EBook Gratuito

APPRENDIMENTO playframework

Free unaffiliated eBook created from
Stack Overflow contributors.

#playframework

work

Sommario

Di

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.

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

Capitolo 1: Iniziare con PlayFramework

Osservazioni

Questa sezione fornisce una panoramica di cosa sono i playframework e perché uno sviluppatore potrebbe volerlo usare.

Dovrebbe anche menzionare tutti i soggetti di grandi dimensioni all'interno di strutture di gioco e collegarsi agli argomenti correlati. Poiché la Documentazione per il gioco è nuova, potrebbe essere necessario creare versioni iniziali di tali argomenti correlati.

Examples

Gioca 1 installazione

Prerequisiti

Per eseguire il framework Play, è necessario Java 6 o versioni successive. Se desideri creare Play dal sorgente, avrai bisogno del client di controllo del codice sorgente Git per recuperare il codice sorgente e Ant per crearlo.

Assicurati di avere Java nel percorso corrente (inserisci `java --version` per verificare)

Play utilizzerà il Java predefinito o quello disponibile sul percorso **$ JAVA_HOME** , se definito.

L'utilità della riga di comando **play** utilizza Python. Quindi dovrebbe funzionare immediatamente su qualsiasi sistema UNIX (tuttavia richiede almeno Python 2.5).

Installazione dal pacchetto binario

Istruzioni generiche

In generale, le istruzioni di installazione sono le seguenti.

1. Installa Java.
2. Scarica l' ultimo pacchetto binario Play ed estrai l'archivio.
3. Aggiungi il comando 'gioca' al tuo percorso di sistema e assicurati che sia eseguibile.

Mac OS X

Java è integrato o installato automaticamente, quindi puoi saltare il primo passaggio.

1. Scarica l'ultimo pacchetto binario di Play ed estrailo in `/Applications` .
2. Modifica `/etc/paths` e aggiungi la riga `/Applications/play-1.2.5` (per esempio).

Un'alternativa su OS X è:

1. Installa HomeBrew
2. Esegui il `brew install play`

Linux

Per installare Java, assicurati di utilizzare il Sun-JDK o OpenJDK (e non gcj, che è il comando Java predefinito su molte distribuzioni Linux)

finestre

Per installare Java, basta scaricare e installare l'ultimo pacchetto JDK. Non è necessario installare Python separatamente, poiché un runtime di Python è in bundle con il framework.

Installazione tramite `sbt`

Se hai già installato `sbt` trovo più semplice creare un progetto Play minimale senza `activator` . Ecco come.

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

Quando i passaggi precedenti sono stati completati, modifica `build.sbt` e aggiungi le seguenti linee

```
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/"
```

Infine, crea un `project` cartella e all'interno crea un file `build.properties` con il riferimento alla versione di Play che desideri utilizzare

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

Questo è tutto! Il tuo progetto è pronto Puoi lanciarlo con `sbt` . Dall'interno di `sbt` hai accesso agli stessi comandi come con l' `activator` .

Iniziare con Play 2.4.x / 2.5.x - Windows, Java

installazioni

Scarica e installa:

1. Java 8: scarica l'installazione pertinente dal sito Oracle .
2. Attivatore: scarica zip da www.playframework.com/download ed estrai i file nella cartella Play di destinazione, ad esempio per:

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

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

Definisci le variabili d'ambiente:

1. **JAVA_HOME** , ad esempio:

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

2. **PLAY_HOME** , ad esempio:

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

3. **SBT_HOME** per esempio:

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

Aggiungi il percorso a tutti e tre i programmi installati alle variabili del percorso:

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

Riproduci la correzione di installazione di 2.5

L'installazione di Play 2.5.3 (l'ultima 2.5 versione stabile) ha un problema minore. Per risolverlo:

1. Modifica il file *activator-dist-1.3.10* \ *bin* \ *activator.bat* e aggiungi il carattere "%" alla fine della riga 55. La riga corretta dovrebbe essere così: *imposta SBT_HOME =% BIN_DIRECTORY%*
2. Crea sotto-directory *conf* sotto la directory root dell'attivatore *activator-dist-1.3.10* .
3. Creare nella directory *conf* un file vuoto denominato *sbtconfig.txt* .

Creazione di una nuova applicazione con CLI

Avviare il *cmd* dalla directory, dove dovrebbe essere creata una nuova applicazione. Il modo più breve per creare una nuova applicazione tramite CLI è fornire un nome e un modello dell'applicazione come argomenti CLI:

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

È possibile eseguire solo:

```
  activator new
```

In questo caso verrà richiesto di selezionare il modello desiderato e il nome dell'applicazione.

Per Play 2.4 aggiungi manualmente a *project / plugins.sbt* :

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

Assicurati di sostituire qui 2.4.x con la versione esatta che desideri utilizzare. Play 2.5 genera automaticamente questa linea.

