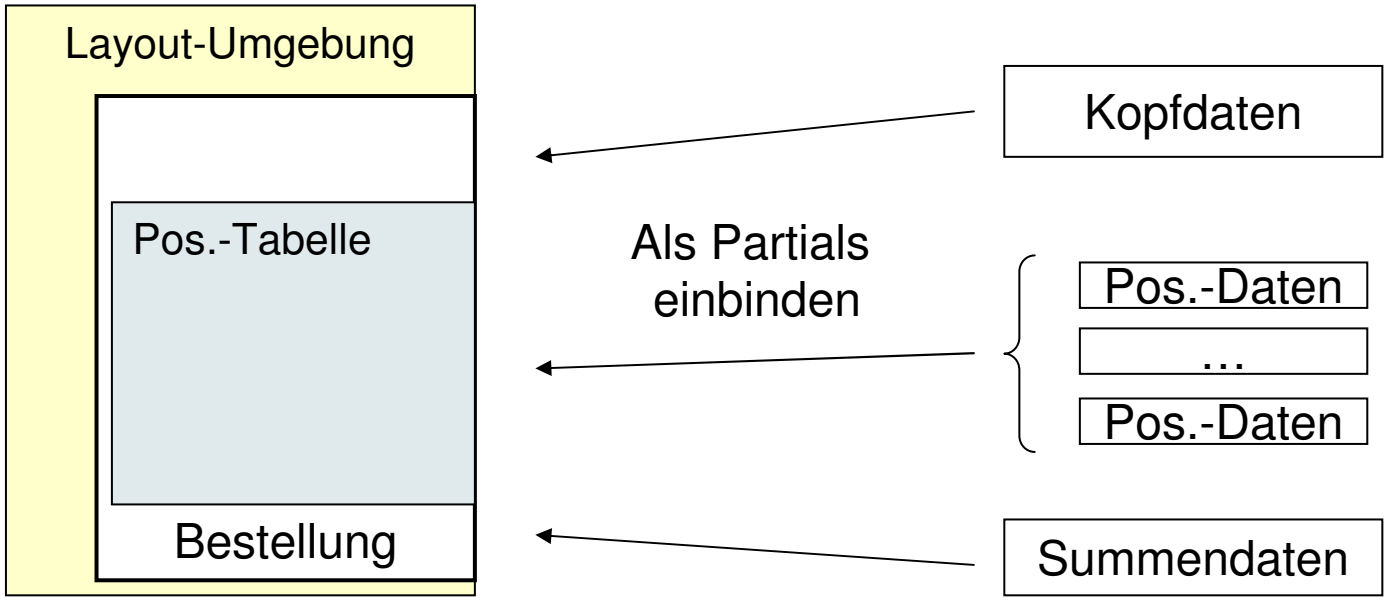
```
def hi
  render :js => "alert('hello')"
end
```

Controller

- Einschub: *Partials*
  - Eigentlich beruht HTTP auf dem Konzept, dass jeder Request ein vollständiges Dokument (HTML-Seite, XML-Dokument, Datei etc.) erhält.
  - Das resultierende Konzept 1 Request = 1 Schablone = 1 HTML-Seite wird aber bereits vom Layout-Ansatz durchbrochen, wenn auch „nur“ im Include-Stil
  - Partials gehen noch einen Schritt weiter und ermöglichen die Zerlegung einer HTML-Seite in wiederverwertbare Bausteine
    - Beispiel: Die Schablonen zu „edit“, und „new“ sind einander sehr ähnlich. Auslagerung der Gemeinsamkeiten in ein *Partial* wäre sehr „DRY“!
  - Partials sind auch als Schleifen-Kerne verwendbar, etwa zur Kapselung des Aufbaus einer Tabellenzeile.
  - Ausblick: Die Ajax-Technik ermöglicht auch die Aktualisierung bestehender HTML-Seiten im Client (Browser), z.B. durch Einfügen neuer Tabellenzeilen. Geschickt gewählte Partials sind dann doppelt verwendbar:
    - als Template-“Unterprogramme“ beim Seitenaufbau
    - als Kern einer Ajax-Servermethode bei der dynamischen Seiten-Aktualisierung
  - Kaskadierbarbeit: Partials können auch Partials aufrufen / includieren

