Kostenloses eBook

LERNEN

playframework

Free unaffiliated eBook created from **Stack Overflow contributors.**

#playframework

Inhaltsverzeichnis

Über 1

Kapitel 1: Erste Schritte mit playframework 2

Bemerkungen 2

Examples 2

Spielen Sie 1 Installation 2

Voraussetzungen 2

Installation aus dem Binärpaket 2

Allgemeine Anweisungen 2

Mac OS X 2

Linux 3

Windows 3

Installation durch "sbt" 3

Erste Schritte mit Play 2.4.x / 2.5.x - Windows, Java 4

Installationen 4

Spielen Sie den Installationsfix 2.5 5

Erstellen einer neuen Anwendung mit CLI 5

Aktivator an einem anderen Port ausführen 6

Kapitel 2: Abhängigkeitsinjektion - Java 8

Examples 8

Abhängigkeitsinjektion mit Guice - Spiel 2.4, 2.5 8

Einspritzung von Play-APIs 8

Kundenspezifische Bindung 8

Injektion mit der @ImplementedBy-Anmerkung 9

Injektionsbindung mit einem Standard-Play-Modul 9

Flexible Injection-Bindung mit einem Standard-Play-Modul 10

Injektionsbindung mit einem benutzerdefinierten Modul 11

Kapitel 3: Abhängigkeitsinjektion - Scala 12

Syntax 12

Examples 12

Grundlegende Verwendung 12
Play-Klassen einspritzen 12
Definieren von benutzerdefinierten Bindungen in einem Modul 13
Kapitel 4: Bau und Verpackung 15
Syntax 15
Examples 15
Fügen Sie der Distribution ein Verzeichnis hinzu 15
Kapitel 5: Einrichten Ihrer bevorzugten IDE 16
Examples 16
IntelliJ IDEA 16
Voraussetzungen 16
Projekt öffnen 16
Ausführen der Anwendungen von Intellij 16
Option zum automatischen Importieren 17
Eclipse als Play IDE - Java, Abspielen 2.4, 2.5 17
Einführung 17
Festlegen der Eclipse-IDE für jedes Projekt 17
Anhängen von Play source an Eclipse 18
Eclipse-IDE global einstellen 18
Debuggen von Eclipse 19
Eclipse IDE 19
Voraussetzungen 19
Scala in Eclipse installieren 20
Sbteclipse einrichten 20
Projekt importieren 20
Kapitel 6: Glatt 21
Examples 21
Slick Code für den Einstieg 21
Ausgabe-DDL 22
Kapitel 7: Java - Hallo Welt 23
Bemerkungen 23

Examples 23
Erstellen Sie Ihr erstes Projekt 23
Holen Sie sich den Aktivator 23
Der erste Lauf 24
Die "Hallo Welt" in der Hallo Welt 25
Kapitel 8: Java - Mit JSON arbeiten 27
Bemerkungen 27
Examples 27
Manuelles Erstellen von JSON 27
Json wird aus String / Datei geladen 27
Laden einer Datei aus Ihrem öffentlichen Ordner 27
Laden von einer Zeichenfolge 27
Übertragen eines JSON-Dokuments 27
Liefert den Namen eines Benutzers (unsicher) 28
Holen Sie sich den Benutzernamen (sicherer Weg) 28
Holen Sie sich das Land, in dem der erste Benutzer arbeitet 28
Holen Sie sich alle Länder 28
Finde jeden Benutzer, der das Attribut "aktiv" enthält 28
Konvertierung zwischen JSON- und Java-Objekten (Basis) 29
Java-Objekt aus JSON erstellen 29
Erstellen Sie ein JSON-Objekt aus einem Java-Objekt 29
Erstellen einer JSON-Zeichenfolge aus einem JSON-Objekt 29
JSON hübsches Drucken 29
Kapitel 9: Mit JSON - Scala arbeiten 31
Bemerkungen 31
Examples 31
JSON manuell erstellen 31
Java: JSON-Anforderungen annehmen 32
Java: JSON-Anforderungen mit BodyParser annehmen 32
Scala: Manuelles Lesen eines JSON 32
Nützliche Methoden 33
Automatische Zuordnung zu / von Fallklassen 33

Umwandlung in Json 34
Konvertierung von Json 34
Kapitel 10: Unit Testing 35
Examples 35
Unit Testing - Java, Abspielen 2.4.2.5 35
Helfer und FakeApplication 35
Controller testen 35
Controller testet Beispiel 36
Spott mit PowerMock 36
Verspottung einer Controller-Aktion 37
Verspottung einer Aktion mit JSON-Body 37
Verspottung einer Aktion mit dem Header der Basisauthentifizierung 38
Verspottung einer Aktion mit Sitzung 38
Kapitel 11: Webservice-Nutzung mit play WSClient 39
Bemerkungen 39
Examples 39
Grundnutzung (Scala) 39
Credits 40

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: playframework

It is an unofficial and free playframework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official playframework.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit playframework

Bemerkungen

In diesem Abschnitt erhalten Sie einen Überblick darüber, was Playframework ist und warum ein Entwickler es verwenden möchte.

Es sollte auch alle großen Themen innerhalb von playframework erwähnen und auf die verwandten Themen verweisen. Da die Dokumentation für playframework neu ist, müssen Sie möglicherweise erste Versionen dieser verwandten Themen erstellen.

Examples

Spielen Sie 1 Installation

Voraussetzungen

Um das Play-Framework auszuführen, benötigen Sie Java 6 oder höher. Wenn Sie "Play from source" erstellen möchten, benötigen Sie den Git-Quellcodeverwaltungsclient , um den Quellcode abzurufen, und Ant , um ihn zu erstellen.

Stellen Sie sicher, dass sich Java im aktuellen Pfad befindet (geben Sie zur Überprüfung `java --version`).

Bei der Wiedergabe wird das Standard-Java oder das unter dem **$ JAVA_HOME-** Pfad verfügbare Java verwendet, sofern definiert.

Das Befehlszeilenprogramm **play** verwendet Python. Es sollte also auf jedem UNIX-System sofort funktionieren (erfordert jedoch mindestens Python 2.5).

Installation aus dem Binärpaket

Allgemeine Anweisungen

Die Installationsanweisungen lauten im Allgemeinen wie folgt.

1. Java installieren
2. Laden Sie das neueste Play-Binärpaket herunter und entpacken Sie das Archiv.
3. Fügen Sie Ihrem Systempfad den Befehl "play" hinzu, und stellen Sie sicher, dass er ausführbar ist.

Mac OS X

Java ist integriert oder wird automatisch installiert, sodass Sie den ersten Schritt überspringen können.

1. Laden Sie das neueste Play-Binärpaket herunter und extrahieren Sie es in `/Applications` .
2. Bearbeiten Sie `/etc/paths` `/Applications/play-1.2.5` und fügen Sie die Zeile `/Applications/play-1.2.5` (zum Beispiel).

Eine Alternative zu OS X ist:

1. Installieren Sie HomeBrew
2. Laufen Sie `brew install play`

Linux

Um Java zu installieren, stellen Sie sicher, dass Sie entweder Sun-JDK oder OpenJDK verwenden (und nicht gcj, das ist der Standard-Java-Befehl in vielen Linux-Distributionen).

Windows

Laden Sie zur Installation von Java einfach das neueste JDK-Paket herunter und installieren Sie es. Sie müssen Python nicht separat installieren, da eine Python-Laufzeitumgebung im Framework enthalten ist.

Installation durch "sbt"

Wenn Sie bereits `sbt` installiert haben, `sbt` mir leichter, ein minimales Play-Projekt ohne `activator` zu erstellen. Hier ist wie.

```
# create a new folder
mkdir myNewProject
# launch sbt
sbt
```

Wenn die vorherigen Schritte abgeschlossen sind, bearbeiten Sie `build.sbt` und fügen Sie die folgenden Zeilen hinzu

```
name := """myProjectName"""

version := "1.0-SNAPSHOT"

offline := true

lazy val root = (project in file(".")).enablePlugins(PlayScala)
scalaVersion := "2.11.6"
# add required dependencies here .. below a list of dependencies I use
libraryDependencies ++= Seq(
  jdbc,
  cache,
  ws,
  filters,
  specs2 % Test,
```

```
  "com.github.nscala-time" %% "nscala-time" % "2.0.0",
  "javax.ws.rs" % "jsr311-api" % "1.0",
  "commons-io" % "commons-io" % "2.3",
  "org.asynchttpclient" % "async-http-client" % "2.0.4",
  cache
)


resolvers += "scalaz-bintray" at "http://dl.bintray.com/scalaz/releases"

resolvers ++= Seq("snapshots", "releases").map(Resolver.sonatypeRepo)

resolvers += "Typesafe Releases" at "http://repo.typesafe.com/typesafe/maven-releases/"
```

Schließlich erstellen Sie einen Ordner `project` und innerhalb einer Datei erstellen `build.properties` mit dem Hinweis auf die Version des Spiels möchten Sie verwenden

```
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.4.3")
```

Das ist es! Ihr Projekt ist fertig. Sie können es mit `sbt` . Innerhalb von `sbt` Sie Zugriff auf dieselben Befehle wie mit dem `activator` .