Assicurati che la versione **sbt** appropriata sia menzionata in project / build.properties. Dovrebbe corrispondere alla versione di **sbt** , installata sulla tua macchina. Ad esempio, per Play2.4.x dovrebbe essere:

```
sbt.version=0.13.8
```

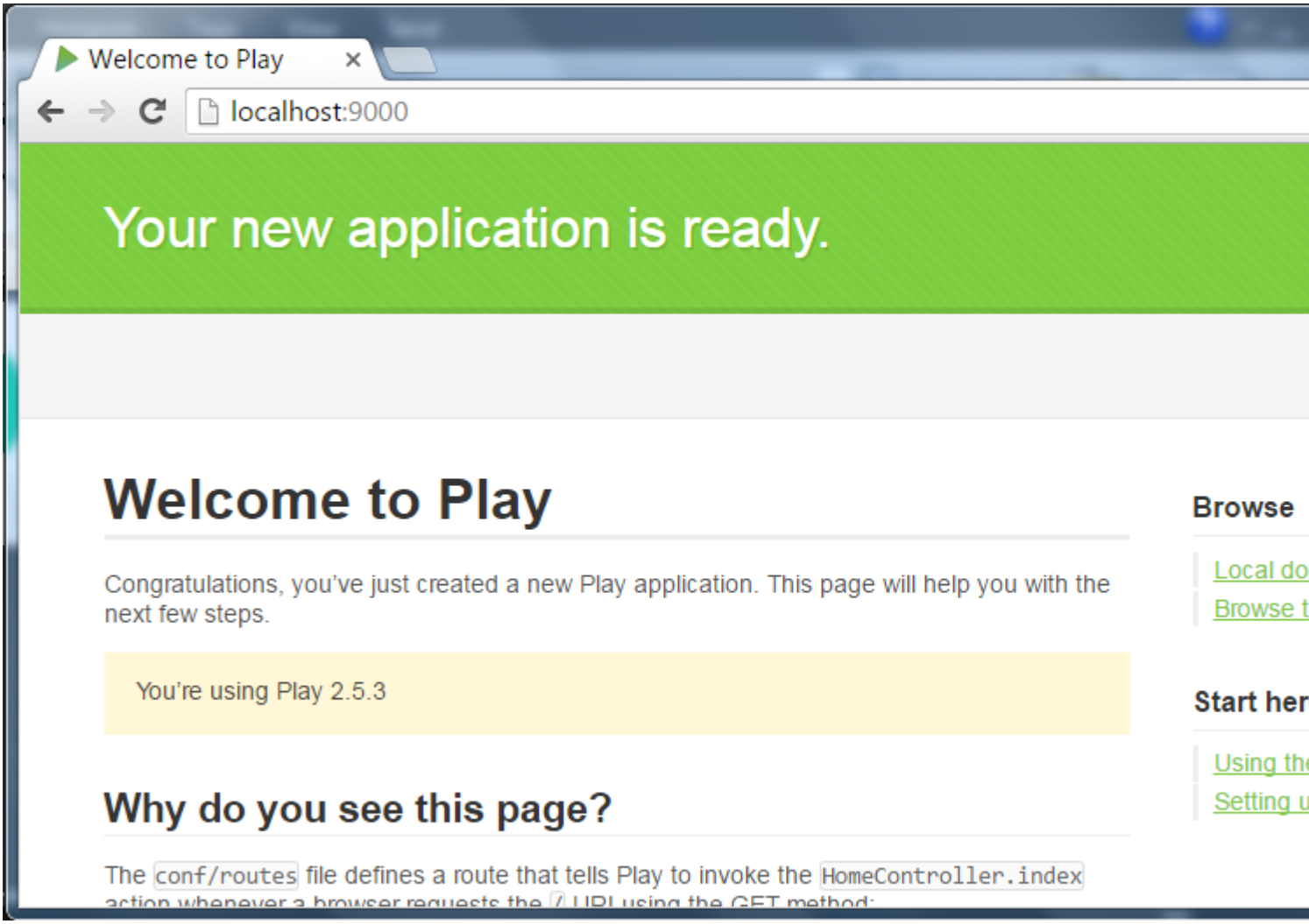
Ecco, ora può essere avviata una nuova applicazione:

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

Dopo un po 'inizierà il server e il seguente prompt dovrebbe apparire sulla console:

```
[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...)
```

Il server per impostazione predefinita è in ascolto sulla porta 9000. È possibile richiederlo da un browser tramite l'URL *http: // localhost: 9000* . Otterrai qualcosa di simile a questo:



Esecuzione dell'attivatore su una porta diversa

Per impostazione predefinita, l'attivatore esegue un'applicazione sulla porta 9000 per http o 443 per https. Per eseguire un'applicazione sulla porta diversa (http):

```
activator "run 9005"
```

Leggi Iniziare con PlayFramework online:
https://riptutorial.com/it/playframework/topic/1052/iniziare-con-playframework

Capitolo 2: chiazza di petrolio

Examples

Slick per iniziare il codice

In `build.sbt` , assicurati di includere (qui per Mysql e 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"
```

Nel tuo file `application.conf` , aggiungi:

```
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"
```

Per avere un'architettura indipendente RDBMS creare un oggetto come il seguente

```
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
  }
}
```

quando si crea un nuovo nuovo modello:

