Бесплатная электронная книга

УЧУСЬ playframework

Free unaffiliated eBook created from
Stack Overflow contributors.

#playframework

work

Около

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

глава 1: Начало работы с playframework

замечания

В этом разделе представлен обзор того, что такое playframework, и почему разработчик может захотеть его использовать.

Он также должен упомянуть о любых крупных предметах в рамках игрового процесса и ссылки на связанные темы. Поскольку документация для playframework является новой, вам может потребоваться создать начальные версии этих связанных тем.

Examples

Play 1 Установка

Предпосылки

Чтобы запустить платформу Play, вам нужна Java 6 или более поздняя. Если вы хотите создать Play из источника, вам понадобится клиент управления Git-источником для извлечения исходного кода и Ant для его создания.

Убедитесь, что Java находится в текущем пути (введите `java -version` для проверки)

Play будет использовать Java по умолчанию или тот, который доступен на пути **$ JAVA_HOME,** если он определен.

Утилита командной строки **воспроизведения** использует Python. Поэтому он должен работать из коробки в любой системе UNIX (однако для этого требуется хотя бы Python 2.5).

Установка из двоичного пакета

Общие инструкции

Как правило, инструкции по установке выглядят следующим образом.

1. Установите Java.
2. Загрузите последний бинарный пакет Play и извлеките архив.
3. Добавьте команду «играть» на свой системный путь и убедитесь, что она выполнена.

Mac OS X

Java встроена или установлена автоматически, поэтому вы можете пропустить первый шаг.

1. Загрузите последний бинарный пакет Play и извлеките его в `/Applications` .
2. Измените `/etc/paths` и добавьте строку `/Applications/play-1.2.5` (например).

Альтернативой для OS X является:

1. Установить HomeBrew
2. Запустите `brew install play`

Linux

Чтобы установить Java, обязательно используйте Sun-JDK или OpenJDK (а не gcj, которая является стандартной Java-командой на многих дистрибутивах Linux)

Windows

Чтобы установить Java, просто загрузите и установите последний пакет JDK. Вам не нужно устанавливать Python отдельно, поскольку среда исполнения Python связана с каркасом.

Установка через `sbt`

Если у вас уже установлен `sbt` мне легче создать минимальный проект Play без `activator` . Вот как.

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

Когда предыдущие шаги будут завершены, отредактируйте `build.sbt` и добавьте следующие строки

```
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/"
```

И, наконец, создать папку `project` и внутри создать файл `build.properties` со ссылкой на версию игры вы хотели бы использовать

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

Это оно! Ваш проект готов. Вы можете запустить его с помощью `sbt` . `sbt` вас есть доступ к тем же командам, что и с `activator` .

Начало работы с Play 2.4.x / 2.5.x - Windows, Java

сооружения

Загрузить и установить:

1. Java 8 - загрузить соответствующую установку с сайта Oracle .
2. Activator - загрузите zip с сайта www.playframework.com/download и извлеките файлы в целевую папку Play, например, чтобы:

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

3. sbt - скачать с сайта www.scala-sbt.org .

Определение переменных среды:

1. Например, **JAVA_HOME** :

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

2. **PLAY_HOME** , например:

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

3. **SBT_HOME,** например:

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

Добавьте путь ко всем трем установленным программам в переменные пути:

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

Исправить установку 2.5

Установка Play 2.5.3 (последняя версия с 2.5 стабильными версиями) имеет незначительную проблему. Починить это:

1. Отредактируйте файл- *активатор-dist-1.3.10* \ *bin* \ *activator.bat* и добавьте символ «%» в конце строки 55. Правильная строка должна быть такой: *set SBT_HOME =% BIN_DIRECTORY%*
2. Создайте подкаталог *conf* в корневой директории *активатора-dist-1.3.10 активатора* .
3. Создайте в каталоге *conf* пустой файл с именем *sbtconfig.txt* .

Создание нового приложения с помощью CLI

Запустите *cmd* из каталога, где должно быть создано новое приложение. Самый короткий способ создания нового приложения через CLI - предоставить имя приложения и шаблон в качестве аргументов CLI:

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

Можно запустить только:

```
activator new
```

В этом случае вам будет предложено выбрать нужный шаблон и имя приложения.

Для Play 2.4 добавьте вручную в *project / plugins.sbt* :

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

Обязательно замените 2.4.x здесь точной версией, которую вы хотите использовать. Play 2.5 автоматически генерирует эту строку.

Убедитесь, что соответствующая версия **sbt** упоминается в файле project / build.properties.