Controller

- *Partials*-Beispiel: Bestelldaten
  - Controller bindet (max. ein) Layout ein
  - Template organisiert den Aufbau einer Bestellung
    - Kopfdaten → Kopfdaten-Partial einbinden
    - Positionsdaten → Tabelle anlegen,
      Schleife bilden: Für jede Position: Positionsdaten-Partial „aufrufen“
    - Summendaten → Summendaten-Partial einbinden



Controller

- Variante `:partial`
  - Nutzung eines Partials anstelle eines vollständigen Templates
  - Liefert nur ein HTML-Fragment, Layout-Kontext ist abgeschaltet.
  - Dennoch gilt auch hier: **Nur ein render-Aufruf pro Request möglich!**
  - Typische Nutzung in Methoden, die auf einen XMLHttpRequest antworten

```
def bar
  # Verwendet Partial „baz“ und erwartet es in Datei
  # views/foo/_baz.html.erb („_“ beachten!)
  render :partial => "baz"
end
```

Controller

- Variante `:update`
  - Zur inline-Erzeugung von JavaScript-Code, ersetzt .js.erb-Templates in einfachen Fällen
  - JavaScript-Erzeugung wird (voraussichtlich) später besprochen, daher sei diese Variante hier nur der Vollständigkeit halber erwähnt und mit einem Beispiel aus der Online-Dokumentation unkommentiert beschrieben.

```
def bar
  # Verwendet Template „baz“ des aktuellen Controllers,
  # Layout wird ersetzt durch „other.html.erb“ in „layouts“
  render :update do
    page.replace_html 'user_list', :partial => 'user',
                                   :collection => @users
    page.visual_effect :highlight, 'user_list'
  end
end
```

Generatoren von JavaScript-Code

ID des HTML-Elements

Controller

- Binärdaten zurücksenden: Die Methode **`send_data`**
  - Anstelle von **`render`** verwenden, zum Senden beliebiger Binärdaten
  - MIME-Type-Angabe nicht vergessen!

```
def bar
  png_data = string_to_png("mail@mydomain.xy")
  send_data png_data,   :type => "image/png",
                        :disposition => "inline"
end
```

Controller

- Datei-Inhalte zurücksenden: Die Methode **`send_file`**
  - Anstelle von **`render`** verwenden, zum Senden beliebiger Binärdaten

```
def bar
  send_file "/files/some_song.mp3", :type => "audio/mpeg"
  headers["Content-Description"] = "Musik" # Opt. Ergänzung
end
```

  - Optionen:

`:buffer_size`	`number`	Blockgröße, wenn Streaming aktiv
`:disposition`	`string`	'inline' oder 'attachment' (Default)
`:filename`	`string`	Suggerierter Dateiname, z.B. beim Abspeichern
`:status`	`string`	Default ist '200 OK'
`:stream`	`true | false`	
`:type`	`string`	*Content type*, *Default* = 'application/octet-stream'

  - Bemerkungen:
    - **`headers`** demonstriert expliziten Zugriff auf einen HTTP *header* in der Antwort

Controller

- **Die Controller-Umgebung**
  - Rails stellt eine Reihe von *Getter*-Methoden zur Kommunikation mit den Elementen eines HTTP *request/response*-Paares bereit.

- Die Methoden zum Zugriff auf diese Umgebung

action_name

- Der Name der aktuell bearbeiteten action-Methode

cookies

- Ein Hash-artiges Objekt. Nutzung in einer action-Methode etwa so:

```
cookies[:time_of_visit] = Time.now.to_i   # Cookie setzen
t = Time.at(cookies[:time_of_visit])      # Cookie lesen
```

headers

- Ein Hash der in der Antwort verwendeten HTTP-Header
- Bem.: „cookie“ nicht explizit setzen!

method

- Liefert die Request-Methode. **`:delete, :get, :head, :post, :put`**

Controller

- Die Methoden zum Zugriff auf diese Umgebung (Forts.)

params

- Ein Hash-artiges Objekt, das die Request-Parameter aus einem Formular bzw. URL sowie Pseudo-Parameter aus dem Routing bereitstellt. Als Keys sind sowohl Strings als auch (bevorzugt) Symbole zulässig:

```
params['id']        #       '1'
params[:id]         #       '1' (gleiches Ergebnis)
```

- „params“ wird von Controllern häufig verwendet!