```
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}
}
```

DDL di uscita

L'intero punto dell'utilizzo di slick è scrivere il meno codice SQL possibile. Dopo aver scritto la definizione della tabella, vorrete creare la tabella nel vostro database.

Se hai `val table = TableQuery[MyModel]` Puoi ottenere la definizione della tabella (codice SQL - DDL) con il seguente comando:

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

Leggi chiazza di petrolio online: https://riptutorial.com/it/playframework/topic/4604/chiazza-di-petrolio

Capitolo 3: Edilizia e imballaggio

Sintassi

- attivatore dist

Examples

Aggiungi una directory alla distribuzione

Per aggiungere per esempio un `scripts` directory al pacchetto di distribuzione:

1. Aggiungi al progetto uno **script di** cartelle
2. In cima a build.sbt, aggiungi:

```
import NativePackagerHelper._
```

3. In build.sbt, aggiungi una mappatura alla nuova directory:

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

4. Costruisci il pacchetto di distribuzione con l' ***attivatore dist*** . L'archivio appena creato in `target/universal/` dovrebbe contenere la nuova directory.

Leggi Edilizia e imballaggio online: https://riptutorial.com/it/playframework/topic/6642/edilizia-e-imballaggio

Capitolo 4: Impostazione del tuo IDE preferito

Examples

IntelliJ IDEA

Prerequisiti

1. Intellij IDEA installato (Community o Ultimate edition)
2. Plugin Scala installato in IntelliJ
3. Un progetto di riproduzione standard, creato ad esempio con Activator (`activator new [nameoftheproject] play-scala`) .

Aprendo il progetto

1. Apri IntelliJ IDEA
2. Vai al menu `File` > `Open ...` > fai clic sull'intera cartella [nome del progetto]> `OK`
3. Un popup si apre con alcune opzioni. I valori predefiniti sono abbastanza buoni nella maggior parte dei casi e, se non ti piacciono, puoi cambiarli da qualche altra parte in seguito. Clicca `OK`
4. Intellij IDEA penserà un po ', quindi proporrà un altro popup per selezionare quali moduli selezionare nel progetto. Ci dovrebbero essere due moduli `root` e `root-build` selezionati per impostazione predefinita. Non modificare nulla e fare `OK` su `OK` .
5. IntelliJ aprirà il progetto. È possibile iniziare a visualizzare i file mentre IntelliJ continua a pensare un po 'come si dovrebbe vedere nella barra di stato in basso, quindi dovrebbe essere finalmente pronto.

Esecuzione delle applicazioni da Intellij

Da lì alcune persone usano l'IDE solo per visualizzare / modificare il progetto, mentre usano la riga di comando `sbt` per compilare / eseguire / lanciare test. Altri preferiscono lanciare quelli da Intellij. È richiesto se si desidera utilizzare la modalità di debug. Passi:

1. Menu `Run` > `Edit configurations...`
2. Nel popup, fai clic sul `+` in alto a sinistra> Scegli `Play 2 App` nell'elenco
3. Assegna un nome alla configurazione, ad esempio [nome del tuo progetto]. Lascia le opzioni predefinite e premi `OK` .
4. Dal menu `Run` o dai pulsanti nell'interfaccia utente, ora puoi `Run` o `Run Debug` utilizzando questa configurazione. `Run` avvia semplicemente l'app, come se si `sbt run` dalla riga di comando. `Debug` farà la stessa cosa ma ti permetterà di posizionare i breakpoint nel codice per interrompere l'esecuzione e analizzare cosa sta succedendo.

Opzione di importazione automatica

Questa è un'opzione globale per il progetto, che è disponibile al momento della creazione e successivamente può essere modificata nel menu `Intellij IDEA` > `Preferences` > `Build, Execution, Deployment` > `Build tools` > `SBT` > `Project-level settings` > `Use auto-import` .

Questa opzione non ha nulla a che fare con le istruzioni `import` nel codice Scala. Stabilisce ciò che Intellij IDEA dovrebbe fare quando si modifica il file `build.sbt` . Se l'importazione automatica è attivata, Intellij IDEA analizzerà immediatamente il nuovo file di costruzione e aggiornerà automaticamente la configurazione del progetto. Diventa fastidioso rapidamente poiché questa operazione è costosa e tende a rallentare Intellij quando si sta ancora lavorando sul file di build. Quando viene disattivata l'importazione automatica, è necessario indicare manualmente a Intellij che è stato modificato il `build.sbt` e si desidera aggiornare la configurazione del progetto. Nella maggior parte dei casi apparirà un popup temporaneo per chiederti se desideri farlo. Altrimenti, vai al pannello SBT nell'interfaccia utente e fai clic sul segno di frecce blu per attivare l'aggiornamento.

Eclipse as Play IDE - Java, Play 2.4, 2.5

introduzione

Play ha diversi plugin per IDE diversi. Il plugin **eclipse** consente di trasformare un'applicazione Play in un progetto eclipse funzionante con l' *attivatore di* comando *eclipse* . Il plugin Eclipse può essere impostato per progetto o globalmente per utente **sbt** . Dipende dal lavoro di squadra, che approccio dovrebbe essere usato. Se l'intero team utilizza IDE eclipse, il plug-in può essere impostato a livello di progetto. Devi scaricare la versione di eclipse che supporta Scala e Java 8: **luna** o **mars** - da http://scala-ide.org/download/sdk.html .

Impostazione di IDE eclipse per progetto

Per importare l'applicazione Play in eclissi:

1. Aggiungi il plugin eclipse in *project / plugins.sbt* :

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

2. Aggiungi a *build.sbt* un flag che impone la compilazione a verificarsi quando viene eseguito il comando eclipse:

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

3. Assicurarsi che il percorso di un repository utente nel file {root utente} .sbt \ repository abbia il formato corretto. I valori appropriati per le proprietà *activator-launcher-local* e *activator-*

local dovrebbero avere almeno tre barre come nell'esempio:

