Wie man in Java Webanwendungen bauen sollte

Webanwendungen mit dem Play!-Framework

Phillip Ghadir, Philipp Haußleiter

Das Play!-Framework ist ein recht junges, vollständiges MVC-Framework, das den Charme von Ruby on Rails hat. Aber anstatt unreflektiert jede Entwurfsentscheidung von Rails ins Play!-Framework zu übernehmen - wie das manch andere Rails-Klone gemacht haben -, haben die Entwickler darauf geachtet, es so zu konstruieren, dass es sich gut in das Ökosystem eines typischen Java-Entwicklers einfügt.

Auch wenn an verschiedenen Stellen „Magie“ herrscht und Einiges implizit geschieht, können wir in Play! sehr expliziten und lesbaren Code schreiben. Grundsätzlich erstaunt beim ersten Kennenlernen die durchgängige Verwendung von statischen Methoden in den Controllern. Im Gegensatz zu anderen „lernintensiveren“ Frameworks (wie z. B. Tapestry) hat das den Vorteil, dass man sich nicht um den Zustand von Controller-Logik sorgen muss, da er nicht in Instanz-Attributen abgelegt wird.

Play! wird von einer kleinen Community weiterentwickelt und wurde vor Kurzem von den Entwicklern hinter Scala als Web-Framework der Wahl ausgewählt (siehe Kasten „Was bringt Play 2.0“).

Der Aufbau einer Play!-Anwendung überrascht

Kontrollzentrum für die Entwicklung mit Play! ist das gleichnamige Skript, mit dem Anwendungen erzeugt oder für die Bearbeitung in einer Entwicklungsumgebung vorbereitet werden können. Zudem werden darüber der Server oder die Tests gestartet sowie andere alltägliche Dinge des Entwickleralltags angestoßen. Mit

```
play new <projektname>
```

legt man ein Verzeichnis **<projektname>** samt Standard-Struktur an. Unterhalb von **app/** werden die Models,Views und Controller (MVC) in den entsprechenden Unterverzeichnissen **models/**, **views/** und **controllers/** angelegt.

Das Programmiermodell bricht mit der üblichen Konvention, die Package-Struktur analog zum Domainnamen des Erstellers zu vergeben. Im Play!-Framework lauten die Packages typischerweise **controllers**, **models**.

Bibliotheken werden im **lib/**-Verzeichnis und statische Inhalte im **public/**-Verzeichnis hinterlegt. Konfigurationen finden ihren Platz in **conf/** und Test-Klassen in **test/**. Darüber hinaus gibt es weitere Verzeichnisse zum Sammeln von Logs, Testergebnissen und Temp-Dateien.

Ein Blick in den vorgenerierten Controller namens **Application** offenbart eine grundlegende Entwurfsphilosophie von Play!: Alle Controller-Methoden sind statisch! Man verwendet im Controller keine Instanz-Variablen. Stattdessen wird der erforderliche Kontext in Thread-lokalen, globalen Variablen (siehe [thread-local]) gehalten. Dies ermöglicht dem Anwendungsentwickler ein nettes single-threaded Programmiermodell.

Was bringt Play! 2.0?

Version 2.0 soll die aktuell verwendeten Sprachen des Frameworks (Java, Python, Groovy) vereinheitlichen. Die Play!-Entwickler arbeiten an Version 2.0 eng mit Typesafe zusammen, der Firma hinter der Sprache Scala.

Scala muss in der neuen Play!-Version nicht mehr, wie bisher, als Modul geladen werden, sondern ist demnächst direkter Framework-Bestandteil.

Groovy wird als Template-Sprache durch Scala abgelöst und das Standard-Build-Tool sbt die Rolle als Build-Tool der Wahl übernehmen. Akka ([akka]), eine Plattform für skalierbare und fehlertolerante Anwendungen, wird ebenfalls in Play! 2.0 enthalten sein.

Ein weiteres großes Feature wird die Einführung einer Play!-Console sein, die – ähnlich der Rails-Console bei Ruby on Rails – erlauben soll, Befehle im Kontext einer Play!-Anwendung auszuführen.

Zum aktuellen Stand der Entwicklung siehe [play-2.0]). Stand 1/2012 sind die Play! 2.0 APIs noch nicht vollständig stabil. Für den produktiven Einsatz ist daher weiterhin Version 1.2.4 empfohlen.