Он должен соответствовать версии **sbt** , установленной на вашем компьютере. Например, для Play2.4.x это должно быть:

```
sbt.version=0.13.8
```

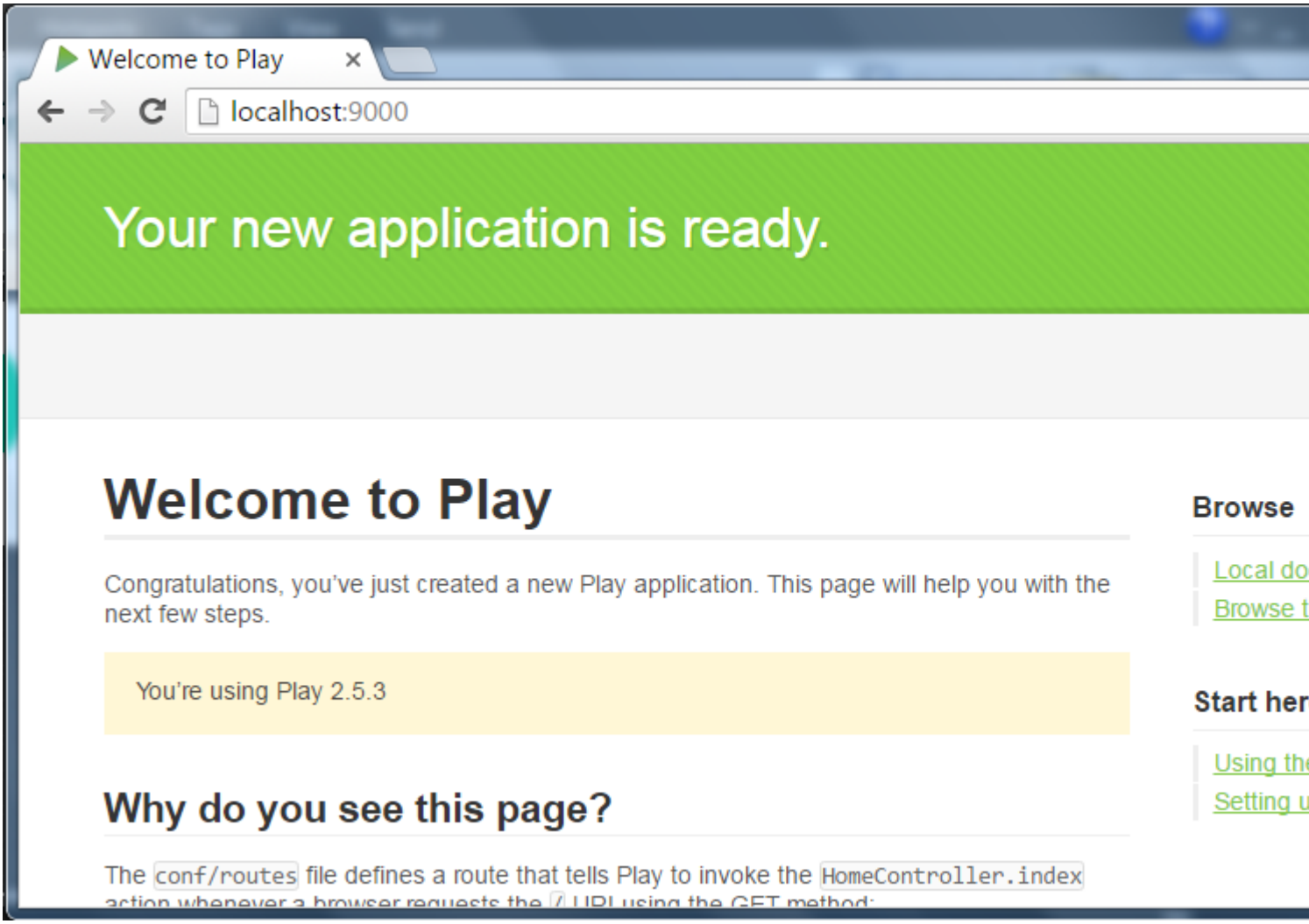
Вот и все, теперь может быть запущено новое приложение:

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

Через некоторое время сервер запустится, и на консоли появится следующее приглашение:

```
[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...)
```

Сервер по умолчанию прослушивает порт 9000. Вы можете запросить его у браузера по URL *http: // localhost: 9000* . Вы получите что-то вроде этого:



Запуск активатора на другом порту

По умолчанию активатор запускает приложение на порту 9000 для http или 443 для https. Чтобы запустить приложение на другом порту (http):

```
activator "run 9005"
```

Прочитайте Начало работы с playframework онлайн:
https://riptutorial.com/ru/playframework/topic/1052/начало-работы-с-playframework

глава 2: Java - Hello World

замечания

- Это руководство предназначено для запуска Play в системе Linux / MacOS

Examples

Создайте свой первый проект

Для создания нового проекта используйте следующую команду (`HelloWorld` - это имя проекта, а `play-java` - это шаблон)

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

Вы должны получить результат, похожий на этот

```
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
```

> Проект будет создан в текущем каталоге (в данном случае это была моя домашняя папка)

Теперь мы готовы начать наше приложение

Получить активатор

Первым шагом в вашем путешествии в мире Play Framework является загрузка Activator. Активатор - это инструмент, используемый для создания, сборки и распространения приложений Play Framework.