```
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. Compila l'applicazione:

```
activator compile
```

5. Preparare un progetto Eclipse per la nuova applicazione con:

```
activator eclipse
```

Ora il progetto è pronto per essere importato in Eclipse tramite **Progetti esistenti nello spazio di lavoro** .

Come allegare Riproduci sorgente a eclissi

1. Aggiungi al *build.sbt* :

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

2. Compila il progetto

Impostazione di IDE eclipse a livello globale

Aggiungi le impostazioni utente di **sbt** :

1. Crea sotto la directory root dell'utente una cartella *.sbt* \ *0.13* \ *plugins* e un file *plugins.sbt* . Ad esempio per utenti di Windows **asch** :

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

2. Aggiungi il plugin eclipse in *plugins.sbt* :

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

3. Creare nella directory *.sbt* dell'utente un file *sbteclipse.sbt* . Ad esempio per utenti di Windows **asch** :

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

4. Inserire in *sbteclipse.sbt* un flag che *imponga la* compilazione a verificarsi quando viene eseguito il comando **activator eclipse** :

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

5. Aggiungi facoltativamente altre impostazioni di **EclipseKeys** .

Debug di eclissi

Per eseguire il debug, avviare l'applicazione con la porta predefinita 9999:

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

o con la diversa porta:

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

In eclissi:

1. Fare clic con il tasto destro del mouse sul progetto e selezionare **Debug come** , **Debug Configurations** .
2. Nella **finestra di** dialogo **Debug Configurations** , fare clic con il tasto destro del mouse su **Remote Java Application** e selezionare **New** .
3. Cambia porta su rilevante (9999 se è stata utilizzata la porta di debug predefinita) e fai clic su **Applica** .

D'ora in poi puoi fare clic su **Debug** per connettersi all'applicazione in esecuzione. L'arresto della sessione di debug non fermerà il server.

Eclipse IDE

Prerequisiti

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

Installazione di Scala in Eclipse

1. Avvia Eclipse
2. Apri `Help` > `Eclipse Marketplace`
3. Digita `Scala` in `Find`

4. Installa Scala IDE

Imposta sbteclipse

1. Apri il progetto di gioco `.\project\ plugins.sbt`
2. Aggiungi il seguente comando in `plugins.sbt` per convertire il progetto eclipse

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

3. Apri il comando e vai a riprodurre il progetto es. `cd C:\play\play-scala` . Digitare seguendo alla riga di comando

   attivatore eclissi

Importazione del progetto

1. Vai al menu `File` > `Import` in Eclipse
2. Seleziona `Existing Projects into Workspace`
3. Seleziona la directory principale

Ora il tuo progetto è pronto per la visualizzazione e la modifica su Eclipse IDE.

Leggi Impostazione del tuo IDE preferito online:
https://riptutorial.com/it/playframework/topic/4437/impostazione-del-tuo-ide-preferito

Capitolo 5: Iniezione delle dipendenze - Java

Examples

Iniezione di dipendenza con Guice - Play 2.4, 2.5

Guice è l'impostazione di dipendenza di default (ulteriore **DI**) di Play. Possono essere usati anche altri framework, ma l'uso di Guice rende più facile lo sviluppo, poiché Play si prende cura delle cose sotto il velo.

Iniezione di API Play

A partire da Play 2.5 diverse API, che erano statiche nelle versioni precedenti, dovrebbero essere create con **DI** . Questi sono, ad esempio, *Configurazione* , *JPAApi* , *CacheApi* , ecc.

Il metodo di iniezione delle API di gioco è diverso per una classe, che viene automaticamente iniettata da Play e per una classe personalizzata. L'iniezione in una classe **automaticamente iniettata** è semplice quanto mettere un'annotazione *@Inject* appropriata su un campo o un costruttore. Ad esempio, per inserire la *configurazione* in un controller con l'iniezione di proprietà:

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

o con l'iniezione del costruttore:

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

L'iniezione in una classe **personalizzata** , che è registrata per **DI** , dovrebbe essere fatta esattamente come è stata fatta per la classe iniettata automaticamente - con *l'* annotazione *@Inject* .

L'iniezione da una classe **personalizzata** , che non è vincolata per **DI** , dovrebbe essere effettuata tramite una chiamata esplicita a un iniettore con *Play.current (). Injector ()* . Ad esempio, per inserire la *configurazione* in una classe personalizzata, definire un membro dei dati di configurazione come questo:

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

Binding a iniezione personalizzata

Il binding personalizzato dell'iniezione può essere eseguito con **l'** annotazione **@ImplementedBy** o in modo programmatico con il **modulo Guice** .