request

- Das Objekt, das den eingetroffenen Request repräsentiert. Attribute:

domain	Die zwei letzten Teile des Host-Namens
remote_ip	Die IP in String-Form. Mehrere möglich bei Proxies
env	**request.env['HTTP_ACCEPT_LANGUAGE']** # Lese-Bsp.
method	Die Request-Methode. **:delete, :get, :head, :post, :put**
delete?, get?, ?head?, post?, put?	**true | false**
xml_http_request? oder **xhr?**	**true** falls **Ajax call**

Controller

- Die Methoden zum Zugriff auf diese Umgebung (Forts.)

response

  - Das Response-Objekt. Es wird normalerweise von Rails aufgebaut und nicht „von Hand“ verändert

session

  - Ein Hash-artiges Objekt, das die Verwaltung von Sessions sehr vereinfacht
  - Siehe separate Folie(n) dazu

- Helfermethoden

logger (Aus ActionPack)

  - Ein einfacher integrierter Logging-Mechanismus, der überall in Rails-Anwendungen verwendbar ist – so auch in Controller-Methoden
  - Protokolliert in **`log/development.log`** bzw. **`log/production.log`**
  - Beispiele

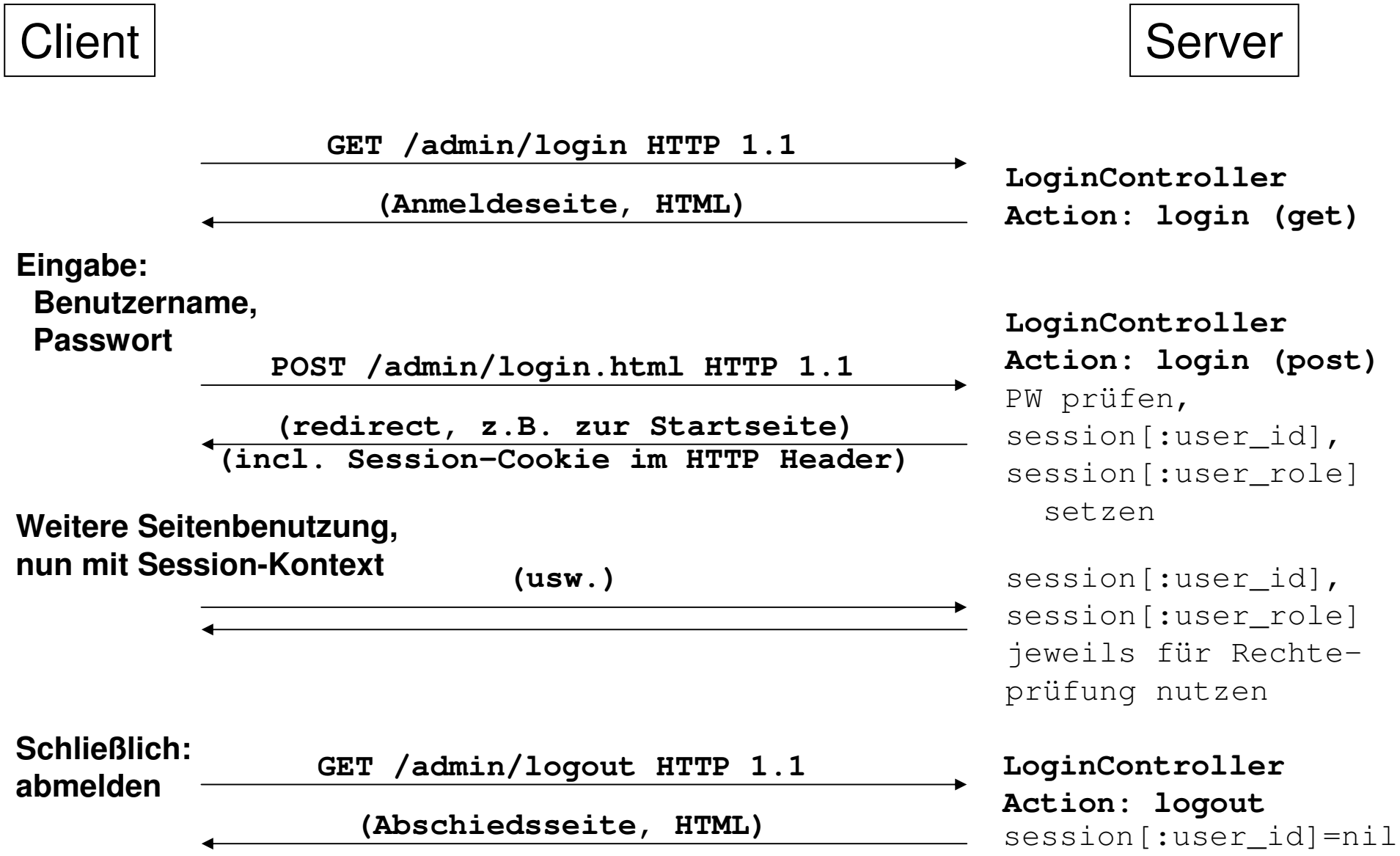
```
logger.info  " Das hier sieht seltsam aus ... "
logger.warn  " Vorsicht, Gefahr! "
logger.error " Zu spaet, nun hat's geknallt. "
logger.fatal " Zwecklos – ich gebe auf! "
```

