7363 - Web-basierte Anwendungen
4750 – Web-Engineering

Eine Vertiefungsveranstaltung

Performance-Aspekte

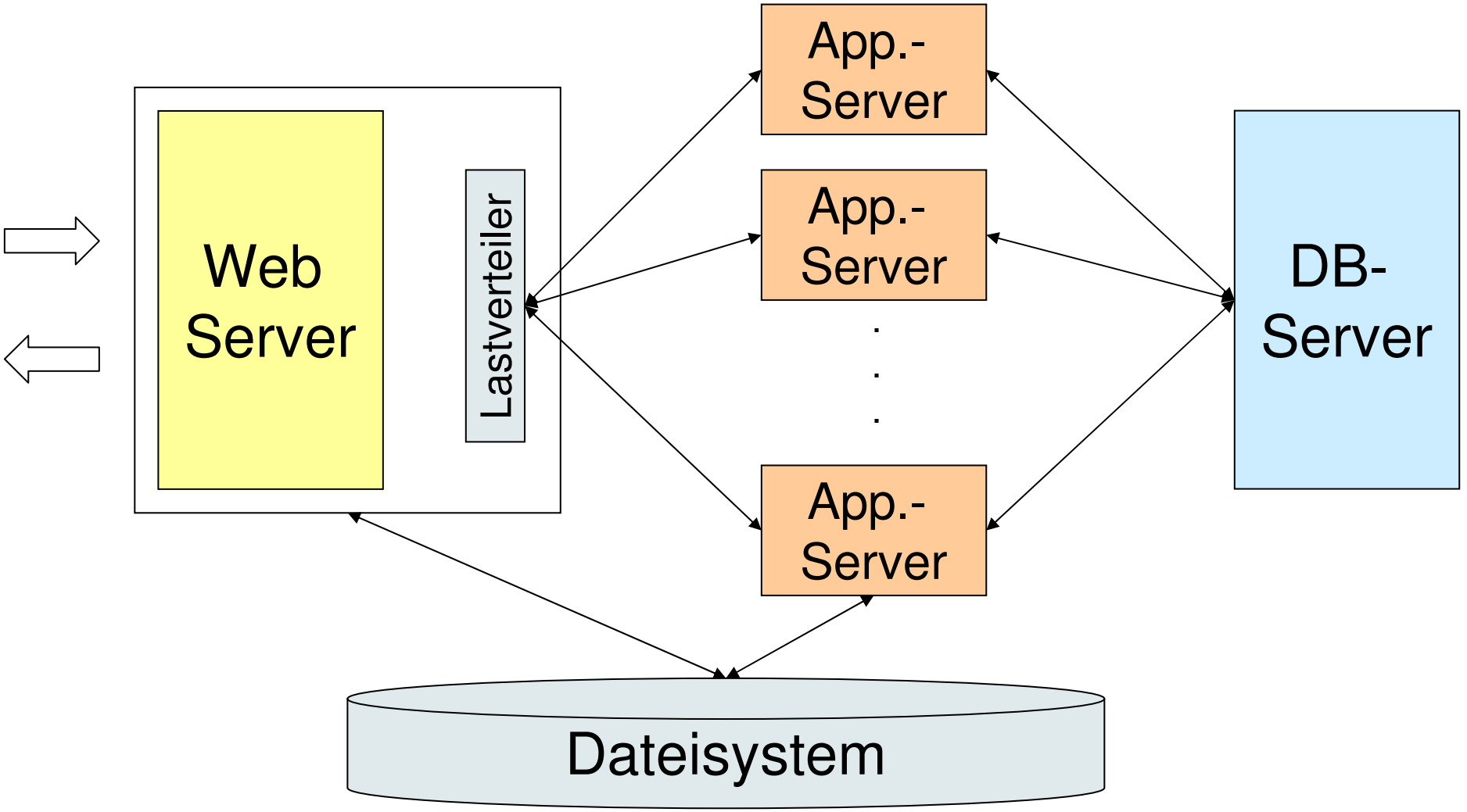
Caching: 3 Stufen; Client-Einbeziehung

App-Server

DB-Schicht

Performance von Web-Anwendungen

- Typische Architektur einer größeren Web-Anwendung:



Performance von Web-Anwendungen

- Wünsche aus Anwendersicht
  - Server jederzeit verfügbar
  - Sofortige Antworten
  - Sicherheit der eigenen Daten
- Wünsche aus Betreibersicht
  - Geringe Hardware- und Betriebskosten
  - Ausfall- und Daten-Sicherheit
- Grundlegende Annahmen
  - Was schon beim Client ist, muss nicht mehr ausgeliefert werden
  - Auslieferung statischer Inhalte schneller als deren dynamische Ermittlung
  - Weniger Datenbankzugriffe = bessere Antwortzeiten

- Nutzung der Client-Caches
  - Unterstützen Sie die für clientseitiges Caching erforderlichen HTTP *header*
    - Cache-Control
    - Last-modified
    - ETag
  - (Beispiele)

Performance von Web-Anwendungen

- Auslieferung statischer Inhalte direkt durch den Web Server
  - Durch geschickte Konfiguration übernimmt bereits der Web Server die Auslieferung statischer Inhalte. Dazu zählen
    - Bild-Dateien u.a. Multimedia-Inhalte
    - JS-Bibliotheken, CSS-Dateien u.a. im HTML-Header angesprochene Dateien
    - Gegenstände von File Downloads wie ZIP-Dateien, MP3- oder Video-Dateien
    - Statische HTML-Dateien (siehe auch „caching“)
  - *Multithreading* im Web Server ermöglicht das parallele Ausliefern mehrerer Dateien. Gerade bei längeren Downloads werden dadurch wertvolle App.-Server-Ressourcen geschont!
  - Vorhalten entsprechender Dateien im Dateisystem schont den Datenbank-Server
    - **(Techn. Details der Konfiguration?)**

Performance von Web-Anwendungen

- Generierung statischer Inhalte direkt durch die App.-Server
  - Die ressourcen-intensive dynamische Ermittlung von Anwendungsseiten kann oftmals vermieden werden
  - Die Anwendung benötigt Hinweise vom Entwickler, welche Seiten sich für Generierung statischer Versionen eignen und wann statische Inhalte erneuert werden müssen
  - Beispiel:
    - index-Views eines Modells ändern sich erst wieder, wenn die Aktionen „new“ bzw. „create“ oder (meistens) „update“ verwendet werden
- Strategie daher („serverseitiges Caching“):
  - Potenzielle statische Seite nach erster Erzeugung im Dateisystem zwischenspeichern
  - Diese statische Kopie bei weiteren Anfragen ausliefern!
    - Routing muss dies unterstützen, besser noch:
    - Delegation an den Web Server ohne Umweg über die Anwendung?
  - Statische Kopie bei Änderungen entfernen bzw. erneuern

Performance von Web-Anwendungen

- Serverseitiges Caching und Rails
  - Rails unterstützt gleich drei derartige Caching-Strategien
    - ***page caching, action caching*** und ***fragment caching (+ body caching?)***

- ***Page caching***
  - Per Deklaration „***caches_page***“ in Controllern wird veranlasst, eine statische Kopie beim erstmaligen Aufrufen eines entsprechenden Views anzulegen. Beispiel:

```
class BookController < ApplicationController
  caches_page :index, :new
…
end
```

  - Wirkung in Ordner „public“:
    - Datei „books.html“, Ordner „books“ sowie Datei „books/new.html“ erscheinen.
    - Inhalt(books.html): Der von index.html.erb erzeugte Code, ohne den Layout-Teil!
    - Inhalt(books/new.html): Der von new.html.erb erzeugte Code, analog

- ***Page caching* (Forts.)**
  - Anmerkung
    - Caching ist standardmäßig nur in der Produktionsversion von Rails aktiv
    - Manuell im Controller einschaltbar:

      ```
      ActionController::Base.perform_caching = true
      ```

    - Z.B. in der Datei config/environments/development.rb einschalten:

      ```
      config.action_controller.perform_caching = true
      ```

  - Beobachtung im Logfile
    - Caching scheint zu funktionieren: Erneute Seitenabrufe bewirken keine Logzeilen mehr, außer man entfernt die statischen Seitenkopien manuell
  - Problem:
    - Paginierung verträgt sich nicht mit *page caching* – offenbar werden URL-Parameter wie „page=2“ nicht zur Unterscheidung herangezogen!