Iniezione con annotazione @ImplementedBy

L'inserimento con annotazione @ImplementedBy è il modo più semplice. L'esempio seguente mostra un servizio che fornisce una facciata alla **cache** .

1. Il servizio è definito da un'interfaccia *CacheProvider* come segue:

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

2. Il servizio è implementato da una classe RunTimeCacheProvider:

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

Nota : il membro dati *appCache* viene iniettato al momento della creazione di un'istanza *RunTimeCacheProvider* .

3. L'ispettore cache è definito come membro di un controller con annotazione *@Inject* e viene chiamato dal controller:

```
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));
  }
```

L' *inserimento* con annotazione *@ImplementedBy* crea l'associazione fissa: *CacheProvider* nell'esempio precedente viene sempre istanziato con *RunTimeCacheProvider* . Tale metodo si adatta solo per un caso, quando c'è un'interfaccia con una singola implementazione. Non può essere d'aiuto un'interfaccia con diverse implementazioni o una classe implementata come un singleton senza un'interfaccia astratta. Onestamente parlando, @ImplementedBy verrà usato in rari casi, se non altro. È più probabile che utilizzi il binding programmatico con il **modulo Guice** .

Associazione dell'iniezione con un modulo di riproduzione predefinito

Il modulo Play predefinito è una classe denominata *Modulo* nella directory del progetto root definita in questo modo:

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

Nota : lo snippet sopra mostra il binding all'interno di configure, ma ovviamente qualsiasi altro metodo di binding verrà rispettato.

Per l'associazione programmatica di *CacheProvider* a *RunTimeCacheProvider* :

1. Rimuovi *l'* annotazione *@ImplementedBy* dalla definizione di *CacheProvider* :

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

2. Implementare il modulo *configurare* come segue:

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

Associazione flessibile dell'iniezione con un modulo di riproduzione predefinito

RunTimeCacheProvider non funziona bene nei test di *JUnit* con un'applicazione falsa (vedi argomento test di unità). Quindi, la diversa implementazione di *CacheProvider* è richiesta per i test unitari. Il legame di iniezione deve essere eseguito in base all'ambiente.

Vediamo un esempio.

1. La classe *FakeCache* fornisce un'implementazione stub di *CacheApi* da utilizzare durante l'esecuzione dei test (la sua implementazione non è così interessante - è solo una mappa).
2. La classe *FakeCacheProvider* implementa *CacheProvider* da utilizzare durante l'esecuzione dei test:

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

```
}
```

2. Il modulo è implementato come segue:

```
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);
    }
  }
}
```

L'esempio è buono solo per scopi educativi. Il binding per i test all'interno del modulo non è la procedura migliore, poiché si collega tra applicazione e test. Il binding per i test dovrebbe essere fatto piuttosto dai test stessi e il modulo non dovrebbe essere a conoscenza dell'esecuzione specifica del test. Vedi come farlo meglio in ....

Associazione di iniezione con un modulo personalizzato

Un modulo personalizzato è molto simile al modulo Play predefinito. La differenza è che può avere qualsiasi nome e appartenere a qualsiasi pacchetto. Ad esempio, un modulo OnStartupModule appartiene ai moduli del pacchetto.

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

Un modulo personalizzato dovrebbe essere abilitato esplicitamente per l'invocazione tramite Play. Per il modulo *OnStartupModule, è* necessario aggiungere quanto segue in *application.conf* :

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

Leggi Iniezione delle dipendenze - Java online:
https://riptutorial.com/it/playframework/topic/6060/iniezione-delle-dipendenze---java

Capitolo 6: Iniezione delle dipendenze - Scala

Sintassi

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

Examples

Utilizzo di base

Una tipica classe singleton:

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

Un'altra classe, che richiede l'accesso al primo.

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

Finalmente un controller che usa l'ultimo. Nota poiché non abbiamo contrassegnato FastFoodService come un singleton, ogni volta che viene iniettato viene creata una nuova istanza.

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

Iniezione di lezioni di gioco

Spesso è necessario accedere alle istanze di classi dal framework stesso (come WSClient o Configuration). Puoi iniettarli nelle tue classi:

```
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
}
```

Alcuni, come ExecutionContext, saranno probabilmente più facili da usare se vengono importati come impliciti. Basta aggiungerli in un secondo elenco di parametri nel costruttore:

```
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
}
```

Definire collegamenti personalizzati in un modulo

L'utilizzo di base dell'iniezione di dipendenza viene eseguito dalle annotazioni. Quando è necessario modificare un po 'le cose, è necessario un codice personalizzato per specificare ulteriormente come si desidera istanziare e iniettare alcune classi. Questo codice va in quello che viene chiamato un modulo.

```
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()
  }
```

```
}
```

Leggi Iniezione delle dipendenze - Scala online:
https://riptutorial.com/it/playframework/topic/3020/iniezione-delle-dipendenze---scala

Capitolo 7: Java - Hello World

Osservazioni

- Questo tutorial è pensato per eseguire Play in un sistema Linux / MacOS

Examples

Crea il tuo primo progetto

Per creare un nuovo progetto usa il seguente comando (`HelloWorld` è il nome del progetto e `play-java` è il modello)

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

Dovresti ottenere un risultato simile a questo

```
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
```

> Il progetto verrà creato nella directory corrente (in questo caso era la mia cartella home)

Ora siamo pronti per iniziare la nostra applicazione

Ottieni attivatore

Il primo passo nel tuo viaggio nel mondo di Play Framework è scaricare Activator. Activator è uno strumento utilizzato per creare, creare e distribuire applicazioni Play Framework.