Modelle zum Spielen

In Play! setzt man typischerweise auf JPA, um Entitäten zu realisieren. Dazu können die JPA-Entitäten völlig ohne Abhängigkeiten zu Bestandteilen des Play!-Frameworks auskommen. Wer möchte, kann aber über das Erben von einer spezifischen Superklasse (**play.db.jpa.Model**) zusätzliche Funktionalität wie dynamische Finder geschenkt bekommen. Die Lebenszyklus-Verwaltung der Entitätsklassen funktioniert dann während der Entwicklung problemloser und man spart ein wenig Schreibarbeit, da das Hantieren mit dem Entity-Manager entfällt.

Play! verwendet ungewöhnliche Konventionen, um Entitäten anzureichern. Man deklariert die Attribute einer Entität **public** und verzichtet auf das Schreiben von Zugriffsmethoden. Trotzdem erstellt das Framework die Zugriffsmethoden und sorgt auch für deren Aufruf. Das ermöglicht auch im Nachgang stets den sicheren Zugriff über speziell dafür erzeugte Methoden. Obwohl wir zum Beispiel in Controller-Methoden schreiben:

```
meineEntitaet.name = "Super Mario";
```

wird daraus beim Laden der Klasse ein entsprechender Aufruf der (automatisch generierten) Setter-Methode gemacht. Das funktioniert dank sogenannter Instrumentierung des Java-Codes (siehe [java-instrumentation]).Trotzdem ist es jederzeit möglich, manuell Zugriffsmethoden zu implementieren, sollte dies erforderlich sein.

Modellklassen um Validierungen erweitern

Validierungen können in Entitäten über einfache Annotationen hinzugefügt werden. Überprüfungen, ob eine Eingabe zum Beispiel eine Zahl innerhalb eines Wertebereichs, eine URL oder eine E-Mail-Adresse ist, können einfach am Attribut innerhalb der Entität annotiert werden (s. Listing 1). Darüber hinaus besteht die Option, Validierungen in Zugriffsmethoden selbst zu implementieren.

In jeder Controller-Action kann einfach auf Validierungsfehler durch Abfrage des **validation**-Objektes reagiert werden (s. Listing 3, Zeile 9 – 11).

```
1. @Entity
2. public class User extends Model {
3.
4.   @Required
5.   public String name;
6.   public String number;
7.   @URL
8.   public String tenant;
9.   public boolean isPowerUser;

10.  public User(String name, String number,
                 String tenant, boolean powerUser) {
11.    isPowerUser = powerUser;
12.    this.name = name;
13.    this.number = number;
14.    this.tenant = tenant;
15.  }
16. }
```

Listing 1: Eine Beispiel-Entität mit zwei Validierungen auf den Attributen

Views

Views werden mit Templates implementiert, in denen zwischen die literal zu verwendenden Textbausteine Groovy-Ausdrücke geschrieben werden. Die Syntax ist für jeden JSP-Veteranen erst einmal irritierend: `${ ... }` für Ausdrücke, `%{ ... }%` für Skripte, `#{ ... /}` für Template-Aufrufe, `&{ ... }` für die Internationalisierung von Meldungen sowie `@{ ... }` und `@@{ ... }` für das Erzeugen von relativen bzw. absoluten URLs mithilfe des Play!-Routers. Listing 2 zeigt beispielsweise eine View, die alle Benutzer darstellt.

```
1. #{extends 'main.html' /}
2. #{set title:'Benutzer' /}
3.
4. <ul>
5.   #{list items:users, as:'user'}
6.     <li>${user.name}</li>
7.   #{/list}
8. </ul>
9.
10. <a href="@{ Application.displayUsers() }">
      Detailansicht darstellen </a>
```

Listing 2: View „Alle Benutzer“

Controller – statisch und zustandslos

Controller in einer Play!-Anwendung sehen sehr aufgeräumt aus: Sie erben von der Basisklasse **`play.mvc.Controller`** und definieren eine beliebige Anzahl statischer Methoden – die sogenannten Controller-Actions (kurz: Actions).

