31.8.–3.9.2015
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Kinderspiel (aber kein Spielzeug)

Webanwendungen mit dem Play Framework

Dr. Stefan Schlott

BeOne Stuttgart GmbH

ABOUT.TXT

Stefan Schlott, BeOne Stuttgart GmbH

Java-Entwickler, Scala-Enthusiast, Linux-Jünger

Seit jeher begeistert für Security und Privacy


OUT OF THE BOX...

FULL-STACK FRAMEWORK

Integrierter Webserver

twirl Template Engine

DB-Zugriff mittels Ebean ORM oder Anorm („SQL DSL“)

Coffeescript- und Less-Support

Modularisierung, entkoppelte Komponenten

Buildsystem

…Serviervorschlag. Viele Module austauschbar.

PHILOSOPHIE

Asynchrone Verarbeitung

Zustandsloser Server

Typsicherheit

Interne Modularität, Erweiterbarkeit durch Modules

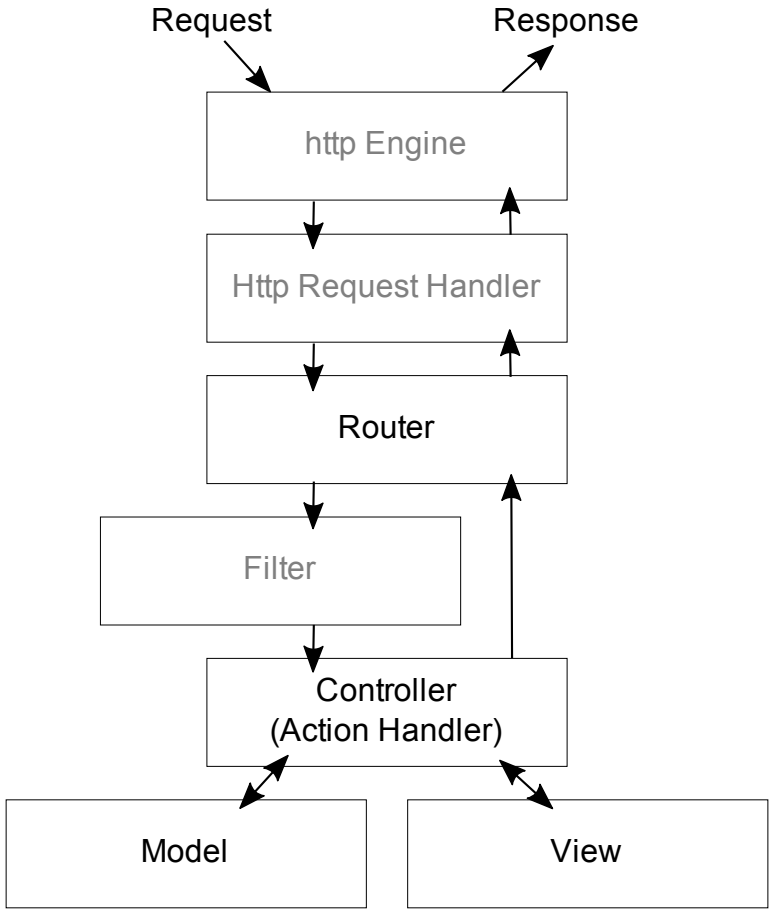
APIs für Java und Scala


AUFBAU UND BASICS

PROJEKT-STRUKTUR

app	Anwendungsquelltext
└ assets	Übersetzte Assets
└ stylesheets	z.B. LESS
└ javascripts	z.B. CoffeeScript
└ controllers	Application Controller
└ models	Business Layer
└ views	Templates
└ Global.java/scala	Globales Setup (optional, deprecated)
build.sbt	sbt Buildscript
conf	Config-Files u.ä. (im Classpath)
└ application.conf	Haupt-Config-Datei
└ messages	String-Tabelle
└ routes	Routen-Definition
public	Öffentliche, statische Assets
project	sbt-Konfiguration
└ build.properties	
└ plugins.sbt	sbt-Plugins (insbes. Play-Plugins)
test	Unit Tests

REQUEST-ABLAUF



HELLO, PLAY WORLD!