- ***Page caching* und Gültigkeitsende**
  - Solange der Entwickler die Kontrolle über alle Datenänderungen hat, genügt das Kommando „expire_page“ an den richtigen Stellen:

```
# In BooksController:
def create
@book = Book.new(params[:book])
  expire_page :action => "index"
  respond_to do |format|
    …
# In AuthorsController:
def create
@author = Author.new(params[:author])
  expire_page :controller => "books", :action => "new"
  respond_to do |format|
    …
```

  - Bem.: Analoge Einträge sind auch an anderen Stellen wie „update“ oder „destroy“ erforderlich

- ***Action caching***
  - Sollte ein vollständiges *page caching* an erforderlichen Filtern scheitern, so wählt man als analog funktionierende Ersatzlösung „*action caching*“.
  - Deklaration „***caches_action***“ in Controllern veranlasst, eine statische Kopie beim erstmaligen Aufrufen eines entsprechenden Views anzulegen.
  - Zwar wird der eigentliche Action-Code danach nicht mehr durchlaufen, wohl aber die *before*- und *after*-Filter. Delegieren der Auslieferung an den Web Server ist daher hier nicht möglich.
  - Typischer Anwendungskontext: Aktionen mit Zugriffsbeschränkungen
  - Beispiel:

```
class BookController < ApplicationController
  before_filter :authenticate, :except => :index
  caches_page :index
  caches_action :new
end
```

  - Gegenstück:

```
expire_action :action => "new"
```

- ***Action caching***
  - Cache-Speicher ist nun wählbar:
    - Hauptspeicher
    - Dateisystem
    - DRb-Server
    - „MemCached“
    - Eigene Lösung
  - Implikationen bei mehreren App.-Servern beachten!
- Beispiele für Konfiguration:

```
ActionController::Base.cache_store = :memory_store
ActionController::Base.cache_store = :file_store,
  "/path/to/cache/directory"
ActionController::Base.cache_store = :drb_store,
  "druby://localhost:9001"
```

- Dokumentation

  http://guides.rubyonrails.org/caching_with_rails.html, Kap. 1.6 „Cache stores“

- ***Fragment caching***
  - Auch *action caching* speichert ganze *Views*. Der Aufbau komplexer Seiten wie spiegel.de, t-online.de etc. könnte aber schon erheblich beschleunigt werden, wenn
    - nur ihre variablen Bestandteile pro Aufruf zu berechnen sind
    - ihre konstanten Bestandteile jedoch einem *caching* unterliegen
  - Genau dies leistet *fragment caching*. Es wird gerne mit *Partials* kombiniert!
  - Kommando „***cache do … end***“ umrahmt dabei solche Teile eines Views, die dem *Caching* unterliegen sollen.

  - Beispiel (Teil eines Views):

```
<p>Dynamisch ermittelte Uhrzeit: <%= Time.now.to_s %>
<% cache do %>
<p>Uhrzeit im Cache-Teil: <%= Time.now.to_s %>
<% end %>
<p>Wieder dynamisch ermittelte Uhrzeit: <%= Time.now.to_s %>
```

- ***Fragment caching (Forts.):***
  - Bei mehreren Cache-Fragmenten pro Seite können Sie diese benennen:

```
<% cache(:action => 'demo', :part => 'Uhrzeit') do %>
```

  - Auch Fragmente veralten und müssen dann gekennzeichnet werden
  - Analog zum *action caching* geschieht dies hier mit „***expire_fragment***“
  - Bei nur einem Fragment pro Controller und *action* genügt einfach

```
expire_fragment
```

  - Bei mehreren Fragmenten ergänzen Sie Angaben wie bei „cache“:

```
expire_fragment(:action => 'demo', :part => 'Uhrzeit')
```

  - Auch der Controller kann explizit angesprochen werden

```
expire_fragment(:controller => 'other',:action => 'demo2')
```

  - Auch für Fragment caching gibt es Speicheroptionen

```
ActionController::Base.fragment_cache_store =
  ActionController::Caching::Fragments::MemoryStore.new
```

Alternativen: `FileStore.new(path)`, `DRbStore.new(url)`, `MemCachedStore.new(host)`