Actions können beliebig lange Parameter-Listen definieren, die automatisch gebunden werden. Für die primitiven Typen und deren objektorientierte Pendants gibt es eine automatische Bindung aus dem Request auf die Parameter. Gleiches gilt bei der Verwendung von Entitäten als Parameter-Typen. Dazu muss der Request-Parameter mit dem Namen des Methoden-Parameters übereinstimmen. Bei Bedarf kann die Bindung mithilfe von Annotationen angepasst oder unterdrückt werden. Dies ist insbesondere für Daten interessant, die landes- oder sprachspezifisch formatiert sind oder aus Sicherheitsgründen gar nicht gebunden werden sollen.

```
1. public class UserController extends Controller {
2.
3.   public static void index() {
4.     List<User> users = User.find("order by name asc").fetch();
5.     render(users);
6.   }
7.   public static void create(@Required String name,
                               @Email String email) {
8.
9.     if ( validation.hasErrors() ) {
10.      flash.error("Name and email are required!");
11.      index();
12.    }
13.    User u = new User(name, email).save();
14.    render(u);
15.  }
16. }
```

Listing 3: UserController

Listing 3 zeigt exemplarisch den **`UserController`**, mit zwei Methoden: **`index()`** zur Anzeige aller Benutzer sowie **`create(String, String)`** zum Anlegen eines neuen Benutzers. Zeile 7 zeigt, wie im Controller Validierungen definiert werden können. Die Annotationen **`@Required`** und **`@Email`** reichern die Parameter der Methode an. Analog kann man die Attribute der Entität User – wie in Listing 1 dargestellt – annotieren. Das wäre sogar die bevorzugte Variante.

Controller erlauben weiterhin das Einfügen von Interceptoren, um vor oder nach jedem Aufruf eine Action-spezifische Funktionalität aufzurufen.

Konfiguration

Play!-Anwendungen werden über die im **`conf/`**-Verzeichnis hinterlegten Konfigurationsdateien konfiguriert. Die zentrale Konfiguration findet dabei über die Datei **`conf/application.conf`** statt. Diese ist in [play-conf] gut beschrieben. Darüber hinaus werden typischerweise Routen, lokalisierte Meldungstexte oder Logging in separaten Konfigurationsdateien konfiguriert.

Routing

Play! erwartet – im Gegensatz zu Ansätzen wie JAX-RS – die Zuordnung von URL-Muster und Http-Verben auf Controller-Methoden in einer separaten Konfigurationsdatei: **`conf/routes`**. Routen werden dort per

```
GET    /                       Application.index
*      /{controller}/{action}  {controller}.{action}
```

definiert. Dabei enthält die erste Spalte das Http-Verb – wobei * für ein beliebiges steht. Die zweite Spalte gibt das URL-Template, die dritte die Controller-Action an. Die geschweiften Klammern beinhalten Variablen.

Das Routing funktioniert in Play! in beide Richtung: Anhand eines URI-Parts wird die entsprechende Controller-Methode gefunden, genauso wie eine URI für eine entsprechende Controller-Methode erzeugt werden kann (s. z. B. Listing 2, Zeile 10).

Modularisieren großer Anwendungen

Eine Play!-Anwendung kann sogenannte Module verwenden. Deren Struktur entspricht der einer Anwendung ohne Applikationskonfiguration. Dafür können Module eine **`conf/routes`**-

Datei enthalten. Eine Play!-Anwendung muss explizit in der eigenen `conf/routes`-Datei die Routen der Module importieren. Dabei haben die Definitionen in der Anwendung Vorrang vor denen in den geladenen Modulen. Module, die sich im Verzeichnis `/modules` befinden, werden beim Start der Anwendung automatisch geladen.

Es gibt auch die Möglichkeit, die Module von einem Module-Repository einzubinden. Die Abhängigkeiten werden über eine Datei namens `/conf/dependencies.yml` definiert und können dann automatisch synchronisiert werden.

Authentifikation & Autorisierung in Play!

Play! unterstützt durch seine Architektur den Umgang mit kritischen Daten und die Entwicklung von sicheren Webanwendungen. Informationen hierzu sind dem Security-Guide ([play-sec-guide]) zu entnehmen. Um z. B. Ressourcen zu schützen, muss das `Secure`-Modul, welches Teil der Play!-Basis-Installation ist, eingebunden werden.