Erste Schritte mit Play 2.4.x / 2.5.x - Windows, Java

Installationen

Herunterladen und installieren:

1. Java 8 - Laden Sie die entsprechende Installation von der Oracle-Site herunter.
2. Activator - Laden Sie zip von www.playframework.com/download herunter und extrahieren Sie Dateien in den Zielordner " Play", z. B. für:

   ```
   c:\Play-2.4.2\activator-dist-1.3.5
   ```

3. sbt - download von www.scala-sbt.org .

Umgebungsvariablen definieren:

1. **JAVA_HOME** zum Beispiel:

   ```
   c:\Program Files\Java\jdk1.8.0_45
   ```

2. **PLAY_HOME** zum Beispiel:

   ```
   c:\Play-2.4.2\activator-dist-1.3.5;
   ```

3. **SBT_HOME** zum Beispiel:

```
c:\Program Files (x86)\sbt\bin;
```

Fügen Sie den Pfadvariablen einen Pfad zu allen drei installierten Programmen hinzu:

```
%JAVA_HOME%\bin;%PLAY_HOME%;%SBT_HOME%;
```

Spielen Sie den Installationsfix 2.5

Die Installation von Play 2.5.3 (die letzte stabile Version 2.5) bringt ein kleines Problem mit sich. Etwas reparieren:

1. Bearbeiten Sie die Datei *activator-dist-1.3.10 \ bin \ activator.bat* und fügen Sie am Ende der Zeile 55 das Zeichen "%" ein. Die richtige Zeile sollte folgendermaßen aussehen: *set SBT_HOME =% BIN_DIRECTORY%*
2. Erstellen Sie das Unterverzeichnis *conf* unter dem Activator-Stammverzeichnis *Activator-dist-1.3.10* .
3. Erstellen Sie im *conf-* Verzeichnis eine leere Datei namens *sbtconfig.txt* .

Erstellen einer neuen Anwendung mit CLI

Starten Sie das *Cmd* aus dem Verzeichnis, in dem eine neue Anwendung erstellt werden soll. Die kürzeste Methode zum Erstellen einer neuen Anwendung über die CLI ist die Angabe eines Anwendungsnamens und einer Vorlage als CLI-Argumente:

```
activator new my-play-app play-java
```

Es ist möglich nur zu laufen:

```
activator new
```

In diesem Fall werden Sie aufgefordert, die gewünschte Vorlage und einen Anwendungsnamen auszuwählen.

Für Play 2.4 fügen Sie die *Datei project / plugins.sbt* manuell *hinzu* :

```
// Use the Play sbt plugin for Play projects
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.4.x")
```

Stellen Sie sicher, dass Sie 2.4.x hier durch die genaue Version ersetzen, die Sie verwenden möchten. Spiel 2.5 erzeugt diese Linie automatisch.

Stellen Sie sicher, dass die richtige **sbt-** Version in project / build.properties angegeben ist. Sie sollte mit der auf Ihrem Computer installierten **sbt-** Version **übereinstimmen** . Zum Beispiel für Play2.4.x sollte es sein:

```
sbt.version=0.13.8
```

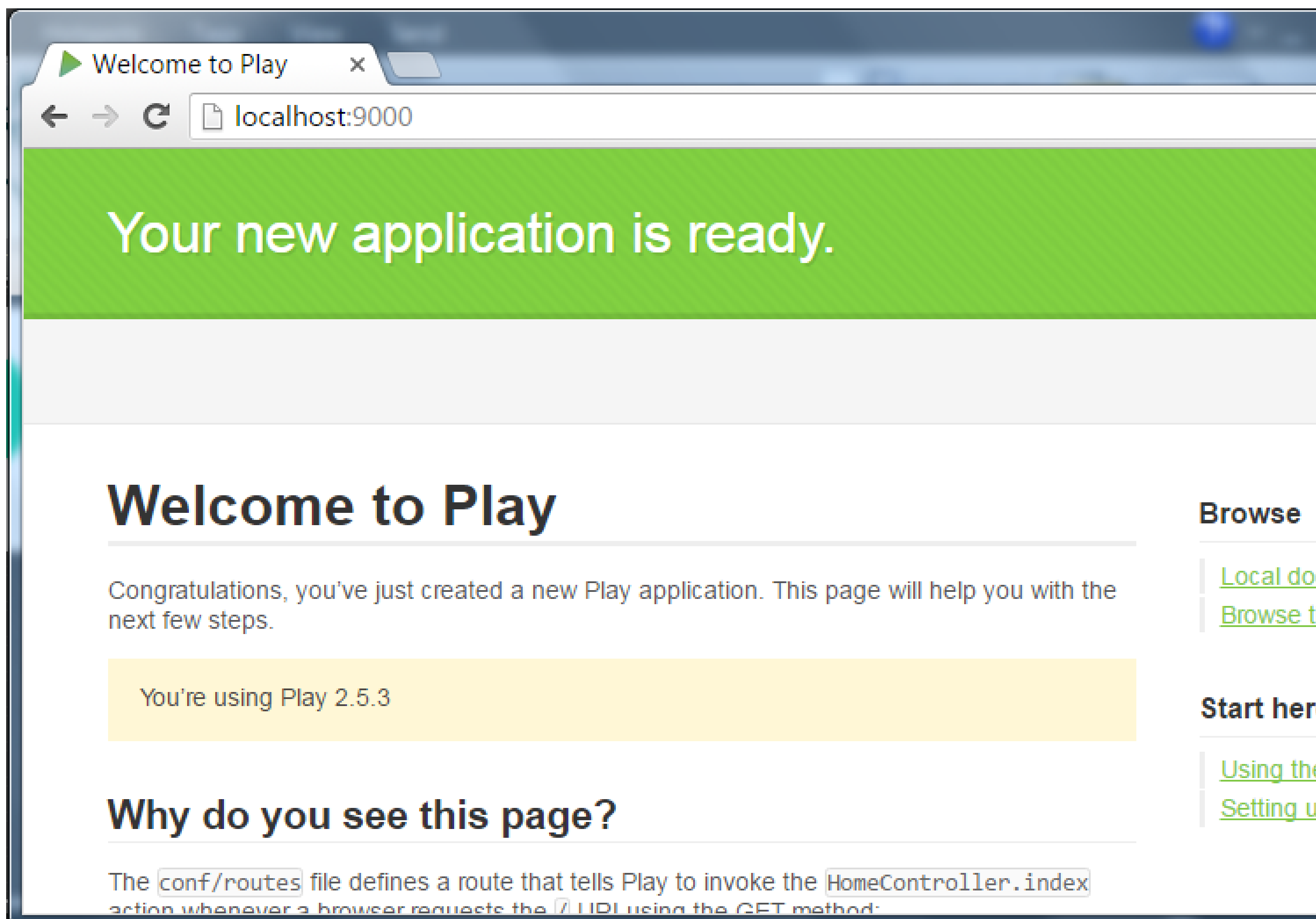
Das ist es, eine neue Anwendung kann jetzt gestartet werden:

```
cd my-play-app
activator run
```

Nach einer Weile startet der Server und die folgende Eingabeaufforderung sollte in der Konsole erscheinen:

```
[info] p.c.s.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000
(Server started, use Ctrl+D to stop and go back to the console...)
```

Der Server überwacht standardmäßig Port 9000. Sie können ihn über die URL *http: // localhost: 9000* von einem Browser aus anfordern. Sie werden so etwas bekommen:



Aktivator an einem anderen Port ausführen

Standardmäßig führt der Aktivator eine Anwendung auf Port 9000 für http oder 443 für https aus. So führen Sie eine Anwendung auf einem anderen Port (http) aus:

```
activator "run 9005"
```

Erste Schritte mit playframework online lesen:
https://riptutorial.com/de/playframework/topic/1052/erste-schritte-mit-playframework

Kapitel 2: Abhängigkeitsinjektion - Java

Examples

Abhängigkeitsinjektion mit Guice - Spiel 2.4, 2.5

Guice ist das standardmäßige Abhängigkeitsinjektionssystem (weiteres **DI**) von Play. Andere Frameworks können ebenfalls verwendet werden, aber die Verwendung von Guice erleichtert die Entwicklungsanstrengungen, da Play die Dinge unter dem Schleier erledigt.