L'attivatore può essere scaricato dalla sezione Play download (qui userò la versione 1.3.10)

Dopo aver scaricato il file, estrai il contenuto in qualche directory in cui hai accesso in scrittura e siamo pronti per partire

> In questo tutorial presumo che Activator sia stato estratto nella tua cartella home

La prima corsa

Quando abbiamo creato il nostro progetto, Activator ci ha detto come possiamo eseguire la nostra applicazione

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

C'è una piccola trappola qui: l'eseguibile di `activator` non è nella nostra root di progetto, ma in `bin/activator` . Inoltre, se hai cambiato la tua directory corrente nella directory del progetto, puoi semplicemente eseguire

```
bin/activator
```

Activator scaricherà ora le dipendenze richieste per compilare ed eseguire il progetto. A seconda della velocità di connessione, questo può richiedere del tempo. Spero che ti venga presentato un prompt

```
[HelloWorld] $
```

Ora possiamo eseguire il nostro progetto usando `~run` : questo dirà ad Activator di eseguire il nostro progetto e controllare i cambiamenti. Se qualcosa cambia, ricompila le parti necessarie e riavvia la nostra applicazione. Puoi interrompere questo processo premendo Ctrl + D (torna alla shell di Activator) o Ctrl + D (vai alla shell del tuo OS)

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

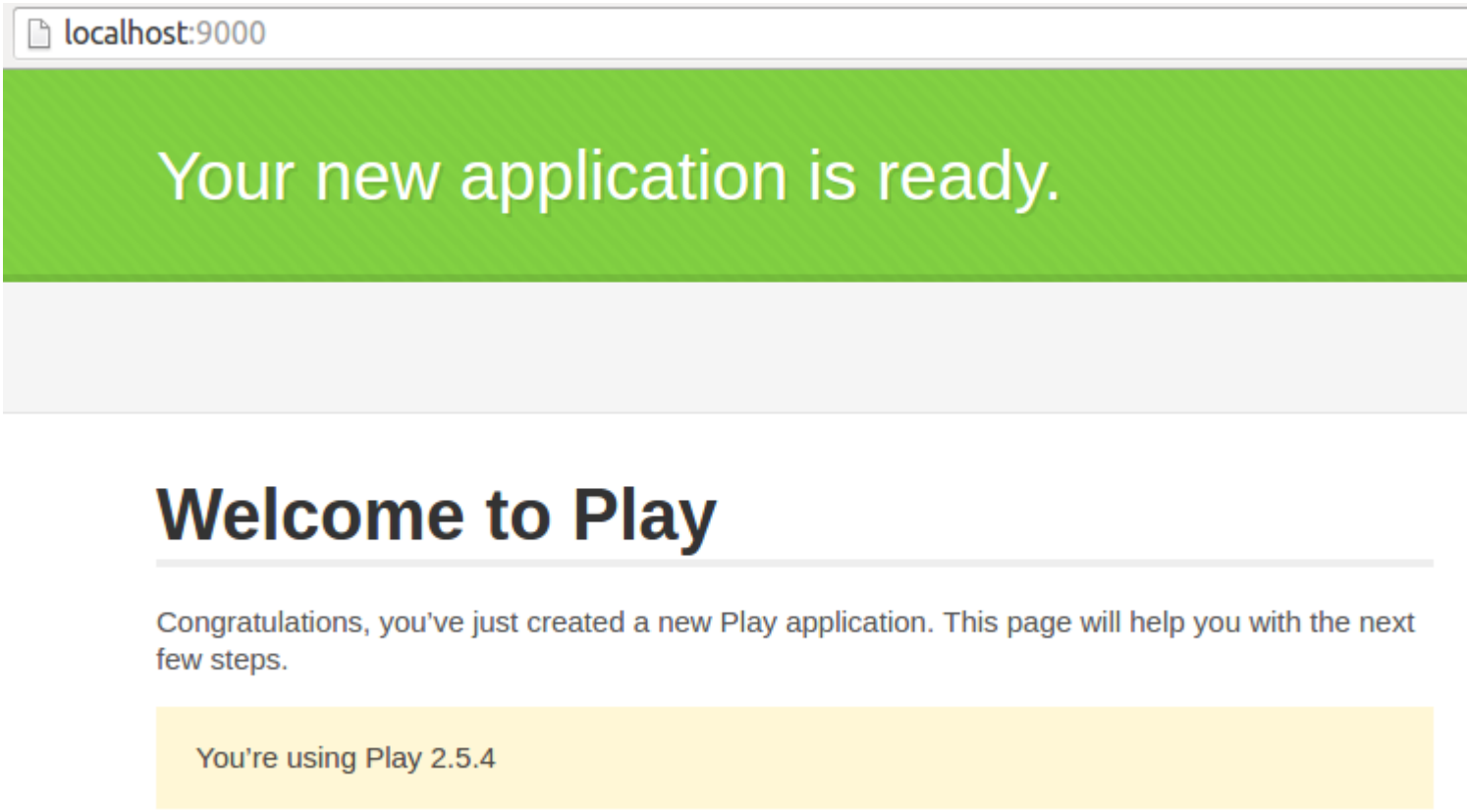
Play ora scaricherà più dipendenze. Al termine di questa procedura, la tua app dovrebbe essere pronta per l'uso:

```
-- (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...)
```

