EBook Gratis

APRENDIZAJE playframework

Free unaffiliated eBook created from
Stack Overflow contributors.

#playframework

work

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con playframework

Observaciones

Esta sección proporciona una descripción general de qué es playframework y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro del marco de juego, y vincular a los temas relacionados. Dado que la Documentación para playframework es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Examples

Instalación de Play 1

Prerrequisitos

Para ejecutar el framework Play, necesitas Java 6 o posterior. Si desea compilar Play from source, necesitará el cliente de control de fuente Git para obtener el código fuente y Ant para compilarlo.

Asegúrese de tener Java en la ruta actual (ingrese `java --version` para verificar)

Play usará el Java predeterminado o el que está disponible en la ruta **$ JAVA_HOME** si está definido.

La utilidad de línea de comandos del **juego** usa Python. Así que debería funcionar fuera de la caja en cualquier sistema UNIX (sin embargo, requiere al menos Python 2.5).

Instalación desde el paquete binario

Instrucciones genéricas

En general, las instrucciones de instalación son las siguientes.

1. Instalar java
2. Descarga el último paquete binario de Play y extrae el archivo.
3. Agregue el comando 'jugar' a la ruta de su sistema y asegúrese de que sea ejecutable.

Mac OS X

Java está incorporado o se instala automáticamente, por lo que puede omitir el primer paso.

1. Descargue el último paquete binario de Play y extráigalo en `/Applications` .
2. Edite `/etc/paths` y agregue la línea `/Applications/play-1.2.5` (por ejemplo).

Una alternativa en OS X es:

1. Instalar HomeBrew
2. Ejecutar `brew install play`

Linux

Para instalar Java, asegúrese de usar Sun-JDK o OpenJDK (y no gcj, que es el comando Java predeterminado en muchas distribuciones de Linux)

Windows

Para instalar Java, simplemente descargue e instale el último paquete JDK. No es necesario instalar Python por separado, ya que el tiempo de ejecución de Python se incluye en el marco.

Instalando a través de `sbt`

Si ya tiene instalado `sbt` me resulta más fácil crear un proyecto de Play mínimo sin `activator` . Así es cómo.

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

Cuando haya completado los pasos anteriores, edite `build.sbt` y agregue las siguientes líneas

```
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/"
```

Finalmente, cree un `project` carpeta y dentro cree un archivo `build.properties` con la referencia a la versión de Play que le gustaría usar

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

¡Eso es! Tu proyecto está listo. Puedes lanzarlo con `sbt` . Desde dentro de `sbt` tiene acceso a los mismos comandos que con el `activator` .

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

Instalaciones

Descargar e instalar:

1. Java 8: descargue la instalación correspondiente del sitio de Oracle .
2. Activador: descargue el archivo zip desde www.playframework.com/download y extraiga los archivos a la carpeta Play de destino, por ejemplo para:

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

3. sbt - descarga desde www.scala-sbt.org .

Definir variables de entorno:

1. **JAVA_HOME** , por ejemplo:

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

2. **PLAY_HOME** , por ejemplo:

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

3. **SBT_HOME** por ejemplo:

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

Agregue la ruta a los tres programas instalados a las variables de ruta:

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

Play 2.5 instalacion fija

La instalación de Play 2.5.3 (la última versión estable 2.5) viene con un problema menor. Arreglarlo:

1. Edite el archivo *activator-dist-1.3.10 \ bin \ activator.bat* y agregue el carácter "%" al final de la línea 55. La línea correcta debe ser así: *establecer SBT_HOME =% BIN_DIRECTORY%*
2. Cree un subdirectorio *conf* bajo el directorio raíz del *activador activator-dist-1.3.10* .
3. Cree en el directorio *conf* un archivo vacío llamado *sbtconfig.txt* .

Creando una nueva aplicación con CLI

Inicie el *cmd* desde el directorio, donde se debe crear una nueva aplicación. La forma más corta de crear una nueva aplicación a través de CLI es proporcionar un nombre de aplicación y una plantilla como argumentos de CLI:

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

Es posible ejecutar sólo:

```
activator new
```

En este caso, se le solicitará que seleccione la plantilla deseada y el nombre de la aplicación.

Para Play 2.4 agregue manualmente a *project / plugins.sbt* :

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

Asegúrese de reemplazar 2.4.x aquí por la versión exacta que desea usar. Play 2.5 genera esta línea automáticamente.