Активатор можно загрузить из раздела «Загрузка файлов» (здесь я буду использовать версию 1.3.10)

После того, как вы загрузили файл, извлеките содержимое в какой-либо каталог, на который у вас есть доступ на запись, и мы готовы пойти

В этом туториале я предполагаю, что активатор был извлечен в вашу домашнюю папку

Первый запуск

Когда мы создали наш проект, Activator рассказал нам, как мы можем запускать наше приложение

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

Здесь есть небольшая ловушка: исполняемый файл `activator` отсутствует в нашем проекте, но в `bin/activator` . Кроме того, если вы изменили свой текущий каталог на каталог проекта, вы можете просто запустить

```
bin/activator
```

Активатор теперь загрузит необходимые зависимости для компиляции и запуска вашего проекта. В зависимости от скорости соединения это может занять некоторое время. Надеемся, вам будет предложено приглашение

```
[HelloWorld] $
```

Теперь мы можем запустить наш проект, используя `~run` : это позволит Activator запустить наш проект и следить за изменениями. Если что-то изменится, оно перекомпилирует необходимые детали и перезапустит наше приложение. Вы можете остановить этот процесс, нажав Ctrl + D (возвращается в оболочку активатора) или Ctrl + D (идет в оболочку вашей ОС)

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

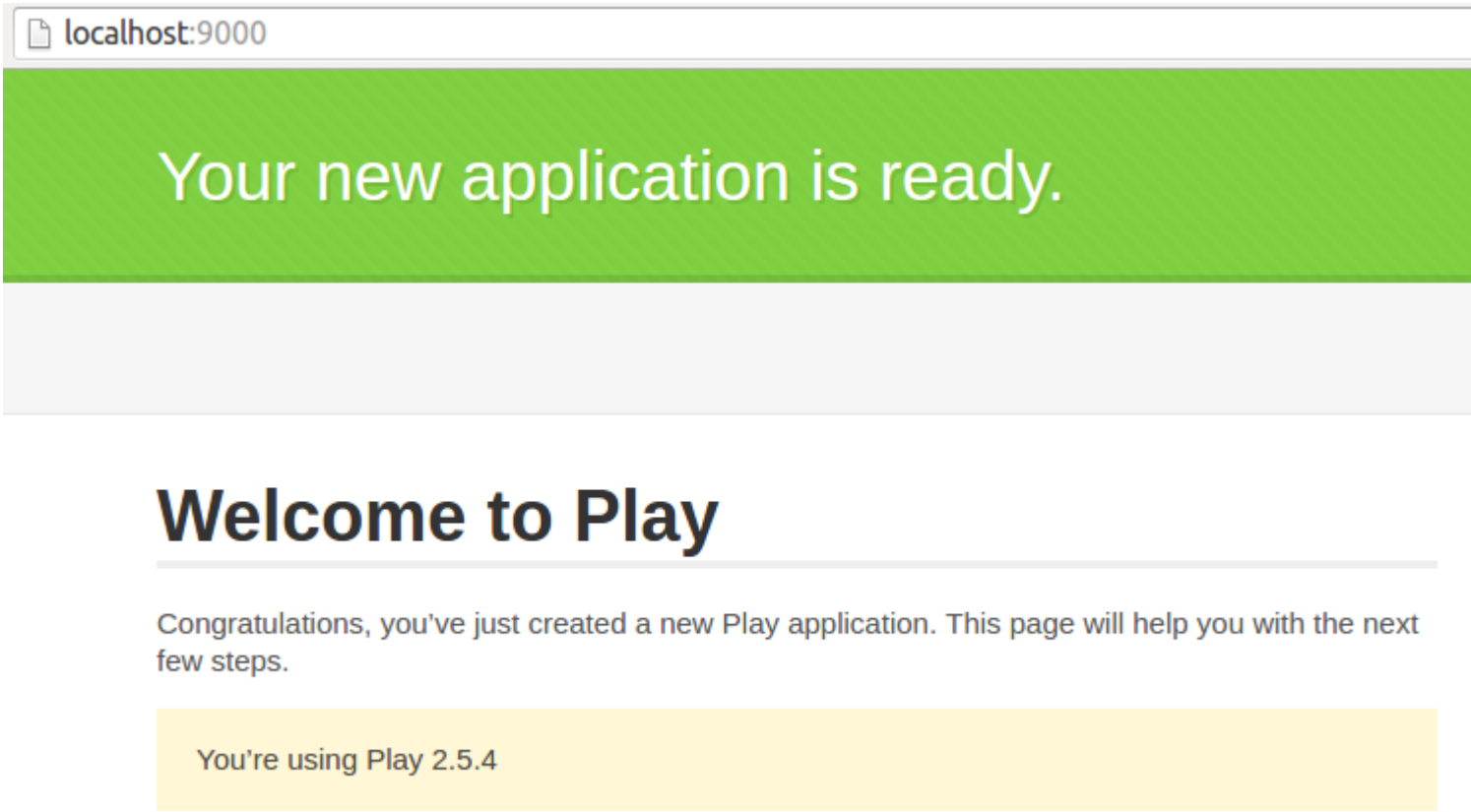
Теперь Play загрузит больше зависимостей. После того, как этот процесс будет завершен, ваше приложение должно быть готово к использованию:

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

Когда вы переходите на localhost: 9000 в своем браузере, вы должны увидеть начальную страницу платформы воспроизведения



Поздравляем, теперь вы готовы внести некоторые изменения в свое приложение!

«Привет мир» в Hello World

«Hello World» не заслуживает этого имени, если он не предоставляет сообщение Hello World. Итак, давайте сделаем это.

В файле `app/controllers/HomeController.java` добавьте следующий метод:

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

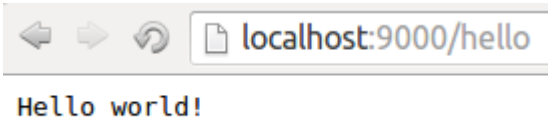
И в вашем файле `conf/routes` добавьте следующее в конце файла:

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

Если вы посмотрите на свой терминал, вы должны заметить, что Play компилирует ваше приложение во время внесения изменений и перезагрузки приложения:

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

Переход на localhost: 9000 / hello , мы наконец получаем наше приветственное мировое сообщение



Прочитайте Java - Hello World онлайн: https://riptutorial.com/ru/playframework/topic/5887/java---hello-world

глава 3: Java - работа с JSON

замечания

Документация по воспроизведению:
https://www.playframework.com/documentation/2.5.x/JavaJsonActions

Examples

Ручное создание 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;
}
```

Загрузка json из строки / файла

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

Загрузка файла из общей папки

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

Загрузка из строки

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

Пересечение документа JSON

В следующих примерах `json` содержит объект JSON со следующими данными:

```
[
  {
    "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
  }
]
```

Получить имя какого-либо пользователя (небезопасного)

```
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*
```

Получить имя пользователя (безопасный способ)

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

Получить страну, где работает первый пользователь

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

Получите все страны

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

Найти каждого пользователя, который содержит атрибут "active"

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

Преобразование между объектами JSON и Java (базовые)

По умолчанию Джексон (библиотека Play JSON использует) попытается сопоставить каждое публичное поле с полем json с тем же именем. Если у объекта есть геттеры / сеттеры, он выведет имя из них. Итак, если у вас есть класс `Book` с частным полем для хранения ISBN и есть методы get / set с именем `getISBN/setISBN` , Джексон будет

- Создайте объект JSON с полем «ISBN» при переходе с Java на JSON
- Используйте метод `setISBN` для определения поля isbn в объекте Java (если объект JSON имеет поле «ISBN»).

Создание объекта Java из 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
```

Создание объекта JSON из объекта Java

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

Создание строки JSON из объекта JSON

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

JSON довольно печатная

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

```
*/

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

Прочитайте Java - работа с JSON онлайн:
https://riptutorial.com/ru/playframework/topic/6318/java---работа-с-json

глава 4: Инъекция зависимостей - Java

Examples

Инъекционная инъекция с помощью Guice - Play 2.4, 2.5

Guice - это стандартная инъекционная система зависимостей (далее **DI**) для Play. Могут использоваться и другие рамки, но использование Guice облегчает процесс разработки, поскольку Play заботится о вещах под вуалью.

Впрыскивание API-интерфейсов Play

Начиная с версии 2.5 несколько API-интерфейсов, которые были статичными в более ранних версиях, должны быть созданы с помощью **DI** . Это, например, *конфигурация* , *JPAApi* , *CacheApi* и т. Д.

Метод инжекции API-интерфейса Play отличается для класса, который автоматически вводится Play и для пользовательского класса. Инъекция в **автоматически вводимом** классе так же просто, как включение соответствующей аннотации *@Inject* в любое поле или конструктор. Например, чтобы ввести *конфигурацию* в контроллер с введением свойств:

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

или с инжектором конструктора:

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

Инъекция в **пользовательский** класс, который зарегистрирован для **DI** , должен выполняться точно так же, как это делается для автоматически вводимого класса - с аннотацией *@Inject* .

Инъекция из **пользовательского** класса, который не связан с **DI** , должен выполняться явным вызовом инжектора с помощью функции *Play.current (). Injector ()* . Например, чтобы ввести *конфигурацию* в пользовательский класс, определите член данных конфигурации следующим образом:

```
private Configuration configuration =
```

```
Play.current().injector().instanceOf(Configuration.class);
```

Индивидуальная вставка

Пользовательское **вложение** может быть выполнено с **помощью** аннотации **@ImplementedBy** или программным способом с помощью **модуля Guice** .

Инъекция с помощью @ImplementedBy аннотации

Инъекция с аннотацией @ImplementedBy является самым простым способом. В приведенном ниже примере показана служба, которая предоставляет фасад в **кеш** .

1. Служба определяется интерфейсом *CacheProvider* следующим образом:

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

2. Служба реализована классом RunTimeCacheProvider:

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

Примечание : *член* данных *appCache* вводится при создании экземпляра *RunTimeCacheProvider* .

3. Инспектор кэша определяется как член контроллера с аннотацией *@Inject* и вызывается из контроллера:

```
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));
  }
```

Инъекция с аннотацией *@ImplementedBy* создает фиксированное связывание: *CacheProvider* в приведенном выше примере всегда *создается* с помощью *RunTimeCacheProvider* . Такой метод подходит только для случая, когда есть интерфейс с

одной реализацией. Это не может помочь для интерфейса с несколькими реализациями или класса, реализованного как одноэлементный без абстрактного интерфейса. Честно говоря, @ImplementedBy будет использоваться в редких случаях, если все это. Скорее всего, он будет использовать программную привязку к **модулю Guice** .

Инъекционное связывание с модулем воспроизведения по умолчанию

Модуль Play по умолчанию - это класс с именем *Module* в корневой директории проекта, определенный следующим образом:

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

Примечание . В приведенном выше фрагменте показан привязку внутри configure, но, конечно, любой другой метод привязки будет соблюден.

Для программной привязки *CacheProvider* к *RunTimeCacheProvider* :

1. Удалите *@ImplementedBy* аннотацию из определения *CacheProvider* :

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

2. Реализация модуля *настройки* следующим образом :

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

Гибкое вложение с модулем Play

RunTimeCacheProvider не работает в тестах *JUnit* с поддельным приложением (см. *Раздел «Тестирование модулей»*). Таким образом, различная реализация *CacheProvider* требуется для модульных тестов. Инъекционное связывание должно выполняться в соответствии с окружающей средой.

Давайте посмотрим пример.

1. Класс *FakeCache* обеспечивает реализацию заглушки *CacheApi,* которая будет использоваться при выполнении тестов (ее реализация не такая интересная - это всего лишь карта).
2. Класс *FakeCacheProvider* реализует *CacheProvider,* который будет использоваться при выполнении тестов:

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

2. Модуль реализован следующим образом:

```
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);
    }
  }
}
```

Пример хорош только для образовательных целей. Связывание для тестов внутри модуля не является лучшей практикой, так как это соединяет между приложением и тестами. Связывание для тестов должно выполняться скорее самими тестами, а модуль не должен знать о конкретной реализации теста. Посмотрите, как это сделать лучше ...

Инъекционное связывание с настраиваемым модулем

Пользовательский модуль очень похож на модуль Play по умолчанию. Разница в том, что он может иметь любое имя и принадлежать любому пакету. Например, модуль OnStartupModule принадлежит модулю пакета.

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

Пользовательский модуль должен быть явно включен для вызова в Play. Для модуля *OnStartupModule* Далее следует добавить в *application.conf:*

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

Прочитайте Инъекция зависимостей - Java онлайн:
https://riptutorial.com/ru/playframework/topic/6060/инъекция-зависимостей---java

глава 5: Инъекция зависимостей - Scala

Синтаксис

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

Examples

Основное использование

Типичный одноэлементный класс:

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

Другой класс, требующий доступа к первому.

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

Наконец, контроллер использует последний. Обратите внимание, поскольку мы не отмечали FastFoodService как singleton, новый экземпляр его создается каждый раз, когда он вводится.

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

Занятия по введению в игру

Вам часто придется обращаться к экземплярам классов из самой структуры (например, WSClient или Configuration). Вы можете вводить их в свои классы:

```
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
}
```

Некоторые, такие как ExecutionContext, скорее всего будут более удобными в использовании, если они импортируются как неявные. Просто добавьте их во второй список параметров в конструкторе:

```
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
}
```

Определение пользовательских привязок в модуле

Аннотации - основное использование инъекции зависимостей. Когда вам нужно немного изменить настройки, вам нужен специальный код, чтобы дополнительно указать, как вы хотите, чтобы некоторые классы были созданы и инъецированы. Этот код используется в том, что называется модулем.