Einspritzung von Play-APIs

Ab Play 2.5 sollten mehrere APIs, die in den früheren Versionen statisch waren, mit **DI erstellt werden** . Dies sind beispielsweise *Konfiguration* , *JPAApi* , *CacheApi* usw.

Die Injektionsmethode von Play API-s unterscheidet sich für eine Klasse, die automatisch von Play und für eine benutzerdefinierte Klasse eingefügt wird. Die Injektion in eine **automatisch injizierte** Klasse ist genauso einfach wie das *Platzieren einer* entsprechenden *@Inject*-Annotation für ein Feld oder einen Konstruktor. Um zum Beispiel *Configuration* in eine Steuerung mit Property-Injection einzuspeisen:

```
@Inject
private Configuration configuration;
```

oder mit Konstruktorinjektion:

```
private Configuration configuration;
@Inject
public MyController(Configuration configuration) {
    this.configuration = configuration;
}
```

Die Injektion in eine **benutzerdefinierte** Klasse, die für **DI** registriert ist, sollte genauso wie für automatisch injizierte Klasse erfolgen - mit der Annotation *@Inject* .

Die Injektion von einer **benutzerdefinierten** Klasse, die nicht an **DI** gebunden ist, sollte durch expliziten Aufruf eines Injektors mit *Play.current (). Injector () erfolgen* . Um beispielsweise *Configuration* in eine benutzerdefinierte Klasse einzufügen, definieren Sie ein Konfigurationsdatenelement wie folgt:

```
private Configuration configuration =
Play.current().injector().instanceOf(Configuration.class);
```

Kundenspezifische Bindung

Die benutzerdefinierte Injektionsbindung kann mit der **@ImplementedBy-** Anmerkung oder auf programmatische Weise mit dem **Guice-Modul vorgenommen werden** .

Injektion mit der @ImplementedBy-Anmerkung

Die Injektion mit der @ImplementedBy-Annotation ist der einfachste Weg. Das folgende Beispiel zeigt einen Dienst, der eine Fassade zum **Cachen** bereitstellt.

1. Der Dienst wird von einer Schnittstelle *CacheProvider* wie folgt definiert:

```
@ImplementedBy(RunTimeCacheProvider.class)
public interface CacheProvider {
   CacheApi getCache();
}
```

2. Der Service wird von einer Klasse RunTimeCacheProvider implementiert:

```
public class RunTimeCacheProvider implements CacheProvider {
   @Inject
   private CacheApi appCache;
   @Override
   public public CacheApi getCache() {
     return appCache;
   }
}
```

Hinweis : Das *appCache-* Datenelement wird bei der Erstellung einer *RunTimeCacheProvider-* Instanz *eingefügt* .

3. Der Cache-Inspektor ist als Mitglied eines Controllers mit der *@Inject-* Annotation definiert und wird vom Controller aufgerufen:

```
public class HomeController extends Controller {
  @Inject
  private CacheProvider cacheProvider;
  ...
  public Result getCacheData() {
        Object cacheData = cacheProvider.getCache().get("DEMO-KEY");
        return ok(String.format("Cache content:%s", cacheData));
  }
```

Die Injektion mit der *@ImplementedBy-* Annotation erstellt die feste Bindung: *CacheProvider* im obigen Beispiel wird immer mit *RunTimeCacheProvider* instanziiert. Eine solche Methode passt nur für einen Fall, wenn eine Schnittstelle mit einer einzelnen Implementierung vorhanden ist. Es kann für eine Schnittstelle mit mehreren Implementierungen oder einer als Singleton implementierten Klasse ohne abstrakte Schnittstelle nicht helfen. Ehrlich gesagt, @ImplementedBy wird in seltenen Fällen verwendet, wenn dies der Fall ist. Es ist wahrscheinlicher, programmatisches Binden mit **Guice-Modul zu verwenden** .

Injektionsbindung mit einem Standard-Play-Modul

Das Standard-Play-Modul ist eine Klasse mit dem Namen *Module* im Hauptprojektverzeichnis, die wie folgt definiert ist:

```
import com.google.inject.AbstractModule;
public class Module extends AbstractModule {
  @Override
  protected void configure() {
      // bindings are here
  }
}
```

Hinweis : Das obige Snippet zeigt die Bindung innerhalb von configure, aber natürlich werden auch andere Bindungsmethoden berücksichtigt.

Für das programmatische Binden von *CacheProvider* an *RunTimeCacheProvider* :

1. Entfernen Sie die *@ImplementedBy-* Annotation aus der Definition des *CacheProvider* :

```
public interface CacheProvider {
  CacheApi getCache();
}
```

2. Implementieren Sie das Modul wie folgt *konfigurieren* :

```
public class Module extends AbstractModule {
  @Override
  protected void configure() {
    bind(CacheProvider.class).to(RunTimeCacheProvider.class);
  }
}
```

Flexible Injection-Bindung mit einem Standard-Play-Modul

RunTimeCacheProvider funktioniert in *JUnit-* Tests mit gefälschter Anwendung nicht *ordnungsgemäß* (siehe *Abschnitt* Unit-Tests). Daher ist die unterschiedliche Implementierung von *CacheProvider* für *Komponententests* erforderlich. Die Injektionsbindung sollte der Umgebung angepasst werden.

Lassen Sie uns ein Beispiel sehen.

1. Die Klasse *FakeCache* bietet eine Stub-Implementierung von *CacheApi* , die während der Ausführung von Tests verwendet werden kann (deren Implementierung ist nicht so interessant - es handelt sich lediglich um eine Karte).
2. Die Klasse *FakeCacheProvider* implementiert *CacheProvider* zur Ausführung von Tests:

```
public class FakeCacheProvider implements CacheProvider {
  private final CacheApi fakeCache = new FakeCache();
  @Override
  public CacheApi getCache() {
    return fakeCache;
  }
```

```
}
```

2. Das Modul wird wie folgt implementiert:

```
public class Module extends AbstractModule {
  private final Environment environment;
  public Module(Environment environment, Configuration configuration) {
    this.environment = environment;
  }
  @Override
  protected void configure() {
      if (environment.isTest() ) {
       bind(CacheProvider.class).to(FakeCacheProvider.class);
     }
     else {
       bind(CacheProvider.class).to(RuntimeCacheProvider.class);
     }
  }
}
```

Das Beispiel ist nur für Bildungszwecke gut. Das Binden für Tests innerhalb des Moduls ist nicht die beste Methode, da dies zwischen Anwendung und Test koppelt. Das Binden von Tests sollte eher durch Tests selbst erfolgen, und das Modul sollte sich nicht mit der testspezifischen Implementierung befassen. Sehen Sie, wie Sie dies in ... verbessern können.

Injektionsbindung mit einem benutzerdefinierten Modul

Ein benutzerdefiniertes Modul ist dem Standard-Play-Modul sehr ähnlich. Der Unterschied besteht darin, dass er einen beliebigen Namen haben kann und zu jedem Paket gehören kann. Zu den Paketmodulen gehört beispielsweise ein Modul OnStartupModule.

```
package modules;
import com.google.inject.AbstractModule;
public class OnStartupModule extends AbstractModule {
   @Override
   protected void configure() {
       ...
   }
}
```

Ein benutzerdefiniertes Modul sollte explizit für den Aufruf von Play aktiviert werden. Für das Modul *OnStartupModule* sollte Folgendes in *application.conf* eingefügt werden:

```
play.modules.enabled += "modules.OnStartupModule"
```

Abhängigkeitsinjektion - Java online lesen:
https://riptutorial.com/de/playframework/topic/6060/abhangigkeitsinjektion---java

Kapitel 3: Abhängigkeitsinjektion - Scala

Syntax

- Klasse MyClassUsingAnother @Inject () (myOtherClassInjected: MyOtherClass) {(...)}
- @Singleton-Klasse MyClassThatShouldBeASingleton (...)