Asegúrese de que la versión **sbt** adecuada se menciona en project / build.properties. Debe coincidir con la versión **sbt** , instalada en su máquina. Por ejemplo, para Play2.4.x debería ser:

```
sbt.version=0.13.8
```

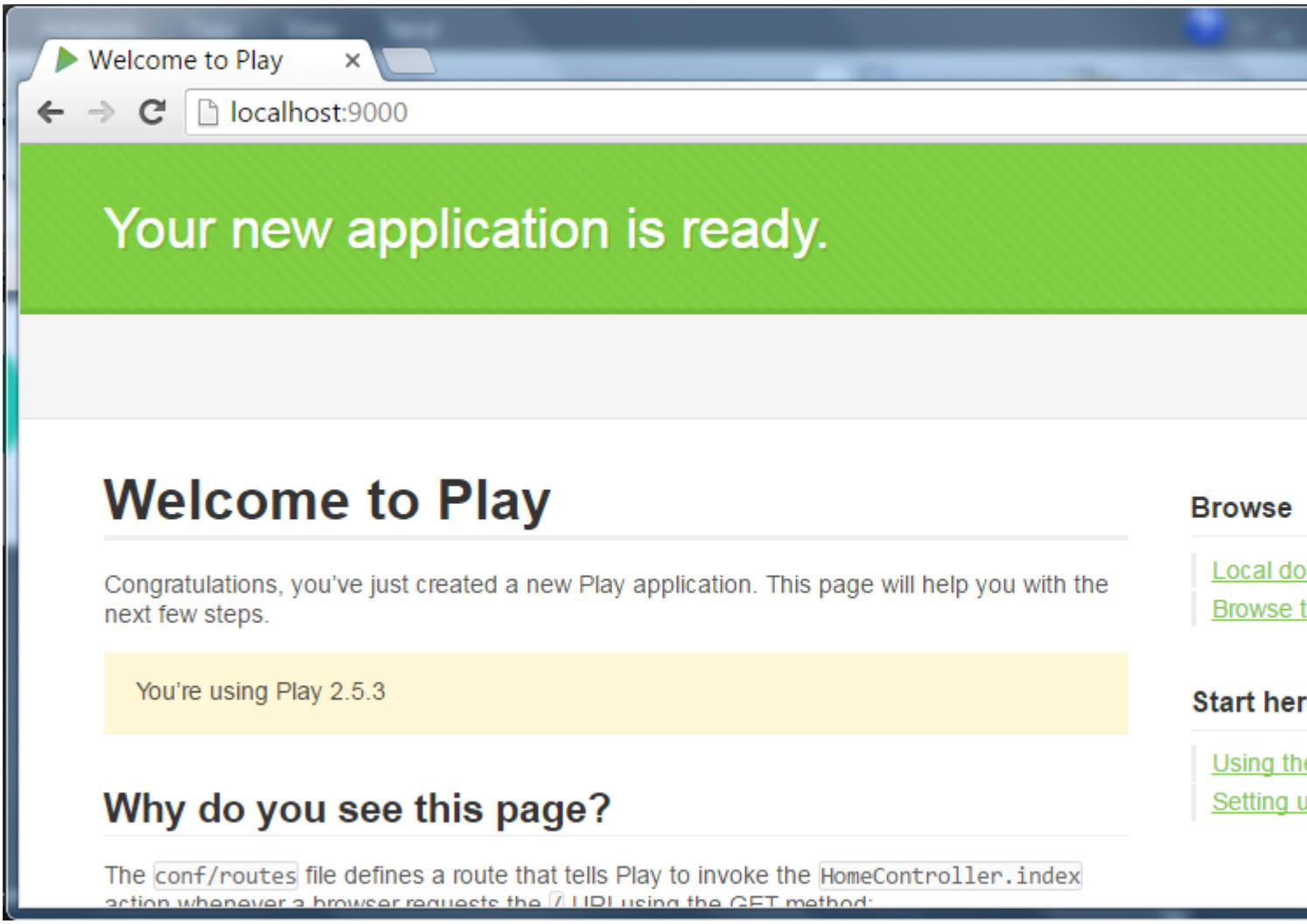
Eso es todo, una nueva aplicación ahora puede ser iniciada:

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

Después de un tiempo, el servidor se iniciará y aparecerá el siguiente mensaje en la consola:

```
[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...)
```

El servidor está escuchando por defecto en el puerto 9000. Puede solicitarlo desde un navegador a través de la URL *http: // localhost: 9000* . Obtendrá algo como esto:



Ejecutando el activador en un puerto diferente

De forma predeterminada, el activador ejecuta una aplicación en el puerto 9000 para http o 443 para https. Para ejecutar una aplicación en el puerto diferente (http):

```
activator "run 9005"
```

Lea Empezando con playframework en línea:
https://riptutorial.com/es/playframework/topic/1052/empezando-con-playframework

Capítulo 2: Configurando su IDE preferido

Examples

IntelliJ IDEA

Prerrequisitos

1. Intellij IDEA instalado (Community o Ultimate edition)
2. Scala Plugin instalado en IntelliJ
3. Un proyecto Play estándar, creado por ejemplo con Activator (`activator new [nameoftheproject] play-scala`) .