Controller

- **Session-Verwaltung**
  - Rails vergibt eine Session ID und speichert sie in einem Cookie mit dem Schlüssel **`_session_id`**
  - „session“ ist ein Hash-artiges Objekt. Für jede Session ID wird ein Datensatz vorgehalten
  - Was man in „session“ schreibt, wird serialisiert und persistent auf dem Server gespeichert. Beim nächsten Request wird „session“ rekonstruiert.
    - Typische Nutzung: Für Daten (Bsp.: account, Rolle) angemeldeter Benutzer

- Konsequenzen
  - Man kann alle (serialisierbaren) Objekte in „session“ speichern
  - Wegen des wiederholten Aufwands zur (De-)Serialisierung sollten keine großen Datenmengen gespeichert werden. Diese hält man direkt in der DB vor und referenziert sie von der Session aus.
  - Kritische oder auch von außen veränderbare Daten gehören ebenfalls in die Datenbank

Sessions und Benutzerverwaltung



Controller

- Problem in der Praxis
  - Session-Objekte können veralten!
    - Beispiel: Ihr Besucher kommt nach 2 Tagen wieder, seine Session wird rekonstruiert. Die in seiner Session gespeicherten Objekte entsprechen dem Versionsstand seines letzten Besuchs. – Inzwischen haben Sie das DB-Schema bzw. einige Modelle aber geändert …
  - Auswege:
    - Session-Laufzeiten auf das notwendige Minimum beschränken
    - Benutzer zum „Ausloggen“ (Schließen einer Session) auffordern
    - Session-Altbestände regelmäßig löschen

- **Notwendigkeit von Sessions?**
  - Widerstreben Sie der Versuchung, *immer* eine Session „mitlaufen“ zu lassen. Sessions verursachen nicht nur Serverlast, sondern setzen Cookies voraus – Sie könnten Anwender verlieren, die diese standardmäßig deaktiviert haben.

Controller

- Abhilfen auf Code-Ebene
  - Session-Daten sollte man nicht länger als nötig vorhalten und ggf. eine Session löschen.

```
class SessionController < ActionController::Base
  # andere actions …
  def logout
    reset_session
  end
end
```

  - Session-Unterstützung nur bei echtem Bedarf (selektiv) einschalten:

```
class ProductsController < ActionController::Base
  session :on, :only => [:create, :update]
end
```

Controller

- Session-Optionen
  - Es gibt eine Reihe von Optionen, mit denen man Einfluss auf die Session-Verwaltung nehmen kann. Sie hängen zusammen mit dem Aufbau des *session cookies*:
    - session_domain, session_id, session_key, session_path, session_secure, new_session, session_expires
  - Nehmen Sie darauf Einfluss, wenn sich mehrere Rails-Anwendungen einen Host teilen (damit sich die Anwendungs-Sessions nicht gegenseitig stören). Sie können z.B. auch den Betrieb von Sessions auf https-Zugriffe einschränken:
    - In der Environment-Datei tragen Sie z.B. ein:

```
ActionController::Base.session_options[:session_key]="app01"
ActionController::Base.session_options[:session_secure]=true
```