Examples

Grundlegende Verwendung

Eine typische Einzelklasse:

```
import javax.inject._
@Singleton
class BurgersRepository {
    // implementation goes here
}
```

Eine andere Klasse, die Zugriff auf die erste Klasse erfordert.

```
import javax.inject._
class FastFoodService @Inject() (burgersRepository: BurgersRepository){
    // implementation goes here
    // burgersRepository can be used
}
```

Schließlich ein Controller, der den letzten verwendet. Da wir den FastFoodService nicht als Singleton gekennzeichnet haben, wird bei jeder Injektion eine neue Instanz erstellt.

```
import javax.inject._
import play.api.mvc._
@Singleton
class EatingController @Inject() (fastFoodService: FastFoodService) extends Controller {
    // implementation goes here
    // fastFoodService can be used
}
```

Play-Klassen einspritzen

Sie müssen häufig auf Instanzen von Klassen aus dem Framework selbst (wie dem WSClient oder der Konfiguration) zugreifen. Sie können sie in Ihre eigenen Klassen injizieren:

```
class ComplexService @Inject()(
  configuration: Configuration,
  wsClient: WSClient,
  applicationLifecycle: ApplicationLifecycle,
  cacheApi: CacheApi,
  actorSystem: ActorSystem,
```

```
  executionContext: ExecutionContext
  ) {
  // Implementation goes here
  // you can use all the injected classes :
  //
  // configuration to read your .conf files
  // wsClient to make HTTP requests
  // applicationLifecycle to register stuff to do when the app shutdowns
  // cacheApi to use a cache system
  // actorSystem to use AKKA
  // executionContext to work with Futures
}
```

Einige, wie der ExecutionContext, sind wahrscheinlich einfacher zu verwenden, wenn sie als implizit importiert werden. Fügen Sie sie einfach in eine zweite Parameterliste im Konstruktor ein:

```
class ComplexService @Inject()(
  configuration: Configuration,
  wsClient: WSClient
  )(implicit executionContext: ExecutionContext) {
  // Implementation goes here
  // you can still use the injected classes
  // and executionContext is imported as an implicit argument for the whole class
}
```

Definieren von benutzerdefinierten Bindungen in einem Modul

Die grundlegende Verwendung der Abhängigkeitseinspritzung erfolgt durch die Anmerkungen. Wenn Sie die Dinge etwas anpassen müssen, benötigen Sie benutzerdefinierten Code, um genauer anzugeben, wie einige Klassen instanziiert und eingefügt werden sollen. Dieser Code geht in ein so genanntes Modul über.

```
import com.google.inject.AbstractModule
// Play will automatically use any class called `Module` that is in the root package
class Module extends AbstractModule {

  override def configure() = {
    // Here you can put your customisation code.
    // The annotations are still used, but you can override or complete them.

    // Bind a class to a manual instantiation of it
    // i.e. the FunkService needs not to have any annotation, but can still
    // be injected in other classes
    bind(classOf[FunkService]).toInstance(new FunkService)

    // Bind an interface to a class implementing it
    // i.e. the DiscoService interface can be injected into another class
    // the DiscoServiceImplementation is the concrete class that will
    // be actually injected.
    bind(classOf[DiscoService]).to(classOf[DiscoServiceImplementation])

    // Bind a class to itself, but instantiates it when the application starts
    // Useful to executes code on startup
    bind(classOf[HouseMusicService]).asEagerSingleton()
  }
```

```
}
```

Abhängigkeitsinjektion - Scala online lesen:
https://riptutorial.com/de/playframework/topic/3020/abhangigkeitsinjektion---scala

Kapitel 4: Bau und Verpackung

Syntax

- Aktivator dist

Examples

Fügen Sie der Distribution ein Verzeichnis hinzu

So fügen Sie beispielsweise ein Verzeichnis `scripts` auf das Verteilungspaket:

1. Fügen Sie das Projekt einen Ordner **Scripts**
2. Fügen Sie oben auf der build.sbt Folgendes hinzu:

```
import NativePackagerHelper._
```

3. Fügen Sie in build.sbt eine Zuordnung zum neuen Verzeichnis hinzu:

```
mappings in Universal ++= directory("scripts")
```

4. Erstellen Sie das Distributionspaket mit ***Activator dist*** . Das neu erstellte Archiv in `target/universal/` sollte das neue Verzeichnis enthalten.

Bau und Verpackung online lesen: https://riptutorial.com/de/playframework/topic/6642/bau-und-verpackung

Kapitel 5: Einrichten Ihrer bevorzugten IDE

Examples

IntelliJ IDEA

Voraussetzungen

1. Intellij IDEA installiert (Community oder Ultimate Edition)
2. Scala Plugin in IntelliJ installiert
3. Ein Standard-Play-Projekt, das beispielsweise mit Activator erstellt wurde (`activator new [nameoftheproject] play-scala)` .

Projekt öffnen

1. Öffnen Sie die IntelliJ IDEA
2. Gehen Sie ins Menü `File` > `Open ...` > klicken Sie auf den gesamten Ordner [Name des Projekts]> `OK`
3. Es öffnet sich ein Popup mit einigen Optionen. Die Standardwerte sind in den meisten Fällen gut genug. Wenn Sie sie nicht mögen, können Sie sie später anderswo ändern. Klicken Sie auf `OK`
4. Intellij IDEA wird ein wenig nachdenken und schlägt dann ein weiteres Popup vor, um die Module auszuwählen, die im Projekt ausgewählt werden sollen. Standardmäßig sollten zwei Module `root` und `root-build` ausgewählt sein. Ändern Sie nichts und klicken Sie auf `OK` .
5. IntelliJ öffnet das Projekt. Sie können beginnen, die Dateien anzuzeigen, während IntelliJ ein wenig nachdenkt, wie Sie in der Statusleiste unten sehen sollten. Dann sollte es endlich fertig sein.

Ausführen der Anwendungen von Intellij

Von dort verwenden einige Leute die IDE nur zum Anzeigen / Bearbeiten des Projekts, während sie die `sbt` Befehlszeile zum Kompilieren / Ausführen / Starten von Tests verwenden. Andere ziehen es vor, diese von Intellij aus zu starten. Dies ist erforderlich, wenn Sie den Debug-Modus verwenden möchten. Schritte :

1. Menü `Run` > `Edit configurations...`
2. Klicken Sie im Popup auf das `+` oben links> Choose `Play 2 App` in der Liste
3. Benennen Sie die Konfiguration, zum Beispiel [Name Ihres Projekts]. Übernehmen Sie die Standardeinstellungen und klicken Sie auf `OK` .
4. Über das Menü " `Run` oder die Schaltflächen in der Benutzeroberfläche können Sie nun mit dieser Konfiguration " `Run` oder " `Debug` ausführen. `Run` startet die App einfach so, als würden

Sie `sbt run` von der Kommandozeile `sbt run` . `Debug` wird dasselbe tun, aber Sie können Haltepunkte im Code platzieren, um die Ausführung zu unterbrechen und zu analysieren, was passiert.

Option zum automatischen Importieren

Dies ist eine Option global für das Projekt, dass zum Zeitpunkt der Erstellung zur Verfügung steht und danach kann im Menü geändert werden `Intellij IDEA` > `Preferences` > `Build, Execution, Deployment` > `Build tools` - `SBT Project-level settings Use auto-import Build tools` > `SBT` > `Project-level settings` > `Use auto-import` Sie `Use auto-import` .

Diese Option hat nichts mit den `import` im Scala-Code zu tun. Sie legt fest, was Intellij IDEA tun soll, wenn Sie die Datei `build.sbt` bearbeiten. Wenn der automatische Import aktiviert ist, analysiert Intellij IDEA die neue Build-Datei sofort und aktualisiert die Projektkonfiguration automatisch. Es wird schnell ärgerlich, da dieser Vorgang teuer ist und Intellij verlangsamt, wenn Sie noch an der Build-Datei arbeiten. Wenn der automatische Import deaktiviert ist, müssen Sie Intellij manuell mitteilen, dass Sie die `build.sbt` bearbeitet `build.sbt` und möchten, dass die Projektkonfiguration aktualisiert wird. In den meisten Fällen werden Sie in einem temporären Popup gefragt, ob Sie dies möchten. Wechseln Sie andernfalls zum SBT-Fenster in der Benutzeroberfläche und klicken Sie auf das blaue Pfeilsymbol, um die Aktualisierung zu erzwingen.