Erzeugen + Starten eines Stub-Projekts mittels Typesafe Activator:

```
activator new helloplayworld play-scala  # bzw. play-java
cd helloplayworld
./activator run
```

Router-Datei in `conf/routes` bildet URLs auf Controller ab

```
GET     /       controllers.Application.index
```

Controller-Action nutzt View:

```
def index = Action {
  Ok(views.html.index("Your new application is ready!"))
}
```

Router, Reverse Router, View-Template von Buildsystem compiliert

TEMPLATE ENGINE

```
@(message: String)              @* 1 *@

@main("Welcome to Play") {      @* 2 *@
  <p>@message                   @* 3 *@
}
```

`@`: „Magic Character“ für Funktionen

`@*` … `*@`: Kommentare

`1`: Erste Zeile definiert Parameter (werden von Controller übergeben). Keine Parameter: `@()`

`2`: Verwenden eines anderen Templates (hier: Seiten-Rumpf)

`3`: Rückgabewert von Funktionen (hier: Variablenwert) = Ausgabe in Seite. Automatisches html-Escaping, wenn nicht vom Typ Html!

ROUTING

http-Verb, URL-Pfad, Controller:

```
GET  /login       controllers.Application.login
POST /login       controllers.Application.handleLogin
```

Dynamische Anteile (Typ respektive RegEx müssen matchen!):

```
GET  /user/:name           controllers.User.showInfo(name: String)
GET  /product/:id          controllers.Product.showInfo(id: Long)
GET  /$lang<[a-z]{2}>/info controllers.Application.showInfo(lang: String)
```

URL-Parameter: Controller-Parameter ohne entsprechenden Platzhalter im URL-Pfad:

```
GET  /products    controllers.Product.showList(page: Option[Int])
```

REVERSE ROUTING

Route Compiler generiert routes-Objekt mit Funktionen, welche entsprechende URL-Pfade liefern

Nie wieder Broken Links!

Beispiel Link auf Login- und User-Seite:

```
GET   /login         controllers.Application.login
GET   /user/:name    controllers.User.showInfo(name: String)
```

```
<a href="@routes.Application.login()">Login</a>
<a href="@routes.User.showInfo("hugo")">Info über User "hugo"</a>
```

Entsprechende JS-Datei kann ebenfalls generiert werden

CONTROLLER

Zustandslos: Abbildung von Eingabe (Request) auf Ausgabe

„Unter der Haube“: Header + Bytestrom-Iteratee ⇒ Header + Bytestrom-Enumerator

Darauf aufsetzend: Abstraktionen für „Handling am Stück“, JSON/XML-Verarbeitung, etc. (Body Parsers)

Implizit asynchron

CONTROLLER ACTIONS

```
def index = Action { implicit request =>
  // Logik
  val userId = request.session.get("user")
  val userName = getFullName(userId)
  // Darstellung: Template rendern
  Ok(views.html.index(userName))
}
```

Rückgabewert: `Result` oder `Future[Result]`

Verschiedene Convenience-Results: `Ok`, `BadRequest`, etc.

Bei asynchronen Operationen: Futures mittels `map` auf (letztlich) `Future[Result]` abbilden

SESSION

Session client-seitig - wird in einem Cookie mitgegeben

Im Klartext lesbar

Schutz vor Manipulation durch Signatur (basiert auf `play.crypto.secret` in `conf/application.conf`)

Auslesen mit `request.session.get(…)`

Setzen bei Liefern des Results:

```
Ok(views.html.index(userName))
  .withSession("property" -> "value")
```


MULARE

CONTAINER UND MAPPER

Container-Objekt: In Scala Case-Class für Formulardaten

```
case class LoginData(login: String, password: String)
```