Abriendo el proyecto

1. Open IntelliJ IDEA
2. Vaya al menú `File` > `Open ...` > haga clic en toda la carpeta [nombre del proyecto]> `OK`
3. Se abre una ventana emergente con algunas opciones. Los valores predeterminados son lo suficientemente buenos en la mayoría de los casos, y si no te gustan, puedes cambiarlos en otro lugar más adelante. Haga clic en `OK`
4. Intellij IDEA pensará un poco, luego propondrá otra ventana emergente para seleccionar qué módulos seleccionar en el proyecto. Debe haber dos módulos `root` y `root-build` seleccionados de forma predeterminada. No cambie nada y haga `OK` en `OK` .
5. IntelliJ abrirá el proyecto. Puedes comenzar a ver los archivos mientras IntelliJ sigue pensando un poco como deberías ver en la barra de estado en la parte inferior, y finalmente debería estar completamente listo.

Ejecutando las aplicaciones desde Intellij

Desde allí algunas personas usan el IDE solo para ver / editar el proyecto, mientras usan la línea de comandos `sbt` para compilar / ejecutar / iniciar pruebas. Otros prefieren lanzarlos desde dentro de Intellij. Es necesario si desea utilizar el modo de depuración. Pasos

1. Menú `Run` > `Edit configurations...`
2. En la ventana emergente, haga clic en `+` en la parte superior izquierda> Elija la `Play 2 App` en la lista
3. Nombre la configuración, por ejemplo [nombre de su proyecto]. Deja las opciones predeterminadas y pulsa `OK` .
4. Desde el menú `Run` , o los botones en la interfaz de usuario, ahora puede `Run` o `Debug` con esta configuración. `Run` simplemente iniciará la aplicación, como si `sbt run` desde la línea de comandos. `Debug` hará lo mismo pero le permitirá colocar puntos de interrupción en el código

para interrumpir la ejecución y analizar lo que está sucediendo.

Opción de importación automática

Esta es una opción global para el proyecto, que está disponible en el momento de la creación y luego se puede cambiar en el menú `Intellij IDEA` > `Preferences` > `Intellij IDEA Build, Execution, Deployment` > `Build tools` > `SBT` > `Project-level settings` > `Use auto-import` .

Esta opción no tiene nada que ver con las declaraciones de `import` en el código de Scala. Indica qué debe hacer Intellij IDEA al editar el archivo `build.sbt` . Si la importación automática está activada, Intellij IDEA analizará el nuevo archivo de compilación inmediatamente y actualizará la configuración del proyecto automáticamente. Se vuelve molesto rápidamente ya que esta operación es costosa y tiende a ralentizar a Intellij cuando aún se está trabajando en el archivo de compilación. Cuando la importación automática está desactivada, debe indicar manualmente a Intellij que editó `build.sbt` y desea que la configuración del proyecto se actualice. En la mayoría de los casos, aparecerá una ventana emergente temporal que le preguntará si desea hacerlo. De lo contrario, vaya al panel SBT en la interfaz de usuario y haga clic en el signo azul de las flechas circulares para forzar la actualización.

Eclipse como Play IDE - Java, Play 2.4, 2.5

Introducción

El juego tiene varios complementos para diferentes IDE-s. El complemento **Eclipse** permite transformar una aplicación Play en un proyecto de eclipse en funcionamiento con el comando *Eclipse del activador* . El plugin de Eclipse se puede configurar por proyecto o globalmente por usuario **sbt** . Depende del trabajo en equipo, qué enfoque se debe utilizar. Si todo el equipo está utilizando el IDE de eclipse, el complemento se puede configurar a nivel de proyecto. Debe descargar la versión de eclipse compatible con Scala y Java 8: **luna** o **mars** - desde http://scala-ide.org/download/sdk.html .

Configuración de eclipse IDE por proyecto

Para importar la aplicación Play en eclipse:

1. Agregue el plugin de eclipse en *project / plugins.sbt* :

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

2. Agregue a *build.sbt* un indicador que obliga a que la compilación ocurra cuando se ejecuta el comando eclipse:

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

3. Asegúrese de que la ruta del repositorio del usuario en el archivo {root del usuario} .sbt \ repositories tenga el formato adecuado. Los valores adecuados para las propiedades *activator-launcher-local* y *activator-local* deberían tener al menos tres barras inclinadas como en el ejemplo:

```
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. Compilar la aplicación:

```
activator compile
```

5. Prepare un proyecto de eclipse para la nueva aplicación con:

```
activator eclipse
```

Ahora el proyecto está listo para ser importado en eclipse a través de **proyectos existentes en Workspace** .

Cómo adjuntar Play source a eclipse

1. Añadir a la *build.sbt* :

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

2. Compilar el proyecto

Configuración de eclipse IDE a nivel mundial

Agregue la **configuración de** usuario **sbt** :

1. Cree en el directorio raíz del usuario una carpeta *.sbt* \ *0.13* \ *plugins* y un archivo *plugins.sbt* . Por ejemplo para el usuario de Windows **ASCH** :

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

2. Agregue el plugin de eclipse en *plugins.sbt* :

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

3. Cree en el directorio *.sbt del* usuario un archivo *sbteclipse.sbt* . Por ejemplo para el usuario de Windows **ASCH** :

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

4. Coloque en *sbteclipse.sbt* un indicador que obliga a la compilación a ocurrir cuando se ejecuta el comando **eclipse** del **activador** :

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

5. Añadir opcionalmente otras configuraciones de **EclipseKeys** .

Depuración de eclipse

Para depurar, inicie la aplicación con el puerto predeterminado 9999:

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

o con el puerto diferente:

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

En eclipse:

1. Haga clic con el botón derecho en el proyecto y seleccione **Depurar como** , **Depurar configuraciones** .
2. En el cuadro de diálogo **Configuraciones de depuración** , haga clic con el botón derecho en **Aplicación Java remota** y seleccione **Nuevo** .
3. Cambie Puerto a relevante (9999 si se usó el puerto de depuración predeterminado) y haga clic en **Aplicar** .

A partir de ahora, puede hacer clic en **Depurar** para conectarse a la aplicación en ejecución. Detener la sesión de depuración no detendrá el servidor.

Eclipse IDE

Prerrequisitos

1. Java8 (1.8.0_91)
2. Eclipse neon (JavaScript y desarrollador web)
3. Play Framework 2.5.4

Instalando Scala en Eclipse

1. Lanzar el eclipse
2. Abrir `Help` > `Eclipse Marketplace`
3. Escribe `Scala` en `Find`
4. Instala Scala IDE

Configurar sbteclipse

1. Abrir proyecto de juego `.\project\ plugins.sbt`
2. Agregue el siguiente comando en `plugins.sbt` para convertir el proyecto eclipse

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

3. Abra el comando y vaya a reproducir el proyecto, por ejemplo, `cd C:\play\play-scala` . Escriba lo siguiente en la línea de comando

   eclipse de activador

Proyecto importador

1. Ir al menú `File` > `Import` en Eclipse
2. Seleccione `Existing Projects into Workspace`
3. Seleccione el directorio raíz

Ahora su proyecto está listo para ver y editar en Eclipse IDE.

Lea Configurando su IDE preferido en línea:
https://riptutorial.com/es/playframework/topic/4437/configurando-su-ide-preferido

Capítulo 3: Construcción y embalaje.

Sintaxis

- activador dist

Examples

Añadir un directorio a la distribución.

Para agregar, por ejemplo, un directorio de `scripts` al paquete de distribución:

1. Añadir al proyecto una carpeta de **scripts.**
2. En la parte superior de la build.sbt, agregue:

```
import NativePackagerHelper._
```

3. En build.sbt, agregue una asignación al nuevo directorio:

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

4. Construye el paquete de distribución con el ***activador dist*** . El archivo recién creado en `target/universal/` debe contener el nuevo directorio.

Lea Construcción y embalaje. en línea:
https://riptutorial.com/es/playframework/topic/6642/construccion-y-embalaje-

Capítulo 4: Examen de la unidad

Examples

Pruebas unitarias - Java, Play 2.4,2.5

Ayudantes y aplicación falsa

Los *Ayudantes de* clase se usan mucho para pruebas unitarias. Imita una aplicación Play, falsifica solicitudes y respuestas HTTP, sesión, cookies, todo lo que sea necesario para las pruebas. Un controlador bajo la prueba debe ejecutarse en el contexto de una aplicación Play. El método *fakeApplication de los Ayudantes* proporciona una aplicación para ejecutar pruebas. Para utilizar los *Ayudantes* y *fakeApplication,* una clase de prueba debe derivar de *WithApplication* .

Se deben usar las siguientes API de los *Ayudantes* :

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

Prueba con *Ayudantes se* ve así:

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

Agregar declaraciones de importación para los métodos de los Ayudantes hace que el código sea más compacto:

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

}

Controladores de prueba

Llamemos a un método de controlador, que está vinculado a la URL particular en las *rutas* como un método **enrutado** . Una invocación de un método **enrutado** se denomina **acción de** controlador y tiene una *llamada de* tipo Java. El juego construye la llamada ruta inversa a cada **acción** . La llamada a una ruta inversa crea un objeto de *llamada* apropiado. Este mecanismo de enrutamiento inverso se utiliza para probar los controladores.

Para invocar una **acción de** controlador desde la prueba, se debe utilizar la siguiente API de ayudantes:

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

Ejemplo de pruebas de controlador

1. Las *rutas* :

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

2. Rutas inversas generadas:

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

3. El método *getConfId* está vinculado a **GET** y no recibe un cuerpo en una solicitud. Puede ser invocado para prueba con:

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

4. El método *addParticipant* está vinculado a **POST** . Se espera recibir un cuerpo en una petición. Su invocación en prueba debe hacerse así:

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

Burlándose de PowerMock

Para habilitar el simulacro de una clase de prueba debe anotarse como sigue:

```
@RunWith(PowerMockRunner.class)
```

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

La burla de una acción del controlador.

Una llamada del controlador se burla con *RequestBuilder* :

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

Para el addParticipant anterior se simula una acción con:

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

Para invocar el método del controlador:

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

Toda la prueba:

```
@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());
 });
}
```

Burlándose de una acción con cuerpo JSON

Supongamos que una entrada es un objeto de tipo *T.* La solicitud de acción de burla se puede hacer de varias maneras.

Opción 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;
}
```

Opcion 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;
}
```

Simulación de una acción con encabezado de autenticación base

La solicitud de acción burlándose:

```
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;
}
```

Burlándose de una acción con sesión.

La solicitud de acción burlándose:

```
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 clase Play *Session* es solo una extensión del *HashMap <String, String>* . Puede ser burlado con código simple:

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

Lea Examen de la unidad en línea: https://riptutorial.com/es/playframework/topic/6192/examen-de-la-unidad

Capítulo 5: Inyección de dependencia - Java

Examples

Inyección de dependencia con Guice - Play 2.4, 2.5

Guice es el marco por defecto de la inyección de dependencia (**DI** adicional) de Play. También se pueden usar otros marcos, pero usar Guice facilita los esfuerzos de desarrollo, ya que Play se encarga de las cosas bajo el velo.

Inyección de Play API-s

A partir de Play 2.5, se deben crear varias API-s, que eran estáticas en las versiones anteriores, con **DI** . Estos son, por ejemplo, *Configuración* , *JPAApi* , *CacheApi* , etc.

El método de inyección de Play API-s es diferente para una clase, que es inyectado automáticamente por Play y para una clase personalizada. La inyección en una clase **inyectada automáticamente** es tan simple como colocar la anotación *@Inject* apropiada en el campo o en el constructor. Por ejemplo, para inyectar la *configuración* en un controlador con inyección de propiedades:

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

o con inyección de constructor:

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

La inyección en una clase **personalizada** , que está registrada para **DI** , se debe hacer como se hace para la clase inyectada automáticamente, con la anotación *@Inject* .

La inyección de una clase **personalizada** , que no está vinculada a **DI** , debe realizarse mediante una llamada explícita a un inyector con *Play.current (). Injector ()* . Por ejemplo, para inyectar la *configuración* en una clase personalizada, defina un miembro de datos de configuración como este:

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

Encuadernación de inyección personalizada

El enlace de inyección personalizado se puede realizar con la anotación **@ImplementedBy** o mediante programación con el **módulo Guice** .

Inyección con @Implementada por anotación.

La inyección con @ImplementedBy anotación es la forma más sencilla. El siguiente ejemplo muestra un servicio, que proporciona una fachada para el **caché** .

1. El servicio está definido por una interfaz *CacheProvider* como sigue:

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

2. El servicio es implementado por una clase RunTimeCacheProvider:

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

Nota : el miembro de datos de *appCache* se inyecta al crear una instancia de *RunTimeCacheProvider* .

3. El inspector de caché se define como un miembro de un controlador con la anotación @*Inject* y se llama desde el controlador:

```
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));
  }
```

La inyección con @*ImplementedBy* anotación crea el enlace fijo: *CacheProvider* en el ejemplo anterior siempre se *crea una* instancia con *RunTimeCacheProvider* . Dicho método se ajusta solo a un caso, cuando hay una interfaz con una sola implementación. No puede ayudar a una interfaz con varias implementaciones o una clase implementada como un singleton sin interfaz abstracta. Hablando honestamente, @ImplementedBy se usará en casos raros, si es que todo. Es más probable que use el enlace programático con el **módulo Guice** .

Encuadernación por inyección con un módulo Play predeterminado

El módulo Play predeterminado es una clase llamada *Módulo* en el directorio del proyecto raíz definido de esta manera:

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

Nota : el fragmento anterior muestra el enlace dentro de configure, pero, por supuesto, se respetará cualquier otro método de enlace.

Para el enlace programático de *CacheProvider* a *RunTimeCacheProvider* :

1. Quitar *@ImplementedBy* anotación de la definición de *CacheProvider* :

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

2. Implementar el módulo *configurar de la* siguiente manera:

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

Encuadernación de inyección flexible con un módulo Play predeterminado

RunTimeCacheProvider no funciona bien en las pruebas de *JUnit* con una aplicación falsa (vea el tema de pruebas de unidad). Por lo tanto, la implementación diferente de *CacheProvider* se exige para pruebas unitarias. La encuadernación por inyección debe hacerse de acuerdo con el entorno.

Veamos un ejemplo.

1. La clase *FakeCache* proporciona una implementación de *código* auxiliar de *CacheApi* para ser utilizada mientras se ejecutan las pruebas (su implementación no es tan interesante, es solo un mapa).
2. La clase *FakeCacheProvider* implementa *CacheProvider* para ser utilizado al ejecutar pruebas:

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

```
  public CacheApi getCache() {
    return fakeCache;
  }
}
```

2. El módulo se implementa de la siguiente manera:

```
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);
    }
  }
}
```

El ejemplo es bueno sólo para fines educativos. La vinculación de las pruebas dentro del módulo no es la mejor práctica, ya que se combina entre la aplicación y las pruebas. La vinculación de las pruebas debe realizarse en lugar de las pruebas en sí, y el módulo no debe conocer la implementación específica de la prueba. Vea cómo hacerlo mejor en ....