Quando navighi su localhost: 9000 nel tuo browser dovresti vedere la pagina iniziale di Play framework



Congratulazioni, ora sei pronto per apportare alcune modifiche alla tua applicazione!

"Hello World" in Hello World

Un "Hello World" non merita questo nome se non fornisce un messaggio Hello World. Quindi facciamo uno.

Nell'app file `app/controllers/HomeController.java` aggiungere il seguente metodo:

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

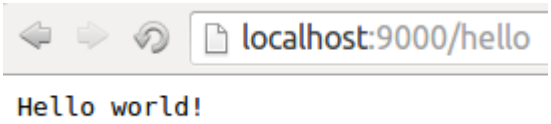
E nel tuo file `conf/routes` aggiungi quanto segue alla fine del file:

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

Se dai un'occhiata al tuo terminale, dovresti notare che Play sta compilando la tua applicazione mentre apporti le modifiche e ricaricando l'app:

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

Navigazione verso localhost: 9000 / ciao , finalmente riceviamo il nostro messaggio ciao mondo



Leggi Java - Hello World online: https://riptutorial.com/it/playframework/topic/5887/java---hello-world

Capitolo 8: Java - Lavorare con JSON

Osservazioni

Riproduci documentazione: https://www.playframework.com/documentation/2.5.x/JavaJsonActions

Examples

Creazione manuale 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;
}
```

Caricamento json da stringa / file

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

Caricamento di un file dalla tua cartella pubblica

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

Carica da una stringa

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

Trasversione di un documento JSON

Nei seguenti esempi, `json` contiene un oggetto JSON con i seguenti dati:

```
[
  {
    "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
  }
]
```

Ottieni il nome di qualche utente (non sicuro)

```
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*
```

Ottieni il nome utente (modo sicuro)

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

Ottieni il paese in cui lavora il primo utente

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

Prendi tutti i paesi

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

Trova tutti gli utenti che contengono l'attributo "attivo"

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

Conversione tra oggetti JSON e Java (base)

Per impostazione predefinita, Jackson (la libreria utilizzata da Play JSON) tenterà di associare ogni campo pubblico a un campo JSON con lo stesso nome. Se l'oggetto ha getter / setter, ne dedurrà il nome. Quindi, se hai una classe `Book` con un campo privato per memorizzare il codice ISBN e hai metodi get / set chiamati `getISBN/setISBN` , Jackson

- Crea un oggetto JSON con il campo "ISBN" durante la conversione da Java a JSON
- Utilizzare il metodo `setISBN` per definire il campo isbn nell'oggetto Java (se l'oggetto JSON ha un campo "ISBN").

Crea un oggetto Java da JSON

```
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
```

Crea un oggetto JSON dall'oggetto Java

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

Creazione di una stringa JSON da un oggetto JSON

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

JSON pretty printing

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

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

```
*/
```

Leggi Java - Lavorare con JSON online: https://riptutorial.com/it/playframework/topic/6318/java---lavorare-con-json

Capitolo 9: Lavorare con JSON - Scala

Osservazioni

Documentazione ufficiale Documentazione del pacchetto

Puoi usare il pacchetto json di gioco indipendentemente da Play includendo

`"com.typesafe.play" % "play-json_2.11" % "2.5.3"` nel tuo `build.sbt` , vedi

- https://mvnrepository.com/artifact/com.typesafe.play/play-json_2.11
- Aggiunta di Play JSON Library a sbt

Examples

Creazione manuale di un JSON

È possibile creare manualmente un albero di oggetti JSON (un `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")
  ))
))
```

Oppure con la sintassi equivalente più corta, basata su poche conversioni implicite:

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

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

Per ottenere la stringa JSON:

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

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

Java: accettazione delle richieste JSON

```
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: accettazione delle richieste JSON con BodyParser