Eclipse als Play IDE - Java, Abspielen 2.4, 2.5

Einführung

Play hat mehrere Plugins für verschiedene IDEs. Das **Eclipse-** Plugin ermöglicht die Umwandlung einer Play-Anwendung in ein funktionierendes Eclipse-Projekt mit dem Befehlsaktivator *Eclipse* . Das Eclipse-Plugin kann pro Projekt oder global pro **SBT-** Benutzer festgelegt werden. Es hängt von der Teamarbeit ab, welcher Ansatz verwendet werden soll. Wenn das gesamte Team die Eclipse-IDE verwendet, kann das Plugin auf Projektebene festgelegt werden. Sie müssen die Eclipse-Version herunterladen, die Scala und Java 8 unterstützt: **Luna** oder **Mars** - von http://scala-ide.org/download/sdk.html .

Festlegen der Eclipse-IDE für jedes Projekt

So importieren Sie die Play-Anwendung in Eclipse:

1. Fügen Sie eclipse plugin zu *project / plugins.sbt hinzu* :

```
//Support Play in Eclipse
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")
```

2. Fügen Sie in *build.sbt* ein Flag hinzu, das die Kompilierung erzwingt, wenn der Befehl

Eclipse ausgeführt wird:

```
EclipseKeys.preTasks := Seq(compile in Compile)
```

3. Stellen Sie sicher, dass ein Benutzer-Repository-Pfad in der Datei {user root} .sbt \ repositorys das richtige Format hat. Die richtigen Werte für die Eigenschaften *activator-launcher-local* und *activator-local* sollten mindestens drei Schrägstriche enthalten, wie im Beispiel:

```
activator-local: file:////${activator.local.repository-C:/Play-2.5.3/activator-dist-
1.3.10//repository},
[organization]/[module]/(scala_[scalaVersion]/)(sbt_[sbtVersion]/)[revision]/[type]s/[artifact](-
[classifier]).[ext]
activator-launcher-local: file:////${activator.local.repository-${activator.home-
${user.home}/.activator}/repository},
[organization]/[module]/(scala_[scalaVersion]/)(sbt_[sbtVersion]/)[revision]/[type]s/[artifact](-
[classifier]).[ext]
```

4. Kompilieren Sie die Anwendung:

```
activator compile
```

5. Bereiten Sie ein Eclipse-Projekt für die neue Anwendung vor mit:

```
activator eclipse
```

Nun kann das Projekt über **vorhandene Projekte in Workspace** in Eclipse importiert werden.

Anhängen von Play source an Eclipse

1. Fügen Sie der *build.sbt hinzu* :

```
EclipseKeys.withSource := true
```

2. Kompilieren Sie das Projekt

Eclipse-IDE global einstellen

Fügen Sie die **sbt-** Benutzereinstellung hinzu:

1. Erstellen Sie unter dem Benutzerstammverzeichnis den Ordner *.sbt \ 0.13 \ plugins* und eine Datei *plugins.sbt* . Zum Beispiel für Windows-Benutzer **asch** :

```
c:\asch\.sbt\0.13\plugins\plugins.sbt
```

2. Fügen Sie eclipse plugin zu *plugins.sbt hinzu* :

```
//Support Play in Eclipse
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")
```

3. Erstellen Sie im Benutzerverzeichnis *.sbt* eine Datei *sbteclipse.sbt* . Zum Beispiel für Windows-Benutzer **asch** :

```
c:\asch\.sbt\0.13\sbteclipse.sbt
```

4. Setzen Sie in *sbteclipse.sbt* ein Flag, das die Kompilierung erzwingt, wenn der **Eclipse-** Befehl des **Aktivators** ausgeführt wird:

```
import com.typesafe.sbteclipse.plugin.EclipsePlugin.EclipseKeys
EclipseKeys.preTasks := Seq(compile in Compile)
```

5. Fügen Sie optional weitere **EclipseKeys-** Einstellungen hinzu.

Debuggen von Eclipse

Starten Sie zum Debuggen die Anwendung mit dem Standardport 9999:

```
activator -jvm-debug run
```

oder mit dem anderen Hafen:

```
activator -jvm-debug [port] run
```

In der Eklipse:

1. Klicken Sie mit der rechten Maustaste auf das Projekt und wählen Sie **Debug As** , **Debug Configurations aus** .
2. Klicken Sie im Dialogfeld **Debug-Konfigurationen** mit der rechten Maustaste auf **Remote-Java-Anwendung,** und wählen Sie **Neu aus** .
3. Ändern Sie Port in relevant (9999, wenn der Standard-Debug-Port verwendet wurde) und klicken Sie auf **Übernehmen** .

Ab jetzt können Sie auf **Debug** klicken, um eine Verbindung zur laufenden Anwendung **herzustellen** . Durch das Beenden der Debugsitzung wird der Server nicht angehalten.

Eclipse IDE

Voraussetzungen

1. Java8 (1.8.0_91)
2. Eclipse Neon (JavaScript und Web Developer)
3. Play Framework 2.5.4

Scala in Eclipse installieren

1. Starten Sie die Eclipse
2. Öffnen Sie `Help` > `Eclipse Marketplace`
3. Geben Sie `Scala` in `Find`
4. Installieren Sie die Scala IDE

Sbteclipse einrichten

1. Play-Projekt öffnen `.\project\ plugins.sbt`
2. Fügen Sie den folgenden Befehl in `plugins.sbt` , um das Eclipse-Projekt zu konvertieren

   addSbtPlugin ("com.typesafe.sbteclipse"% "sbteclipse-plugin"% "4.0.0")

3. Befehl öffnen und Projekt abspielen, z. B. `cd C:\play\play-scala` . Geben Sie Folgendes in der Befehlszeile ein

   Aktivator-Eklipse

Projekt importieren

1. Gehen Sie zum Menü `File` > `Import` in Eclipse
2. Wählen Sie `Existing Projects into Workspace`
3. Wählen Sie das Stammverzeichnis aus

Jetzt kann Ihr Projekt in der Eclipse IDE angezeigt und bearbeitet werden.

Einrichten Ihrer bevorzugten IDE online lesen:
https://riptutorial.com/de/playframework/topic/4437/einrichten-ihrer-bevorzugten-ide

Kapitel 6: Glatt

Examples

Slick Code für den Einstieg

`build.sbt` Sie in `build.sbt` sicher, dass Sie Folgendes angeben (hier für Mysql und PostGreSQL):

```
"mysql" % "mysql-connector-java" % "5.1.20",
"org.postgresql" % "postgresql" % "9.3-1100-jdbc4",
"com.typesafe.slick" %% "slick" % "3.1.1",
"com.typesafe.play" %% "play-slick" % "1.1.1"
```

Fügen Sie in Ihrer `application.conf` Folgendes hinzu:

```
mydb.driverjava="slick.driver.MySQLDriver$"
mydb.driver="com.mysql.jdbc.Driver"
mydb.url="jdbc:mysql://hostaddress:3306/dbname?zeroDateTimeBehavior=convertToNull"
mydb.user="username"
mydb.password="password"
```

Erstellen Sie für eine RDBMS-unabhängige Architektur ein Objekt wie das folgende

```
package mypackage

import slick.driver.MySQLDriver
import slick.driver.PostgresDriver

object SlickDBDriver{
  val env = "something here"
  val driver = env match{
    case "postGreCondition" => PostgresDriver
    case _                  => MySQLDriver
  }
}
```

beim Erstellen eines neuen neuen Modells:

```
import mypackage.SlickDBDriver.driver.api._
import slick.lifted.{TableQuery, Tag}
import slick.model.ForeignKeyAction

case class MyModel(
  id: Option[Long],
  name: String
) extends Unique


class MyModelDB(tag: Tag) extends IndexedTable[MyModel](tag, "my_table"){
  def id              = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def name            = column[String]("name")
```

```
  def * = (id.? , name) <> ((MyModel.apply _).tupled, MyModel.unapply _)
}

class MyModelCrud{
   import play.api.Play.current

   val dbConfig = DatabaseConfigProvider.get[JdbcProfile](Play.current)
   val db =  dbConfig.db

   val query = TableQuery[MyModelDB]

   // SELECT * FROM my_table;
   def list = db.run{query.result}
}
```

Ausgabe-DDL

Das Wichtigste bei der Verwendung von Slick ist, möglichst wenig SQL-Code zu schreiben. Nachdem Sie Ihre Tabellendefinition geschrieben haben, möchten Sie die Tabelle in Ihrer Datenbank erstellen.