Encuadernación por inyección con un módulo personalizado.

Un módulo personalizado es muy similar al módulo de reproducción predeterminado. La diferencia es que puede tener cualquier nombre y pertenecer a cualquier paquete. Por ejemplo, un módulo OnStartupModule pertenece a los módulos del paquete.

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

Un módulo personalizado debe habilitarse explícitamente para la invocación de Play. Para el módulo *OnStartupModule, se* debe agregar lo siguiente en *application.conf* :

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

Lea Inyección de dependencia - Java en línea:
https://riptutorial.com/es/playframework/topic/6060/inyeccion-de-dependencia---java

Capítulo 6: Inyección de dependencia - Scala

Sintaxis

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

Examples

Uso básico

Una clase típica de singleton:

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

Otra clase, que requiere acceso a la primera.

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

Finalmente un controlador utilizando el último. Tenga en cuenta que, como no marcamos FastFoodService como singleton, se crea una nueva instancia cada vez que se inyecta.

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

Clases de juego de inyección

A menudo, necesitará acceder a instancias de clases desde el propio marco (como WSClient o la Configuración). Puedes inyectarlos en tus propias clases:

```
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
}
```

Algunos, como el ExecutionContext, probablemente sean más fáciles de usar si se importan como implícitos. Solo agréguelos a una segunda lista de parámetros en el constructor:

```
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
}
```

Definición de enlaces personalizados en un módulo

El uso básico de la inyección de dependencia se realiza mediante las anotaciones. Cuando necesita ajustar un poco las cosas, necesita un código personalizado para especificar cómo desea que se instalen e inyecten algunas clases. Este código va en lo que se llama un módulo.

```
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()
  }

}
```

Lea Inyección de dependencia - Scala en línea:
https://riptutorial.com/es/playframework/topic/3020/inyeccion-de-dependencia---scala

Capítulo 7: Java - Hola Mundo

Observaciones

- Este tutorial está dirigido a ejecutar Play en un sistema Linux / MacOS

Examples

Crea tu primer proyecto

Para crear un nuevo proyecto, use el siguiente comando (`HelloWorld` es el nombre del proyecto y `play-java` es la plantilla)

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

Deberías obtener una salida similar a esta

```
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
```

> El proyecto se creará en el directorio actual (en este caso era mi carpeta de inicio)

Ahora estamos listos para comenzar nuestra aplicación.

Obtener activador

El primer paso en tu viaje en el mundo de Play Framework es descargar Activator. Activator es una herramienta que se utiliza para crear, construir y distribuir aplicaciones de Play Framework.

El activador se puede descargar desde la sección de descargas de Play (aquí usaré la versión 1.3.10)

Después de descargar el archivo, extraiga el contenido a algún directorio que tenga acceso de escritura y estamos listos para comenzar

> En este tutorial, asumiré que Activator fue extraído a su carpeta de inicio

La primera carrera

Cuando creamos nuestro proyecto, Activator nos dijo cómo podemos ejecutar nuestra aplicación

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

Aquí hay una pequeña dificultad: el ejecutable del `activator` no está en la raíz de nuestro proyecto, sino en `bin/activator` . Además, si cambió su directorio actual a su directorio de proyecto, simplemente puede ejecutar

```
bin/activator
```

Activator ahora descargará las dependencias necesarias para compilar y ejecutar su proyecto. Dependiendo de la velocidad de su conexión, esto puede llevar algún tiempo. Con suerte, se le presentará con un aviso

```
[HelloWorld] $
```

Ahora podemos ejecutar nuestro proyecto usando `~run` : esto le indicará a Activator que ejecute nuestro proyecto y observe los cambios. Si algo cambia, recompilará las partes necesarias y reiniciará nuestra aplicación. Puedes detener este proceso presionando Ctrl + D (vuelve al shell Activator) o Ctrl + D (va al shell de tu SO)