Mapper: Abbildung zwischen Container-Objekt und `Form` (enthält Zustand der Felder mit Werten, Fehlermeldungen, etc.)

```
val loginForm = Form(
  mapping(
    "loginform.login" -> nonEmptyText,
    "loginform.password" -> text
  )(LoginData.apply)(LoginData.unapply)
)
```

Einfache Validierung kann hier definiert werden

Namen im Mapper: Gleichzeitig CSS-IDs und IDs für String-Lookup

VIEW: HTML-HELPER

Helper-Funktionen generieren Formularfelder (mit Feldbeschriftung, Eingabefeld, Anzeige von Validierungsfehlern):

```
@import helper._
@(loginForm: Form[models.forms.LoginData])

@main("Bitte anmelden") {
  @helper.form(action = routes.Application.loginSubmission) {
    @inputText(loginForm("login"),
      '_label -> "Benutzername", 'placeholder -> "Benutzername")
    @inputPassword(loginForm("password"),
      '_label -> "Passwort", 'placeholder -> "Nicht verraten :-)")
    <input type="submit" value="Submit">
  }
}
```

`loginForm(…)` ergibt (im Mapper deklariertes) `Field`

FORM SUBMISSION

Daten können als Key-Value-Paare verarbeitet werden:

```
val data = request.body.asFormUrlEncoded
```

Komfortabler: Binding-Mechanismus verwenden:

```
val data: LoginData = loginForm.bindFromRequest.get
```

Wirft Exception im Falle von ValidierungsFehlern. Besser daher:

```
loginForm.bindFromRequest.fold(
  formWithErrors: Form[LoginData] => { … },
  login: LoginData => { … }
)
```

XSRF-SCHUTZ

Verhindern, dass bösartige Skripte Formulardaten abschicken

Controller Action beim Rendern des Formulars:

```
def login = CSRFAddToken {
  Action { implicit request => Ok(views.html.login(LoginData.loginForm)) }
}
```

Beim Rendern des Formulars: XSRF-Token einbinden

```
@helper.form(action = routes.Application.loginSubmission) {
  (…)
  @CSRF.formField
}
```

Prüfung bei der Formularbearbeitung:

```
def loginSubmission = CSRFCheck {
  Action { implicit request => (…) }
}
```

ADVANCED STUFF

STATELESS SERVER (DEMO)

haproxy und zwei Instanzen

ACTION COMPOSITION

Action Composition: Zusätzliche Behandlung für bestimmte Actions

Beispiele: XSRF-Schutz, Rechteprüfung, Logging, etc.

In Java: Mittels Annotation

In Scala: Ableitung von ActionFilter (statt `Action` verwenden)

Verkettung möglich

MODULE

Statt Initialisierung über Global

Implementierung von `play.api.inject.Module`

Angabe der Klasse in application.config in `play.modules.enabled`

Zugriff über Dependency Injection

DEPENDENCY INJECTION

Default: Guice, ist aber austauschbar

Alternative Cake Pattern (Compile-time DI) ist ebenfalls vorbereitet

Lifecycle:

- Instanzen werden bei jedem Bedarf neu erzeugt (sonst mit `@Singleton` annotieren)
- Instanzen werden lazy (also erst bei Bedarf) initialisiert
- Kein automatischer Cleanup. Über `ApplicationLifecycle` kann Code beim Herunterfahren der Anwendung ausgeführt werden.

FAZIT

Modernes Framework, „fühlt sich richtig an“

Schnelle Entwicklung (automatisches Übersetzen, sprechende Fehlerseiten)

Zustandslosigkeit: Ungewohnt, aber zwingt zum Nachdenken über „die richtigen Dinge“

BeOne

stuttgart

Dr. Stefan Schlott
http://www.beone-group.com/
stefan.schlott@beone-group.com
Twitter: @_skyr

BILDQUELLEN

- Money Box (CC) BY-NC-ND Haley Luvison
- under construction (CC) BY-NC-SA Pedro Moura Pinheiro
- look at alll the papers! (CC) BY-NC-SA Sara Grajeda
- 145/365 - Mad Science (CC) BY-NC-SA Dennis Wilkinson