Wenn Sie `val table = TableQuery[MyModel]` können Sie die Tabellendefinition (SQL-Code - DDL) mit folgendem Befehl `val table = TableQuery[MyModel]` :

```
import mypackage.SlickDBDriver.driver.api._
table.schema.createStatements
```

Glatt online lesen: https://riptutorial.com/de/playframework/topic/4604/glatt

Kapitel 7: Java - Hallo Welt

Bemerkungen

- Dieses Tutorial richtet sich an Play auf einem Linux / MacOS-System

Examples

Erstellen Sie Ihr erstes Projekt

Um ein neues Projekt zu erstellen, benutze den folgenden Befehl (`HelloWorld` ist der Name des Projekts und `play-java` ist die Vorlage)

```
$ ~/activator-1.3.10-minimal/bin/activator new HelloWorld play-java
```

Sie sollten eine ähnliche Ausgabe wie diese erhalten

```
Fetching the latest list of templates...

OK, application "HelloWorld" is being created using the "play-java" template.

To run "HelloWorld" from the command line, "cd HelloWorld" then:
/home/YourUserName/HelloWorld/activator run

To run the test for "HelloWorld" from the command line, "cd HelloWorld" then:
/home/YourUserName/HelloWorld/activator test

To run the Activator UI for "HelloWorld" from the command line, "cd HelloWorld" then:
/home/YourUserName/HelloWorld/activator ui
```

> Das Projekt wird im aktuellen Verzeichnis erstellt (in diesem Fall war es mein Home-Ordner)

Wir können jetzt mit der Bewerbung beginnen

Holen Sie sich den Aktivator

Der erste Schritt auf Ihrer Reise in die Play Framework-Welt ist das Herunterladen von Activator. Activator ist ein Tool zum Erstellen, Erstellen und Verteilen von Play Framework-Anwendungen.

Activator kann von Play-Downloads heruntergeladen werden (hier werde ich Version 1.3.10 verwenden)

Nachdem Sie die Datei heruntergeladen haben, extrahieren Sie den Inhalt in ein Verzeichnis, in dem Sie Schreibzugriff haben, und wir können loslegen

> In dieser Anleitung werde ich davon ausgehen, dass Activator in Ihren Home-Ordner extrahiert wurde

Der erste Lauf

Als wir unser Projekt erstellten, erzählte uns Activator, wie wir unsere Anwendung ausführen können

```
To run "HelloWorld" from the command line, "cd HelloWorld" then:
    /home/YourUserName/HelloWorld/activator run
```

Hier gibt es eine kleine Falle: Die ausführbare Datei des `activator` befindet sich nicht in unserer Projektwurzel, sondern im `bin/activator` . Wenn Sie Ihr aktuelles Verzeichnis in Ihr Projektverzeichnis geändert haben, können Sie es auch ausführen

```
bin/activator
```

Activator lädt jetzt die erforderlichen Abhängigkeiten herunter, um Ihr Projekt zu kompilieren und auszuführen. Dies kann je nach Verbindungsgeschwindigkeit einige Zeit dauern. Hoffentlich erhalten Sie eine Aufforderung

```
[HelloWorld] $
```

Wir können unser Projekt jetzt mit `~run` : Dadurch wird Activator angewiesen, unser Projekt auszuführen und auf Änderungen zu achten. Wenn sich etwas ändert, werden die benötigten Teile neu kompiliert und die Anwendung neu gestartet. Sie können diesen Vorgang stoppen, indem Sie Strg + D (zurück zur Activator-Shell) oder Strg + D (zu Ihrer OS-Shell) drücken.

```
[HelloWorld] $ ~run
```

Das Spiel lädt jetzt weitere Abhängigkeiten herunter. Nachdem dieser Vorgang abgeschlossen ist, sollte Ihre App einsatzbereit sein:

```
-- (Running the application, auto-reloading is enabled) ---

[info] p.c.s.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)
```

Wenn Sie in Ihrem Browser zu localhost: 9000 navigieren, sollte die Startseite des Play-Frameworks angezeigt werden

localhost:9000

Your new application is ready.

Welcome to Play

Congratulations, you've just created a new Play application. This page will help you with the next few steps.

You're using Play 2.5.4

Herzlichen Glückwunsch, Sie können jetzt einige Änderungen an Ihrer Anwendung vornehmen!

Die "Hallo Welt" in der Hallo Welt

Ein "Hello World" verdient diesen Namen nicht, wenn er keine Hello World-Nachricht enthält. Also machen wir eins.

`app/controllers/HomeController.java` Sie in der Datei `app/controllers/HomeController.java` die folgende Methode hinzu:

```
public Result hello() {
    return ok("Hello world!");
}
```

`conf/routes` in Ihrer `conf/routes` Datei am Ende der Datei Folgendes hinzu:

```
GET     /hello                      controllers.HomeController.hello
```

Wenn Sie sich Ihr Terminal ansehen, sollten Sie feststellen, dass Play Ihre Anwendung kompiliert, während Sie die Änderungen vornehmen und die App erneut laden:

```
[info] Compiling 4 Scala sources and 1 Java source to
/home/YourUserName/HelloWorld/target/scala-2.11/classes...
[success] Compiled in 4s
```

Navigieren zu localhost: 9000 / hallo , wir bekommen endlich unsere hallo Weltnachricht

```
Hello world!
```

Java - Hallo Welt online lesen: https://riptutorial.com/de/playframework/topic/5887/java---hallo-welt

Kapitel 8: Java - Mit JSON arbeiten

Bemerkungen

Spieldokumentation: https://www.playframework.com/documentation/2.5.x/JavaJsonActions

Examples

Manuelles Erstellen von JSON

```
import play.libs.Json;


public JsonNode createJson() {
    // {"id": 33, "values": [3, 4, 5]}
    ObjectNode rootNode = Json.newObject();
    ArrayNode listNode = Json.newArray();

    long values[] = {3, 4, 5};
    for (long val: values) {
        listNode.add(val);
    }

    rootNode.put("id", 33);
    rootNode.set("values", listNode);
    return rootNode;
}
```

Json wird aus String / Datei geladen

```
import play.libs.Json;
// (...)
```

Laden einer Datei aus Ihrem öffentlichen Ordner

```
// Note: "app" is an play.Application instance
JsonNode node = Json.parse(app.resourceAsStream("public/myjson.json"));
```

Laden von einer Zeichenfolge

```
String myStr = "{\"name\": \"John Doe\"}";
JsonNode node = Json.parse(myStr);
```

Übertragen eines JSON-Dokuments

In den folgenden Beispielen enthält `json` ein JSON-Objekt mit den folgenden Daten:

```
[
  {
    "name": "John Doe",
    "work": {
      "company": {
        "name": "ASDF INC",
        "country": "USA"
      },
      "cargo": "Programmer"
    },
    "tags": ["java", "jvm", "play"]
  },
  {
    "name": "Bob Doe",
    "work": {
      "company": {
        "name": "NOPE INC",
        "country": "AUSTRALIA"
      },
      "cargo": "SysAdmin"
    },
    "tags": ["puppet", "ssh", "networking"],
    "active": true
  }
]
```

Liefert den Namen eines Benutzers (unsicher)

```
JsonNode node = json.get(0).get("name"); // --> "John Doe"
// This will throw a NullPointerException, because there is only two elements
JsonNode node = json.get(2).get("name"); // --> *crash*
```

Holen Sie sich den Benutzernamen (sicherer Weg)

```
JsonNode node1 = json.at("/0/name"); // --> TextNode("John Doe")
JsonNode node2 = json.at("/2/name"); // --> MissingNode instance
if (! node2.isMissingNode()) {
    String name = node2.asText();
}
```

Holen Sie sich das Land, in dem der erste Benutzer arbeitet

```
JsonNode node2 = json.at("/0/work/company/country"); // TextNode("USA")
```

Holen Sie sich alle Länder

```
List<JsonNode> d = json.findValues("country"); // List(TextNode("USA"), TextNode("AUSTRALIA"))
```

Finde jeden Benutzer, der das Attribut "aktiv" enthält

```
List<JsonNode> e = json.findParents("active"); // List(ObjectNode("Bob Doe"))
```

Konvertierung zwischen JSON- und Java-Objekten (Basis)

Standardmäßig versucht Jackson (die Bibliothek, die Play JSON verwendet), jedes öffentliche Feld einem json-Feld mit demselben Namen zuzuordnen. Wenn das Objekt Getter / Setter hat, wird daraus der Name abgeleitet. Wenn Sie eine `Book` Klasse mit einem privaten Feld zum Speichern der ISBN haben und get / set-Methoden mit dem Namen `getISBN/setISBN` , wird Jackson dies `getISBN/setISBN`

- Erstellen Sie ein JSON-Objekt mit dem Feld "ISBN", wenn Sie von Java in JSON konvertieren
- Verwenden Sie die `setISBN` Methode, um das isbn-Feld im Java-Objekt zu definieren (wenn das JSON-Objekt ein "ISBN" -Feld hat).