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

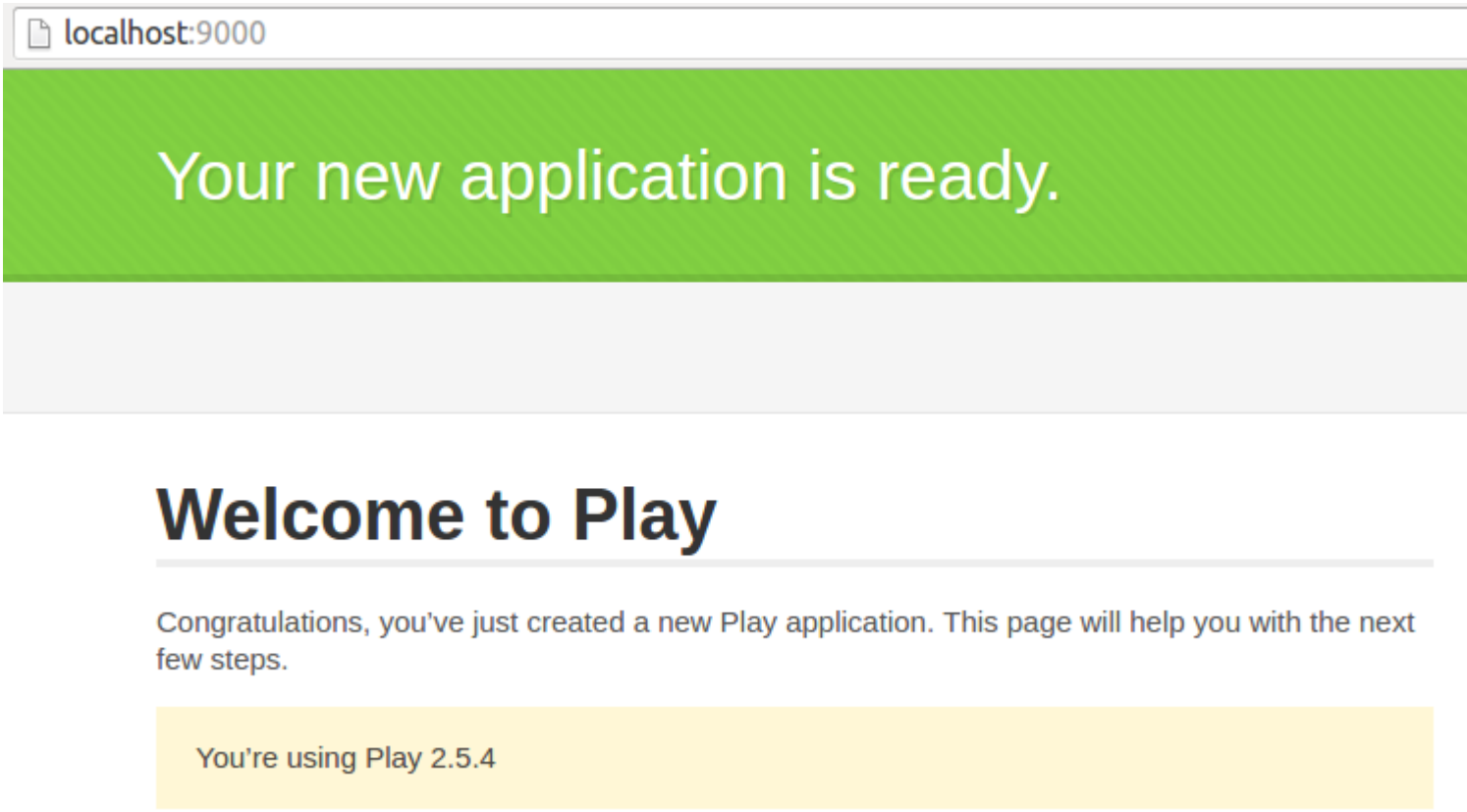
El juego ahora descargará más dependencias. Después de que se realiza este proceso, su aplicación debe estar lista para usar:

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

Cuando navegue a localhost: 9000 en su navegador, debería ver la página de inicio de Play framework



¡Felicitaciones, ahora está listo para realizar algunos cambios en su aplicación!

El "Hola Mundo" en el Hola Mundo

Un "Hello World" no merece este nombre si no proporciona un mensaje de Hello World. Así que vamos a hacer uno.

En el archivo `app/controllers/HomeController.java` agregue el siguiente método:

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

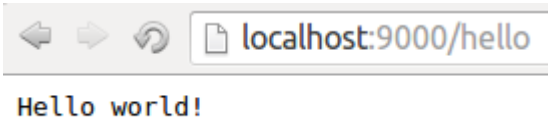
Y en su archivo `conf/routes` agregue lo siguiente al final del archivo:

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

Si echas un vistazo a tu terminal, deberías notar que Play está compilando tu aplicación mientras haces los cambios y recargas la aplicación:

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

Navegando a localhost: 9000 / hola , finalmente recibimos nuestro mensaje hola mundo



Lea Java - Hola Mundo en línea: https://riptutorial.com/es/playframework/topic/5887/java---hola-mundo

Capítulo 8: Java - Trabajando con JSON

Observaciones

Reproducir documentación:
https://www.playframework.com/documentation/2.5.x/JavaJsonActions

Examples

Manual de creación de 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;
}
```

Cargando json desde cadena / archivo

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

Cargando un archivo de su carpeta pública

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

Cargar desde una cadena

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

Transversando un documento JSON

En los siguientes ejemplos, `json` contiene un objeto JSON con los siguientes datos:

```
[
  {
    "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
  }
]
```

Obtener el nombre de algún usuario (inseguro)