```
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()
  }

}
```

Прочитайте Инъекция зависимостей - Scala онлайн:
https://riptutorial.com/ru/playframework/topic/3020/инъекция-зависимостей---scala

глава 6: Использование Webservice с игрой WSClient

замечания

Ссылка на официальную документацию:
https://www.playframework.com/documentation/2.5.x/ScalaWS

Examples

Основное использование (Scala)

Запросы HTTP выполняются через класс WSClient, который вы можете использовать в качестве введенного параметра в свои собственные классы.

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

Прочитайте Использование Webservice с игрой WSClient онлайн:
https://riptutorial.com/ru/playframework/topic/2981/использование-webservice-с-игрой-wsclient

глава 7: Настройка предпочтительной среды IDE

Examples

IntelliJ IDEA

Предпосылки

1. Intellij IDEA (версия сообщества или Ultimate)
2. Плагин Scala установлен в IntelliJ
3. Стандартный проект Play, созданный, например, с Activator (`activator new [nameoftheproject] play-scala`) .

Открытие проекта

1. Открыть IntelliJ IDEA
2. Перейдите в меню `File` > `Open ...` > щелкните по всей папке [nameoftheproject]> `OK`
3. Всплывающее окно открывается несколькими вариантами. Значения по умолчанию достаточно хороши в большинстве случаев, и если вам они не нравятся, вы можете изменить их в другом месте позже. Нажмите `OK`
4. Intellij IDEA немного подумает, а затем предложит другое всплывающее окно, чтобы выбрать, какие модули выбрать в проекте. По умолчанию должны быть два модуля `root` и `root-build` . Не меняйте ничего и нажмите « `OK` .
5. IntelliJ откроет проект. Вы можете начать просмотр файлов, в то время как IntelliJ продолжает думать немного, как вы должны видеть в строке состояния внизу, тогда он должен быть полностью готов.

Запуск приложений от Intellij

Оттуда некоторые люди используют IDE только для просмотра / редактирования проекта, используя командную строку `sbt` для компиляции / запуска / запуска тестов. Другие предпочитают запускать их из Intellij. Это необходимо, если вы хотите использовать режим отладки. Шаги:

1. Меню `Run` > `Edit configurations...`
2. В всплывающем окне нажмите `+` в левом верхнем углу> Выбрать `Play 2 App` в списке
3. Назовите конфигурацию, например [nameofyourproject]. Оставьте настройки по

умолчанию и нажмите `OK` .

4. Из меню « `Run` или кнопок в пользовательском интерфейсе теперь вы можете `Run` или `Debug` эту конфигурацию. `Run` просто запустит приложение, как если бы вы выполнили `sbt run` из командной строки. `Debug` будет делать то же самое, но позволяет разместить точки останова в коде, чтобы прервать выполнение и проанализировать, что происходит.

Опция автоматического импорта

Это вариант, глобальный для проекта, который доступен во время создания и впоследствии может быть изменен в меню `Intellij IDEA` > `Preferences` > `Build, Execution, Deployment` > `Build tools` > `SBT` > `Project-level settings` > `Use auto-import` .

Этот параметр не имеет ничего общего с `import` в коде Scala. Это диктует, что должен делать Intellij IDEA при редактировании файла `build.sbt` . Если активирован автоматический импорт, Intellij IDEA будет автоматически анализировать новый файл сборки и автоматически обновлять конфигурацию проекта. Это быстро раздражает, так как эта операция стоит дорого и имеет тенденцию замедлять Intellij, когда вы все еще работаете над файлом сборки. Когда автоматическое импортирование деактивируется, вы должны указать вручную Intellij, что вы отредактировали `build.sbt` и `build.sbt` , чтобы конфигурация проекта была обновлена. В большинстве случаев появится временное всплывающее окно с вопросом, хотите ли вы это сделать. В противном случае перейдите на панель SBT в пользовательском интерфейсе и нажмите значок синего кругового стрелка, чтобы принудительно обновить.

Eclipse as Play IDE - Java, Play 2.4, 2.5

Вступление

В Play есть несколько плагинов для разных IDE-файлов. Плагин **eclipse** позволяет преобразовать приложение Play в рабочий проект eclipse с помощью команды *eclipse* . Eclipse , плагин может быть установлен для каждого проекта или глобально для каждого пользователя **SBT.** Это зависит от командной работы, какой подход следует использовать. Если вся команда использует eclipse IDE, плагин может быть установлен на уровне проекта. Вам нужно загрузить версию eclipse, поддерживающую Scala и Java 8: **luna** или **mars** - из http://scala-ide.org/download/sdk.html .

Настройка Eclipse IDE для каждого проекта

Чтобы импортировать приложение Play в eclipse:

1. Добавьте плагин eclipse в *project / plugins.sbt* :

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

2. Добавьте в *build.sbt* флаг, который заставляет компиляцию произойти, когда запускается команда eclipse:

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

3. Убедитесь, что путь репозитория пользователя в файле {user root} .sbt \ repositories имеет правильный формат. Правильные значения для *активатора-запуска-локального* и *локатора-активатора* должны иметь как минимум три слэша, например, в примере:

```
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. Скомпилируйте приложение:

```
activator compile
```

5. Подготовьте проект eclipse для нового приложения с помощью:

```
activator eclipse
```

Теперь проект готов импортироваться в eclipse через **существующие проекты в рабочее пространство** .