- **DB-Ebene**
  - SQL *caching* (von Rails implizit ausgeführt während einer *action*)
  - Vermeidung von Anfragen
  - Optimierung von Anfragen
    - Z.B. nur die ersten *n* Werte statt gleich alle anfordern
    - Index-Ergänzung
  - Auslagern von „Langläufern“
  - Umstieg auf leistungsfähigeres RDBMS
  - SQL *Tuning*
    - Nutzung spezieller leistungssteigernder Maßnahmen durch Entwicklung von meist produktspezifischem SQL-Code
    - ActiveRecord wird dabei i.d.R. umgangen oder nur noch zur reinen Ausführung vorbereiteter SQL-Statements verwendet

Performance von Web-Anwendungen

- Ebene der Applikations-Server
  - Parallelisierung auf Thread-Ebene
    - Erfordert z.Z. JRuby und eine Java-Infrastruktur
    - Nützlich bereits mit S*ingle-core* CPUs
  - Parallelisierung auf Prozess-Ebene
    - „*A pack of Mongrels*“
    - Gut geeignet für *Multicore*-CPUs
  - Parallelisierung auf Server-Ebene
    - Anzahl Anfragen pro Sekunde linear mit Rechenleistung steigerbar
    - Engpass Datenbank-Server beachten!
    - Engpass Dateisystem-I/O ?

Performance von Web-Anwendungen

- Optimierung der Applikations-Server
  - Ruby 1.8.x ist relativ langsam. Abhilfen?
  - Option Ruby 1.9.x:
    - Byte-code engine „YARV“?
    - Faktor 2?
  - Option JRuby & Java VM:
    - Faktor 2?
    - Native thread-Support!
  - Option *Ruby Enterprise Edition* (in *Phusion Passenger*):
    - Speicher-Optimierung (bis zu 30% Ersparnis), Basis: Ruby 1.8.6

Performance von Web-Anwendungen

- Wahl der **Kombination** Web Server / Applikations-Server
  - WEBrick
    - Pure Ruby-Server, nur für kleine Vorhaben und *development*
  - Mongrel
    - Ruby + C *extensions*, schneller als WEBrick, einfach in der Anwendung
  - Apache 2.x oder LightTPD und fast_cgi
    - Effizient und mit dynamischer Lastverteilung, auch über Netzwerkgrenzen
    - Stabilitätsprobleme?
  - Apache 2.x + mod_proxy_balancer + mehrere Mongrels
    - Erwas mühsam zu konfigurieren, aber aktuell beliebt
  - Apache 2.x + mod_rails (*Phusion Passenger*):
    - Dynamisch wachsender & schrumpfender Pool von Rails-Instanzen, ähnlich wie bei FastCGI; wirksame Delegation des Auslieferns statischer Dateien an Apache
    - Speichereffizienter als Mongrel-Lösungen
    - Kopplung über Unix Domain Sockets vermeidet Port-Konfiguration
    - Besonders einfach zu konfigurieren; Neustart der App-Server implizit

Performance von Web-Anwendungen

- JRuby + Glassfish + Warbler: Die Java-Option
  - Wer bereits eine umfangreiche Java-Infrastruktur betreibt, findet hier Werkzeuge, um Rails-Projekte zu integrieren:
  - JRuby:
    - Pure Java-Implementierung der Sprache Ruby, incl. *native thread*-Support
    - Inzwischen kompatibel mit 1.8.x und (fast) mit 1.9.0
    - Angeblich deutlich schneller als Matz‘s Ruby, aber (noch?) nicht ganz so performant wie YARV
    - Ermöglicht volle Integation mit Java-Bibliotheken sowie Einbindung von Ruby-Code in Java-Projekte
  - Warbler: Ein Gem, das .WAR-Dateien aus Rails-Projekten erzeugt
  - GlassFish: Ein Java EE-Server
    - „*with out-of-the-box-clustering and high-availability support*“
  - Einzelheiten: wiki.jruby.org/wiki/JRuby_on_Rails_in_GlassFish
- *Deployment*
  - WAR-Datei erzeugen, in GlassFish einspielen - fertig

Performance von Web-Anwendungen

- **Fazit: Die Gesamtlösung muss ausgewogen sein, Optimierung weniger Komponenten genügt nicht!**