```
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*
```

Obtener el nombre de usuario (forma segura)

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

Consigue el país donde trabaja el primer usuario.

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

Conseguir todos los paises

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

Encuentra a cada usuario que contiene el atributo "activo"

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

Conversión entre objetos JSON y Java (básico)

De forma predeterminada, Jackson (la biblioteca que utiliza Play JSON) intentará asignar cada campo público a un campo json con el mismo nombre. Si el objeto tiene getters / setters, inferirá el nombre de ellos. Por lo tanto, si tiene una clase de `Book` con un campo privado para almacenar el ISBN y tiene métodos get / set llamados `getISBN/setISBN` , Jackson

- Cree un objeto JSON con el campo "ISBN" al convertir de Java a JSON
- Utilice el método `setISBN` para definir el campo isbn en el objeto Java (si el objeto JSON tiene un campo "ISBN").

Crear objeto Java desde 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
```

Crear objeto JSON a partir de objeto Java

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

Creando una cadena JSON desde un objeto JSON

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

JSON impresión bonita

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

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

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

Lea Java - Trabajando con JSON en línea:
https://riptutorial.com/es/playframework/topic/6318/java---trabajando-con-json

Capítulo 9: Mancha

Examples

Código de inicio de Slick

En `build.sbt` , asegúrese de incluir (aquí para Mysql y 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"
```

En su `application.conf` , agregue:

```
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"
```

Para tener una arquitectura independiente de RDBMS cree un objeto como el siguiente

```
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
  }
}
```

al crear un nuevo modelo nuevo:

```
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}
}
```

Salida DDL

El punto central del uso de Slick es escribir el menor código SQL posible. Una vez que haya escrito su definición de tabla, querrá crear la tabla en su base de datos.

Si tiene `val table = TableQuery[MyModel]` , puede obtener la definición de la tabla (código SQL - DDL) ejecutando el siguiente comando:

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

Lea Mancha en línea: https://riptutorial.com/es/playframework/topic/4604/mancha

Capítulo 10: Trabajando con JSON - Scala

Observaciones

Documentación oficial Documentación del paquete.

Puede usar el paquete play json independientemente de Play al incluir

`"com.typesafe.play" % "play-json_2.11" % "2.5.3"` en su `build.sbt` , vea

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

Examples

Creando un JSON manualmente

Puede construir un árbol de objetos JSON (un `JsValue`) manualmente

```
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")
  ))
))
```

O con la sintaxis equivalente más corta, basada en unas pocas conversiones implícitas:

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

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

Para obtener la cadena 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: aceptando peticiones 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: aceptar solicitudes 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);
    }
}
```

Sugerencia: la ventaja de esta manera es que Play responderá automáticamente con un código de estado HTTP 400 si la solicitud no era válida (el tipo de contenido se configuró como `application/json` *pero no se proporcionó JSON)*

Scala: leyendo un JSON manualmente

Si le dan una cadena JSON:

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

Puede analizarlo para obtener un JsValue, que representa el árbol JSON

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

Y atraviesa el árbol para buscar valores específicos:

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

Metodos utiles

- `\` para ir a una clave específica en un objeto JSON
- `\\` para ir a todas las apariciones de una clave específica en un objeto JSON, buscando recursivamente en objetos anidados
- `.apply(idx)` (es decir, `(idx)`) para ir a un índice en una matriz
- `.as[T]` para lanzar a un subtipo preciso
- `.asOpt[T]` para intentar convertir a un subtipo preciso, devolviendo Ninguno si es del tipo incorrecto
- `.validate[T]` para intentar convertir un valor JSON a un subtipo preciso, devolviendo 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
```

Mapeo automático a / desde clases de casos

En general, la forma más fácil de trabajar con JSON es tener una asignación de clase de caso directamente a JSON (mismo nombre de campos, tipos equivalentes, etc.).

```
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]
```

Convertir a 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"}}
```

Convertir desde 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))
```

Lea Trabajando con JSON - Scala en línea:
https://riptutorial.com/es/playframework/topic/2983/trabajando-con-json---scala

Capítulo 11: Uso del servicio web con el juego WSClient

Observaciones

Enlace a la documentación oficial: https://www.playframework.com/documentation/2.5.x/ScalaWS

Examples

Uso básico (Scala)

Las solicitudes HTTP se realizan a través de la clase WSClient, que puede usar como un parámetro inyectado en sus propias clases.

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

Lea Uso del servicio web con el juego WSClient en línea:
https://riptutorial.com/es/playframework/topic/2981/uso-del-servicio-web-con-el-juego-wsclient

Creditos

S. No	Capítulos	Contributors
1	Empezando con playframework	Abhinab Kanrar, Anton, asch, Community, implicitdef, James, John, robguinness
2	Configurando su IDE preferido	Alice, asch, implicitdef
3	Construcción y embalaje.	JulienD
4	Examen de la unidad	asch
5	Inyección de dependencia - Java	asch
6	Inyección de dependencia - Scala	asch, implicitdef
7	Java - Hola Mundo	Salem
8	Java - Trabajando con JSON	Salem
9	Mancha	John
10	Trabajando con JSON - Scala	Anton, asch, implicitdef, John, Salem
11	Uso del servicio web con el juego WSClient	implicitdef, John, Salem