Как подключить источник воспроизведения для затмения

1. Добавьте в *build.sbt* :

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

2. Скомпилировать проект

Настройка eclipse IDE глобально

Добавьте **настройку** пользователя **sbt** :

1. Создайте в корневом каталоге пользователя папку *.sbt \ 0.13 \ plugins* и файл *plugins.sbt* . Например, для пользователя Windows **asch** :

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

2. Добавьте плагин eclipse в *plugins.sbt* :

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

3. Создайте в *user.sbt* каталог файл *sbteclipse.sbt* . Например, для пользователя Windows **asch** :

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

4. Поместите в *sbteclipse.sbt* флаг, который заставляет компиляцию произойти, когда запускается команда **затмения активатора** :

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

5. Добавьте необязательные другие настройки **EclipseKeys** .

Отладка от eclipse

Чтобы отладить, запустите приложение с портом по умолчанию 9999:

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

или с другим портом:

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

В затмении:

1. Щелкните правой кнопкой мыши проект и выберите **Debug As** , **Debug Configurations** .
2. В диалоговом окне « **Конфигурации отладки»** щелкните правой кнопкой мыши « **Удаленное приложение Java»** и выберите « **Создать»** .

3. Измените порт на соответствующий (9999, если использовался порт отладки по умолчанию) и нажмите « **Применить**» .

С этого момента вы можете нажать « **Отладка**» для подключения к запущенному приложению. Остановка сеанса отладки не остановит сервер.

Eclipse IDE

Предпосылки

1. Java8 (1.8.0_91)
2. Eclipse neon (JavaScript и веб-разработчик)
3. Play Framework 2.5.4

Установка Scala в Eclipse

1. Запустить Eclipse
2. Открыть `Help` > `Eclipse Marketplace`
3. Тип `Scala` в `Find`
4. Установка Scala IDE

Настройка sbteclipse

1. Открыть проект воспроизведения `.\project\ plugins.sbt`
2. Добавьте следующую команду в `plugins.sbt` для преобразования проекта eclipse

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

3. Откройте команду и перейдите к воспроизведению проекта, например, `cd C:\play\play-scala` . Введите следующее в командной строке

   затмение активатора

Импорт проекта

1. Перейдите в меню `File` > `Import` в Eclipse
2. Выбор `Existing Projects into Workspace`
3. Выберите корневой каталог

Теперь ваш проект готов к просмотру и редактированию в Eclipse IDE.

Прочитайте Настройка предпочтительной среды IDE онлайн: https://riptutorial.com/ru/playframework/topic/4437/настройка-предпочтительной-среды-ide

глава 8: Работа с JSON - Scala

замечания

Официальная документация Документация по упаковке

Вы можете использовать пакет play json независимо от Play, включая

`"com.typesafe.play" % "play-json_2.11" % "2.5.3"` **В** `build.sbt` **, СМ.**

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

Examples

Создание JSON вручную

Вы можете создать дерево объектов JSON (`JsValue`) вручную

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

Или с более коротким эквивалентным синтаксисом, основанным на нескольких неявных преобразованиях:

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

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

Чтобы получить строку 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: прием запросов 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: прием запросов JSON с помощью 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);
    }
}
```

Подсказка: Преимуществом этого способа является то, что Play автоматически ответит кодом статуса HTTP 400, если запрос недействителен (Content-type был установлен в `application/json` *но JSON не был предоставлен)*

Scala: чтение JSON вручную

Если вам дана строка JSON:

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

```
    |    }
    |}""".stripMargin
```

Вы можете разобрать его, чтобы получить JsValue, представляя дерево JSON

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

И пересечь дерево для поиска конкретных значений:

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

Полезные методы

- `\` чтобы перейти к определенному ключу в объекте JSON
- `\\` чтобы перейти ко всем вхождениям определенного ключа в объект JSON, искать рекурсивно во вложенных объектах
- `.apply(idx)` (т.е. `(idx)`), чтобы перейти к индексу в массиве
- `.as[T]` для точного подтипа
- `.asOpt[T]` чтобы попытаться `.asOpt[T]` к точному подтипу, возвращая None, если это неправильный тип
- `.validate[T]` чтобы попытаться `.validate[T]` значение JSON в точный подтип, возвращая JsSuccess или 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
```

Автоматическое отображение в / из классов case

В целом, самый простой способ работы с JSON - это сопоставление классов case непосредственно с JSON (имя одного и того же поля, эквивалентные типы и т. Д.).

```
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]
```

Преобразование в 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"}}
```

Преобразование из 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))
```

Прочитайте Работа с JSON - Scala онлайн: https://riptutorial.com/ru/playframework/topic/2983/работа-с-json---scala