Durch Importieren der Default-Routen aus dem `Secure`-Modul und durch das Annotieren der den Routen zugeordneten Controller mit `@With(Secure.class)` werden Ressourcen vor unautorisiertem Zugriff geschützt.

Als Nächstes muss der Authentifikationsprozess angepasst werden, indem wir eine eigene Ableitung der Klasse `Secure.Security` mit der bezeichneten Methode implementieren:

```
public class Security extends Secure.Security {
  static boolean authenticate(String username, String password) {
    return User.connect(username, password) != null;
  }
}
```

Für weitergehende Rollen/Berechtigungsmodelle gibt es das Secure-Permissions-Play!-Modul ([play-secperm-modul]). Dieses wertet Regeln zur Berechtigungsprüfung mittels Drools (siehe [drools]) aus, die in der Konfigurationsdatei `conf/permissions.drl` hinterlegt werden können.

Im Controller kann die Berechtigung dann entweder über die Annotation `@CheckPermission` oder programmatisch mittels `Secure.checkPermission()` abgefragt werden (s. Listing 4).

```
@With(Secure.class)
public class Application extends Controller {
  /* You can do this using annotations */
  public static void saveItem(@CheckPermission("update") Item item) {
    ...
  }
  /* Or programatically */
  public static void deleteItem(Long itemID) {
    Item item = Item.findByID(itemID);
    if(!Secure.checkPermission(item, "delete"))
      forbidden();
    ...
  }
}
```

Listing 4: Application-Controller, mit Secure-Annotation und zwei Berechtigungsprüfungen

Internes Caching

Play! bietet eine Schnittstelle zur Framework-eigenen Caching-Implementierung, die aber auch problemlos durch eine existierende Memcached-Installation [mem-cached] ausgetauscht werden kann.

Play!-Objekte werden mittels `Cache.set(key, value, expiration)` serialisiert und im Cache abgelegt. Per `Cache.get(key)` wird das zugehörige Objekt aus dem Cache geladen und deserialisiert. Der häufigste Anwendungsfall ist das Zwischenspeichern von instanziierten Modellen im Cache.

Möchte man einen Eintrag aus dem Cache entfernen, so bietet das Framework hierzu zwei unterschiedliche Möglichkeiten:

Durch den Aufruf der Methode `Cache.delete(String key)` wird der Eintrag als invalide gekennzeichnet, allerdings wird hier nicht abgewartet, bis das Objekt tatsächlich aus dem Cache entfernt worden ist.

Durch den langsameren, blockierenden Aufruf von `Cache.safeDelete(key)` wird das Objekt sofort aus dem Cache gelöscht.

Anders als bei vielen Java-Frameworks haben die Entwickler von Play! Wert darauf gelegt, dass eine Benutzersession nicht zum Cachen von Daten verwendet wird. Sessions werden in Play! über die Ablage der eigentlichen Daten (nicht Schlüsseln) in Cookies realisiert. Und bekanntermaßen sind Cookies auf 4 KB begrenzt. Wenn es dennoch nötig ist, kann der Cache verwendet werden:

```
Cache.set("items_"+session.getId(), items);
...
List<Item> items = Cache.get("items_"+session.getId(), List.class)
```

Zu beachten ist, dass wir selbst dafür verantwortlich sind, einen Session-übergreifend eindeutigen Schlüssel zu erzeugen. Wenn man damit anfängt, sollte unbedingt die Information, wie ein Schlüssel Session-übergreifend eindeutig zugeordnet werden kann, an genau einer Stelle gekapselt sein – und nicht wie hier vereinfacht dargestellt redundant!

Play! in mehreren Betriebsmodi

Eine Play!-Anwendung lässt sich entweder in der Konsole per `play run` im Vordergrund oder als Hintergrundprozess mit `play start` starten, sowie mit `strg+c` bzw. `play stop` beenden. Mit `play status` lässt sich der Status der Anwendung ausgeben (geladene Module, verwendete Threads, aufgerufene Ressourcen usw.). Dies ist hilfreich, um HotSpots der Anwendung zu identifizieren.