Controller

- *Session Store*-Optionen
  - Rails bietet mehrere Möglichkeiten zur persistenten Speicherung von Session-Daten. Alle haben ihre Vor- und Nachteile.
  - Sie können zwischen diesen wählen oder sogar eigene implementieren.
- Auswahl:

```
ActionController::Base.session_store = xxx
```

- Eingebaut (xxx =)
  - :p_store / CGI::Session::PStore (Default lt. Buch; ggf. :tmpdir anpassen!)
  - :active_record_store (empfohlen)

```
$ rake db:sessions:create          # Erzeugt DB-Tabelle „sessions“
```

  - :cookie_store (Default lt. Website)
  - :drb_store
  - :mem_cache_store
  - :memory_store
  - (:database_manager => CGI::Session::FileStore)

Controller

- *Session Store*-Optionen
  - Einzelheiten später bzw. nachlesen
    - Online-Quelle: ActionController::SessionManagement::ClassMethods
      (leider sehr dürftig)
  - Wichtig wird die Session-Verwaltung beim *performance tuning*, insb. wenn man mehrere Applikationsserver zur Lastverteilung betreibt
  - Je nach Store-Option sind Strategien zum Aufräumen (Löschen veralteter Session-Daten) erforderlich – ohne laufende Sessions zu stören…
- Tipp
  - Wer *session management* verstanden hat, versteht bereits wesentliche Teile von Web-basierten Anwendungen. Dieses Thema ist daher besonders prüfungsrelevant!

Controller

- Der „*flash*“ (Teil der Session-Daten)
  - Rails leitet oft die Kontrolle zwischen „*actions*“ hin und her. Dazwischen liegt meist ein *Response*-Schritt sowie ein neuer *Request* durch den Browser (UA).
  - Das Problem: Zwischenergebnisse wie z.B. Attribute gehen dabei verloren!
  - Im *Hash*-artig organisierten „*flash*“ kann man bequem Zwischenergebnisse von einer „*action*“ in die nächste hinüberretten – aber nicht weiter, denn:
  - Der *flash* wird nach jedem *Request* wieder gelöscht!

- Nutzung
  - Weiterleiten aufwändig berechneter Objekte (*Overhead* für persistente Speicherung in Session-DB muss sich lohnen!)
  - Vormerken von kurzen Fehlertexten, die der Anwender im aktuellen View oder dem nächsten sehen soll.
    - Nicht verwechseln mit dem Logger, dessen Daten nur Admins sehen!

Controller

- Beispiel
  - Der folgende Code wurde für unser „Lib“-Projekt automatisch erzeugt. Er stammt aus BooksController:

```
def create
  @book = Book.new(params[:book])
  respond_to do |format|
    if @book.save
        flash[:notice] = 'Book was successfully created.'
        format.html { redirect_to(@book) }
        format.xml  { render :xml => @book,
                             :status => :created,
                             :location => @book }
    else
        format.html { render :action => "new" }
        format.xml  { render :xml => @book.errors,
                             :status => :unprocessable_entity }
    end
  end
end
```

Controller

- Beispiel (Forts.)
  - **`redirect_to(@book)`** leitet weiter an **`show()`**
  - Im Layout von „show“ findet sich folgender Code:

```
<body>
  <p style="color: green"><%= flash[:notice] %></p>

  <%= yield  %>
</body>
```

  - Wirkung:
    - Einen Zyklus weiter (der Client/Browser hat inzwischen einen neuen Request „show“ ausgelöst) können wir noch auf unsere Notiz zugreifen!

Controller

- Filter
  - Aspekte-orientiertes Programmieren: Controllermethoden (*actions*) in Rails besitzen *hooks* zu Beginn und am Ende, in die man Filtermethoden einklinken kann. Man unterscheidet
    - **before filter**
    - **after filter**
    - **around filter**
  - Filtermethoden werden nur „angemeldet“ in einem deklarativen Stil.

```
class SomeController < ApplicationController
  before_filter :authorize, :except => :index, :show
  # …
end
```

  - Wirkung hier:
    - Alle *actions* des Controllers außer „index“ und „show“ (nur Lesezugriffe) durchlaufen erst eine (meist private) Methode „authorize“
    - Diese kann z.B. im Fehlerfall eine Meldung per *flash* und ein **`redirect_to`** bewirken.

Controller

- Filter
  - Quellcode-Beispiel:
    Authentifizierung und Autorisierung mit Filtern im Projekt AoR