глава 9: скользкий

Examples

Слайдовый код запуска

В `build.sbt` убедитесь, что вы включили (здесь для Mysql и 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"
```

В вашем `application.conf` добавьте:

```
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"
```

Чтобы независимая архитектура RDBMS создала объект, подобный следующему

```
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
  }
}
```

при создании новой новой модели:

```
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}
}
```

Выходной DDL

Весь смысл использования slick - писать как можно меньше SQL-кода. После того, как вы написали определение таблицы, вы захотите создать таблицу в своей базе данных.

Если у вас есть `val table = TableQuery[MyModel]` Вы можете получить определение таблицы (код SQL - DDL), выполнив следующую команду:

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

Прочитайте скользкий онлайн: https://riptutorial.com/ru/playframework/topic/4604/скользкий

глава 10: Строительство и упаковка

Синтаксис

- активатор

Examples

Добавить каталог в дистрибутив

Чтобы добавить, например, `scripts` каталога в дистрибутив:

1. Добавьте в проект **сценарии** папки
2. В верхней части build.sbt добавьте:

```
import NativePackagerHelper._
```

3. В build.sbt добавьте сопоставление в новый каталог:

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

4. Создайте дистрибутив с ***активатором dist*** . Вновь созданный архив в `target/universal/` должен содержать новый каталог.

Прочитайте Строительство и упаковка онлайн:
https://riptutorial.com/ru/playframework/topic/6642/строительство-и-упаковка

глава 11: Тестирование устройства

Examples

Модульное тестирование - Java, Play 2.4.2.5

Помощники и подделка

Помощники класса часто используются для модульных тестов. Он имитирует приложение Play, подделывает HTTP-запросы и ответы, сеанс, файлы cookie - все, что может понадобиться для тестов. Контроллер под тестом должен быть выполнен в контексте приложения Play. Метод *fakeApplication Помощники* обеспечивает приложение для выполнения тестов. Чтобы использовать *Helpers* и *fakeApplication,* тестовый класс должен быть получен из *WithApplication* .

Должны использоваться следующие API-интерфейсы *помощников* :

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

Тест с *помощниками* выглядит следующим образом:

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

Добавление операторов импорта для методов Helpers делает код более компактным:

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

}

Контрольные контроллеры

Давайте назовем метод контроллера, который привязан к конкретному URL-адресу *маршрутов* в качестве **маршрутизируемого** метода. Вызов **маршрутизируемого** метода называется **действием** контроллера и имеет *вызов* типа Java. Игра строит так называемый обратный маршрут к каждому **действию** . Вызов обратного маршрута создает соответствующий объект *Call* . Этот механизм обратной маршрутизации используется для тестирования контроллеров.

Чтобы вызвать **действие** контроллера из теста, следует использовать следующий API-интерфейс:

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

Пример тестирования контроллера

1. *Маршруты* :

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

2. Сгенерированные обратные маршруты:

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

3. Метод *getConfId* связан с **GET** и не получает тело в запросе. Он может быть вызван для тестирования с помощью:

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

4. Метод *addParticipant* связан с **POST** . Он ожидает получить тело в запросе. Его вызов в тесте должен выполняться следующим образом:

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

Издевательство над PowerMock

Чтобы включить насмешку над тестовым классом, необходимо аннотировать следующее:

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

Издевательство над действием контроллера

Вызов контроллера *вызван* с помощью *RequestBuilder* :

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

Для вышеупомянутого addParticipant действие издевается над:

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

Чтобы вызвать метод контроллера:

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

Весь тест:

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

Издевательство над действием с корпусом JSON

Предположим, что вход является объектом типа *T.* Искажение действия может быть сделано несколькими способами.

Опция 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;
}
```

Вариант 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;
}
```

Отказывание действия с заголовком базовой аутентификации

Запрос действия издевается:

```
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;
}
```

Стыковка действия с сеансом

Запрос действия издевается:

```
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;
}
```

Класс Play *Session* - это просто расширение *HashMap <String, String>* . Его можно издеваться простым кодом:

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

Прочитайте Тестирование устройства онлайн:
https://riptutorial.com/ru/playframework/topic/6192/тестирование-устройства

кредиты

S. No	Главы	Contributors
1	Начало работы с playframework	Abhinab Kanrar, Anton, asch, Community, implicitdef, James, John, robguinness
2	Java - Hello World	Salem
3	Java - работа с JSON	Salem
4	Инъекция зависимостей - Java	asch
5	Инъекция зависимостей - Scala	asch, implicitdef
6	Использование Webservice с игрой WSClient	implicitdef, John, Salem
7	Настройка предпочтительной среды IDE	Alice, asch, implicitdef
8	Работа с JSON - Scala	Anton, asch, implicitdef, John, Salem
9	скользкий	John
10	Строительство и упаковка	JulienD
11	Тестирование устройства	asch