Dabei gibt es im Wesentlichen zwei Spielarten: Im *Entwicklungsmodus* fährt der Server hoch, startet die Webanwendung aber erst beim ersten Zugriff. Bei jedem Zugriff werden die lokalen Änderungen am Quelltext direkt übersetzt und angezogen. Der Entwicklungszyklus, „Quelltext anpassen“ -> „Ergebnis im Browser sehen“ funktioniert direkt und schnell. Kompliziertes Kompilieren, Zusammenbauen, Deployen und vor allem langes Warten entfallen. – Etwas, das wir so vor Jahren bereits an Rails lieben gelernt haben.

Im *Produktionsmodus* dagegen fährt der Server hoch, startet unmittelbar die Anwendung und lädt dann nichts mehr dynamisch nach. Für den produktiven Betrieb einer Play!-Anwendung sollte die Konfiguration angepasst werden. Beispielsweise sollten in der Datei `conf/log4j.properties` das Rotieren der Log-Dateien eingeschaltet werden und Datenquellen für das Produktivsystem definiert werden.

Eine Play!-Anwendung lässt sich sowohl mittels eingebautem Webserver, JBoss Netty, betreiben als auch als Webanwendung – als WAR – innerhalb eines Java-Servlet-Containers. Das Erstellen eines WARs erfolgt über `play war -o .war`. Damit werden alle notwendigen Bibliotheken, die zum Betrieb der Play!-Anwendung notwendig sind, ins WAR gepackt.

Fazit

Play! ist ein sehr interessantes Framework für die Entwicklung von Webanwendungen. Wir konnten viel Wissenswertes in der Kürze gar nicht aufzählen, das Play! zu einem unserer liebsten Java-Web-Frameworks macht: beispielsweise die Verwaltung von Datenbank-Migrationen (sogenannten Evolutions) sowie der integrierte Scheduler, der es erlaubt, Hintergrund-Tasks zu realisieren.

Die Tatsache, dass Play!-Anwendungen auf eine Reihe vorgefertigter Module zurückgreifen können, ist ebenso interessant wie das Unterstützen praktisch jeder JVM-Sprache für die Anwendungsentwicklung.

Die Verwendung statischer Methoden im Controller ist nur zu Beginn gewöhnungsbedürftig. Wirklich gelungen ist die Kapselung von Entitäten, sodass Client-Code stets über Zugriffsmethoden auf Attribute zugreift, obwohl wir den direkten Attribut-Zugriff hinschreiben.

Mit Play! 2 stehen einige grundlegende Neuerungen bevor. Die engere Verknüpfung mit Scala, eine neue Template-Engine und eine Persistenzschicht, die für NoSQL-Datenbanken besser geeignet sein wird, lassen für das bevorstehende Major Upgrade hoffen. Ob die aktuelle oder die zukünftige Version: Play! ist ein gelungenes Framework, das den Flair von Ruby on Rails mit der Typsicherheit der Java-Welt verbindet.

Links

[akka] Akka – event-driven, scalable and fault-tolerant architectures on the JVM, `http://akka.io/`
[drools] Jboss Drools, `http://www.jboss.org/drools`
[java-instrumentation] `http://docs.oracle.com/javase/6/docs/technotes/guides/instrumentation/index.html`
[mem-cached] memcached – a distributed memory object caching system, `http://memcached.org`
[play-2.0] Play!, Version 2.0, `http://www.playframework.org/2.0`
[play-cloud] Play! Cloud-based hosting, `http://www.playframework.org/documentation/1.2.4/deployment#cloud`
[play-conf] Play! Konfiguration, `http://www.playframework.org/documentation/1.2.4/configuration`
[playframework] `http://www.playframework.org`
[playrepo] `http://www.playframework.org/modules`
[play-scala-modul] Play! Scala Modul, `http://www.playframework.org/modules/scala`
[play-sec-guide] `http://www.playframework.org/documentation/1.2.4/security`
[play-secperm-modul] `http://www.playframework.org/modules/securepermissions`
[playTutorial] `http://www.playframework.org/documentation/1.2.4/home#guide`
[thread-local] `http://en.wikipedia.org/wiki/Thread-local`

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com

Philipp Haußleiter arbeitet als Consultant bei innoQ Deutschland GmbH. Seine Schwerpunkte liegen neben Themen wie Java EE und OSGi auf der Entwicklung von HTML5-Anwendung mit leichtgewichtigen Web-Frameworks wie Ruby on Rails und Play!.
E-Mail: philipp.haussleiter@innoq.com