```
@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);
    }
}
```

Suggerimento: il vantaggio in questo modo è che Play risponderà automaticamente con un codice di stato HTTP 400 se la richiesta non è valida (Content-type è stato impostato su `application/json` *ma non è stato fornito JSON)*

Scala: lettura manuale di un JSON

Se ti viene assegnata una stringa JSON:

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

Puoi analizzarlo per ottenere un JsValue, che rappresenta l'albero JSON

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

E attraversa l'albero per cercare valori specifici:

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

Metodi utili

- `\` per andare a una chiave specifica in un oggetto JSON
- `\\` per andare a tutte le occorrenze di una chiave specifica in un oggetto JSON, cercando ricorsivamente negli oggetti nidificati
- `.apply(idx)` (es. `(idx)`) per andare a un indice in una matrice
- `.as[T]` per trasmettere a un sottotipo preciso
- `.asOpt[T]` per tentare di eseguire il cast in un sottotipo preciso, restituendo None se è il tipo sbagliato
- `.validate[T]` per tentare di `.validate[T]` un valore JSON ad un sottotipo preciso, restituendo un JsSuccess o un JsError

```
(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
```

Mappatura automatica a / da classi del caso

Nel complesso, il modo più semplice per lavorare con JSON è quello di avere un mapping di case case direttamente sul JSON (stesso nome di campi, tipi equivalenti, ecc.).

```
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]
```

Conversione 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"}}
```

Conversione da 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))
```

Leggi Lavorare con JSON - Scala online:
https://riptutorial.com/it/playframework/topic/2983/lavorare-con-json---scala

Capitolo 10: Test unitario

Examples

Test unitario - Java, Play 2.4.2.5

Helpers e fakeApplication

Gli *aiutanti di* classe sono molto usati per i test unitari. Imita un'applicazione Play, finge richieste e risposte HTTP, sessione, cookie - tutto ciò che può essere necessario per i test. Un controller sotto il test dovrebbe essere eseguito nel contesto di un'applicazione Play. Il metodo *Helpers fakeApplication* fornisce un'applicazione per eseguire test. Per utilizzare *Helpers* e *fakeApplication,* una classe di test dovrebbe derivare da *WithApplication* .

Dovrebbero essere usati i seguenti API di *Helpers* :

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

Test con gli *helper* è simile a questo:

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

L'aggiunta di istruzioni di importazione per i metodi Helpers rende il codice più compatto:

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

}

Testare i controller

Chiamiamo un metodo controller, che è associato al particolare URL nelle *rotte* come metodo **indirizzato** . Un'invocazione di un metodo **indirizzato** è chiamata **azione** controller e ha una *chiamata di* tipo Java. Gioca costruisce il cosiddetto percorso inverso per ogni **azione** . Chiamare su una rotta inversa crea un oggetto *chiamata* appropriato. Questo meccanismo di routing inverso viene utilizzato per testare i controller.

Per richiamare **un'azione del** controllore da test, utilizzare la seguente API Helpers:

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

Esempio di test del controller

1. I *percorsi* :

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

2. Percorsi inversi generati:

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

3. Il metodo *getConfId* è associato a **GET** e non riceve un corpo in una richiesta. Può essere invocato per il test con:

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

4. Il metodo *addParticipant* è associato a **POST** . Si aspetta di ricevere un corpo in una richiesta. La sua invocazione in test dovrebbe essere fatta in questo modo:

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

Mocking con PowerMock

Per abilitare il mocking, una classe di test dovrebbe essere annotata come segue:

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

Derisione di un'azione del controller

Una chiamata al controller viene derisa con *RequestBuilder* :

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

Per il suddetto addParticipant viene derisa un'azione con:

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

Per richiamare il metodo del controller:

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

L'intero 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());
 });
}
```

Derisione di un'azione con il corpo JSON

Supponiamo che un input sia un oggetto di tipo *T.* La richiesta di azione mocking può essere eseguita in diversi modi.

Opzione 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;
}
```

Opzione 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;
}
```

Mocking di un'azione con l'intestazione di autenticazione di base

La richiesta di azione che deride:

```
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;
}
```

Derisione di un'azione con sessione

La richiesta di azione che deride:

```
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;
}
```

La classe Play *Session* è solo un'estensione di *HashMap <String, String>* . Può essere deriso con un codice semplice:

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

Leggi Test unitario online: https://riptutorial.com/it/playframework/topic/6192/test-unitario

Capitolo 11: Utilizzo del servizio Web con il gioco WSClient

Osservazioni

Link alla documentazione ufficiale: https://www.playframework.com/documentation/2.5.x/ScalaWS

Examples

Uso di base (Scala)

Le richieste HTTP vengono eseguite tramite la classe WSClient, che è possibile utilizzare come parametro iniettato nelle proprie classi.

```
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}")
    }
  }
}
```

Leggi Utilizzo del servizio Web con il gioco WSClient online:
https://riptutorial.com/it/playframework/topic/2981/utilizzo-del-servizio-web-con-il-gioco-wsclient

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con PlayFramework	Abhinab Kanrar, Anton, asch, Community, implicitdef, James, John, robguinness
2	chiazza di petrolio	John
3	Edilizia e imballaggio	JulienD
4	Impostazione del tuo IDE preferito	Alice, asch, implicitdef
5	Iniezione delle dipendenze - Java	asch
6	Iniezione delle dipendenze - Scala	asch, implicitdef
7	Java - Hello World	Salem
8	Java - Lavorare con JSON	Salem
9	Lavorare con JSON - Scala	Anton, asch, implicitdef, John, Salem
10	Test unitario	asch
11	Utilizzo del servizio Web con il gioco WSClient	implicitdef, John, Salem