Java-Objekt aus JSON erstellen

```
public class Person {
    String id, name;
}

JsonNode node = Json.parse("{\"id\": \"3S2F\", \"name\", \"Salem\"}");
Person person = Json.fromJson(node, Person.class);
System.out.println("Hi " + person.name); // Hi Salem
```

Erstellen Sie ein JSON-Objekt aus einem Java-Objekt

```
// "person" is the object from the previous example
JsonNode personNode = Json.toJson(person)
```

Erstellen einer JSON-Zeichenfolge aus einem JSON-Objekt

```
// personNode comes from the previous example
String json = personNode.toString();
// or
String json = Json.stringify(json);
```

JSON hübsches Drucken

```
System.out.println(personNode.toString());
/* Prints:
{"id":"3S2F","name":"Salem"}
*/

System.out.println(Json.prettyPrint(personNode));
/* Prints:
{
```

```
  "id" : "3S2F",
  "name" : "Salem"
}
*/
```

Java - Mit JSON arbeiten online lesen: https://riptutorial.com/de/playframework/topic/6318/java---mit-json-arbeiten

Kapitel 9: Mit JSON - Scala arbeiten

Bemerkungen

Offizielle Dokumentation Paketdokumentation

Sie können das play json-Paket unabhängig von Play verwenden, indem Sie dazu zählen

`"com.typesafe.play" % "play-json_2.11" % "2.5.3"` in Ihrer `build.sbt` , siehe

- https://mvnrepository.com/artifact/com.typesafe.play/play-json_2.11
- Play JSON Library zu sbt hinzufügen

Examples

JSON manuell erstellen

Sie können einen JSON-Objektbaum (einen `JsValue`) manuell `JsValue`

```
import play.api.libs.json._

val json = JsObject(Map(
  "name" -> JsString("Jsony McJsonface"),
  "age" -> JsNumber(18),
  "hobbies" -> JsArray(Seq(
    JsString("Fishing"),
    JsString("Hunting"),
    JsString("Camping")
  ))
))
```

Oder mit der kürzeren äquivalenten Syntax, basierend auf einigen impliziten Konvertierungen:

```
import play.api.libs.json._

val json = Json.obj(
  "name" -> "Jsony McJsonface",
  "age" -> 18,
  "hobbies" -> Seq(
    "Fishing",
    "Hunting",
    "Camping"
  )
)
```

So erhalten Sie die JSON-Zeichenfolge:

```
json.toString
// {"name":"Jsony McJsonface","age":18,"hobbies":["Fishing","Hunting","Camping"]}
Json.prettyPrint(json)
```

```
//  {
//    "name" : "Jsony McJsonface",
//    "age" : 18,
//    "hobbies" : [ "Fishing", "Hunting", "Camping" ]
//  }
```

Java: JSON-Anforderungen annehmen

```
public Result sayHello() {
    JsonNode json = request().body().asJson();
    if(json == null) {
        return badRequest("Expecting Json data");
    } else {
        String name = json.findPath("name").textValue();
        if(name == null) {
            return badRequest("Missing parameter [name]");
        } else {
            return ok("Hello " + name);
        }
    }
}
```

Java: JSON-Anforderungen mit BodyParser annehmen

```
@BodyParser.Of(BodyParser.Json.class)
public Result sayHello() {
    JsonNode json = request().body().asJson();
    String name = json.findPath("name").textValue();
    if(name == null) {
        return badRequest("Missing parameter [name]");
    } else {
        return ok("Hello " + name);
    }
}
```

Hinweis: Der Vorteil dieser Methode besteht darin, dass Play automatisch mit einem HTTP-Statuscode 400 reagiert, wenn die Anforderung nicht gültig war (Content-Typ wurde auf `application/json` jedoch kein JSON bereitgestellt).

Scala: Manuelles Lesen eines JSON

Wenn Sie eine JSON-Zeichenfolge erhalten:

```
val str =
    """{
    |    "name" : "Jsony McJsonface",
    |    "age" : 18,
    |    "hobbies" : [ "Fishing", "Hunting", "Camping" ],
    |    "pet" : {
    |        "name" : "Doggy",
    |        "type" : "dog"
    |    }
    |}""".stripMargin
```

Sie können es analysieren, um einen JsValue zu erhalten, der den JSON-Baum darstellt

```
val json = Json.parse(str)
```

Durchlaufen Sie den Baum, um nach bestimmten Werten zu suchen:

```
(json \ "name").as[String]          // "Jsony McJsonface"
```

Nützliche Methoden

- `\` , um zu einem bestimmten Schlüssel in einem JSON-Objekt zu gelangen
- `\\` , um zu allen Vorkommen eines bestimmten Schlüssels in einem JSON-Objekt zu gelangen und rekursiv in verschachtelten Objekten zu suchen
- `.apply(idx)` (dh `(idx)`), um zu einem Index in einem Array zu gelangen
- `.as[T]` , um einen genauen Untertyp zu erhalten
- `.asOpt[T]` zu versuchen, auf einen genauen Untertyp `.asOpt[T]` , und gibt None zurück, wenn es sich um den falschen Typ handelt
- `.validate[T]` für den Versuch, einen JSON-Wert in einen genauen Subtyp umzuwandeln, wobei ein JsSuccess oder ein JsError zurückgegeben wird

```
(json \ "name").as[String]          // "Jsony McJsonface"
(json \ "pet" \ "name").as[String]  // "Doggy"
(json \\ "name").map(_.as[String])  // List("Jsony McJsonface", "Doggy")
(json \\ "type")(0).as[String]      // "dog"
(json \ "wrongkey").as[String]      // throws JsResultException
(json \ "age").as[Int]              // 18
(json \ "hobbies").as[Seq[String]]  // List("Fishing", "Hunting", "Camping")
(json \ "hobbies")(2).as[String]    // "Camping"
(json \ "age").asOpt[String]        // None
(json \ "age").validate[String]     // JsError containing some error detail
```

Automatische Zuordnung zu / von Fallklassen

Insgesamt ist die einfachste Methode für die Arbeit mit JSON die Zuordnung einer Fallklasse direkt zu JSON (gleicher Feldname, gleichwertige Typen usw.).

```
case class Person(
  name: String,
  age: Int,
  hobbies: Seq[String],
  pet: Pet
)

case class Pet(
  name: String,
  `type`: String
)

// these macros will define automatically the conversion to/from JSON
// based on the cases classes definition
```

```
implicit val petFormat = Json.format[Pet]
implicit val personFormat = Json.format[Person]
```

Umwandlung in Json

```
val person = Person(
  "Jsony McJsonface",
  18,
  Seq("Fishing", "Hunting", "Camping"),
  Pet("Doggy", "dog")
)

Json.toJson(person).toString
// {"name":"Jsony
McJsonface","age":18,"hobbies":["Fishing","Hunting","Camping"],"pet":{"name":"Doggy","type":"dog"}}
```

Konvertierung von Json

```
val str =
    """{
    |    "name" : "Jsony McJsonface",
    |    "age" : 18,
    |    "hobbies" : [ "Fishing", "Hunting", "Camping" ],
    |     "pet" : {
    |       "name" : "Doggy",
    |       "type" : "dog"
    |     }
    |}""".stripMargin


Json.parse(str).as[Person]
// Person(Jsony McJsonface,18,List(Fishing, Hunting, Camping),Pet(Doggy,dog))
```

Mit JSON - Scala arbeiten online lesen: https://riptutorial.com/de/playframework/topic/2983/mit-json---scala-arbeiten

Kapitel 10: Unit Testing

Examples

Unit Testing - Java, Abspielen 2.4.2.5

Helfer und FakeApplication

Klassenhelfer werden häufig für Komponententests verwendet. Es imitiert eine Play-Anwendung, fälscht HTTP-Anfragen und -Antworten, Sitzung, Cookies - alles, was für Tests benötigt wird. Ein Controller unter dem Test sollte im Kontext einer Play-Anwendung ausgeführt werden. Die *Helpers*- Methode *fakeApplication* stellt eine Anwendung zum Ausführen von Tests bereit. Um zu verwenden , *Helfer* und *fakeApplication* sollte eine Testklasse von *WithApplication* abzuleiten.

Die folgenden *Helpers*- APIs sollten verwendet werden:

```
Helpers.running(Application application, final Runnable block);
Helpers.fakeApplication();
```

Test mit *Helfern* sieht so aus:

```
public class TestController extends WithApplication {
 @Test
 public void testSomething() {
     Helpers.running(Helpers.fakeApplication(), () -> {
         // put test stuff
         // put asserts
     });
 }
}
```

Durch das Hinzufügen von Importanweisungen für Helpers-Methoden wird Code kompakter:

```
 import static play.test.Helpers.fakeApplication;
 import static play.test.Helpers.running;
 ...
 @Test
 public void testSomething() {
      running(fakeApplication(), () -> {
           // put test stuff
           // put asserts
       });
 }
```

}

Controller testen

Nennen wir eine Controller-Methode, die als **Routing-** Methode an die jeweilige URL in den *Routen* gebunden ist. Ein Aufruf einer **gerouteten** Methode wird als Controller- **Aktion bezeichnet** und hat einen Java-Typ *Call* . Play baut für jede **Aktion eine** sogenannte umgekehrte Route auf. Beim Aufruf einer umgekehrten Route wird ein entsprechendes *Anrufobjekt erstellt* . Dieser Reverse-Routing-Mechanismus wird zum Testen von Controllern verwendet.

Um eine Controller- **Aktion** aus dem Test aufzurufen, sollte die folgende Helpers-API verwendet werden:

```
Result result = Helpers.route(Helpers.fakeRequest(Call action));
```

Controller testet Beispiel

1. Die *Routen* :

```
GET /conference/:confId    controllers.ConferenceController.getConfId(confId: String)
POST /conference/:confId/participant
controllers.ConferenceController.addParticipant(confId:String)
```

2. Generierte umgekehrte Routen:

```
controllers.routes.ConferenceController.getConfId(conferenceId)
controllers.routes.ConferenceController.addParticipant(conferenceId)
```

3. Die Methode *getConfId* ist an **GET** gebunden und erhält keinen Body in einer Anfrage. Es kann für den Test aufgerufen werden mit:

```
Result result =
Helpers.route(Helpers.fakeRequest(controllers.routes.ConferenceController.getConfId(conferenceId)
```

4. Die Methode *addParticipant* ist an den **POST** gebunden. Es erwartet, dass Sie eine Stelle in einer Anfrage erhalten. Ihr Aufruf im Test sollte folgendermaßen erfolgen:

```
ParticipantDetails inputData = DataSimulator.createParticipantDetails();
Call action = controllers.routes.ConferenceController.addParticipant(conferenceId);
Result result = route(Helpers.fakeRequest(action).bodyJson(Json.toJson(inputData));
```

Spott mit PowerMock

Um das Verspotten zu aktivieren, sollte eine Testklasse wie folgt kommentiert werden:

```
@RunWith(PowerMockRunner.class)
@PowerMockIgnore({"javax.management.*", "javax.crypto.*"})
public class TestController extends WithApplication {
....
```

Verspottung einer Controller-Aktion

Ein Controller-Aufruf wird mit *RequestBuilder* verspottet:

```
RequestBuilder fakeRequest = Helpers.fakeRequest(action);
```

Für den obigen addParticipant wird eine Aktion mit:

```
RequestBuilder mockActionRequest =
Helpers.fakeRequest(controllers.routes.ConferenceController.addParticipant(conferenceId));
```

So rufen Sie die Controller-Methode auf:

```
Result result = Helpers.route(mockActionRequest);
```

Der ganze Test:

```
@Test
public void testLoginOK() {
 running(fakeApplication(), () -> {
      ///*whatever mocking*/Mockito.when(...).thenReturn(...);
      RequestBuilder mockActionRequest = Helpers.fakeRequest(
           controllers.routes.LoginController.loginAdmin());
      Result result = route(mockActionRequest);
      assertEquals(OK, result.status());
 });
}
```

Verspottung einer Aktion mit JSON-Body

Nehmen wir an, eine Eingabe ist ein Objekt vom Typ *T*. Das Spähen der Aktionsanforderung kann auf verschiedene Arten erfolgen.

Option 1:

```
public static <T> RequestBuilder fakeRequestWithJson(T input, String method, String url) {
  JsonNode jsonNode = Json.toJson(input);
  RequestBuilder fakeRequest = Helpers.fakeRequest(method, url).bodyJson(jsonNode);
  System.out.println("Created fakeRequest="+fakeRequest +",
body="+fakeRequest.body().asJson());
  return fakeRequest;
}
```

Option 2:

```
public static <T> RequestBuilder fakeActionRequestWithJson(Call action, T input) {
  JsonNode jsonNode = Json.toJson(input);
  RequestBuilder fakeRequest = Helpers.fakeRequest(action).bodyJson(jsonNode);
  System.out.println("Created fakeRequest="+fakeRequest +",
body="+fakeRequest.body().asJson());
```

```
  return fakeRequest;
}
```

Verspottung einer Aktion mit dem Header der Basisauthentifizierung

Die Handlungsanforderung spottet:

```
public static final String BASIC_AUTH_VALUE = "dummy@com.com:12345";
public static RequestBuilder fakeActionRequestWithBaseAuthHeader(Call action) {
  String encoded = Base64.getEncoder().encodeToString(BASIC_AUTH_VALUE.getBytes());
  RequestBuilder fakeRequest =
Helpers.fakeRequest(action).header(Http.HeaderNames.AUTHORIZATION,
                                              "Basic " + encoded);
  System.out.println("Created fakeRequest="+fakeRequest.toString() );
  return fakeRequest;
}
```

Verspottung einer Aktion mit Sitzung

Die Handlungsanforderung spottet:

```
public static final String FAKE_SESSION_ID = "12345";
public static RequestBuilder fakeActionRequestWithSession(Call action) {
  RequestBuilder fakeRequest = RequestBuilder fakeRequest =
Helpers.fakeRequest(action).session("sessionId", FAKE_SESSION_ID);
  System.out.println("Created fakeRequest="+fakeRequest.toString() );
  return fakeRequest;
}
```

Die Play *Session*- Klasse ist nur eine Erweiterung der *HashMap* <*String, String*> . Es kann mit einfachem Code verspottet werden:

```
public static Http.Session fakeSession() {
  return new Http.Session(new HashMap<String, String>());
}
```

Unit Testing online lesen: https://riptutorial.com/de/playframework/topic/6192/unit-testing

Kapitel 11: Webservice-Nutzung mit play WSClient

Bemerkungen

Link zur offiziellen Dokumentation: https://www.playframework.com/documentation/2.5.x/ScalaWS

Examples

Grundnutzung (Scala)

HTTP-Anforderungen werden über die WSClient-Klasse gestellt, die Sie als injizierten Parameter in Ihre eigenen Klassen verwenden können.

```
import javax.inject.Inject

import play.api.libs.ws.WSClient

import scala.concurrent.{ExecutionContext, Future}

class MyClass @Inject() (
  wsClient: WSClient
)(implicit ec: ExecutionContext){

  def doGetRequest(): Future[String] = {
    wsClient
      .url("http://www.google.com")
      .get()
      .map { response =>
      // Play won't check the response status,
      // you have to do it manually
      if ((200 to 299).contains(response.status)) {
        println("We got a good response")
        // response.body returns the raw string
        // response.json could be used if you know the response is JSON
        response.body
      } else
        throw new IllegalStateException(s"We received status ${response.status}")
    }
  }
}
```

Webservice-Nutzung mit play WSClient online lesen:
https://riptutorial.com/de/playframework/topic/2981/webservice-nutzung-mit-play-wsclient

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit playframework	Abhinab Kanrar, Anton, asch, Community, implicitdef, James, John, robguinness
2	Abhängigkeitsinjektion - Java	asch
3	Abhängigkeitsinjektion - Scala	asch, implicitdef
4	Bau und Verpackung	JulienD
5	Einrichten Ihrer bevorzugten IDE	Alice, asch, implicitdef
6	Glatt	John
7	Java - Hallo Welt	Salem
8	Java - Mit JSON arbeiten	Salem
9	Mit JSON - Scala arbeiten	Anton, asch, implicitdef, John, Salem
10	Unit Testing	asch
11	Webservice-Nutzung mit play WSClient	implicitdef, John, Salem