Play Framework Cookbook Second Edition

As web and mobile systems become more sophisticated, anchoring systems in a mature, solid framework has become increasingly important. Play 2 provides developers with the necessary tools to build robust web applications.

This book is a compilation of useful recipes aimed at helping developers discover the power of Play 2. The introductory section serves as a primer to Play Framework, wherein all the fundamentals of the framework are covered extensively. It then explains the usage of controllers and how modules can be leveraged for optimal performance. Next, the book walks you through creating and using APIs, followed by extensive real-world applications. Finally, you will learn to manage applications post production.

What this book will do for you...

- Implement the Model-View-Controller (MVC) web architecture with Play 2
- Convert data between incompatible type systems using ORM tools such as Ebean and Anorm
- Serve and receive JSON messages to transmit data objects
- Create long-running, asynchronous controllers using Akka
- Learn to use modules for Amazon S3, Redis, and MongoDB
- Consume public APIs using the Play WS API
- Make your application production-ready by integrating it with AngularJS
- Automate the creation of a portable development environment using Vagrant
- Deploy Play 2 applications with Docker

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Over 60 hands-on recipes to create dynamic and reactive web-based applications with Play 2

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Leveraging Modules'
- A synopsis of the book’s content
- More information on Play Framework Cookbook Second Edition


About the Authors

**Alexander Reelsen** is a software engineer living in Munich, Germany, where he has been working on different software systems such as a touristic booking engine, campaign management, and messaging platform, and a B2B e-commerce portal. He began using the Play Framework in 2009, and was immediately astonished by the sheer simplicity of this framework, while still dealing with pure Java. Other interests include scaling shared-nothing web architectures and NoSQL databases.

Being a system engineer, when he started playing around with Linux at the age of 14, Alexander got to know about software engineering during his studies and decided that web applications are more interesting than system administration.

When not doing something hacky, he enjoys playing a good game of basketball or street ball.

Sometimes he even tweets at [http://twitter.com/spinscale](http://twitter.com/spinscale) and can be reached anytime at alexander@reelsen.net
Giancarlo Inductivo is the founder of DYNAMIC OBJX, a web and mobile development firm based in Manila, Philippines, and is the current technology head. Over the last 10 years, Giancarlo has been based in San Francisco and Manila, working in software development for companies in various industries such as online recruitment, social networking, Internet media, and software consultancy.

With DYNAMIC OBJX, Giancarlo has been utilizing the Play Framework for client projects since version 1.2 and has successfully launched many client projects using version 2.x for systems such as inventory management, logistics, financials, social media content aggregation, mobile content management, and so on.

Giancarlo is also interested in other technologies such as Apache Spark, Docker, and Arduino.

Apart from technology, Giancarlo is also obsessed with his Stratocasters, the Golden State Warriors, and his two kids, Makayla and Elijah.

You can reach Giancarlo on Quora:

http://www.quora.com/Giancarlo-Inductivo
Preface

Web applications have come a long way in the last 5-10 years. From the glory days of Geocities and Friendster, the advent of massively popular websites, in the case of social media sites, such as Facebook, and Twitter, and the more utilitarian web applications that are Software as a service (SAAS) offerings such as SalesForce and Github, there is no denying that with all these advancements in consumer and enterprise web software, there arises a need for a solid web technology platform to build on top of, not only for end-user web clients but also with a variety of sophisticated and complex backend services, all of which can compose the modern web application.

This is where Play Framework 2.0 comes in to the scene. Play provides developers with a powerful and mature web development platform that is lightweight and stateless by nature, which was made with development speed and web application scalability in mind.

This book aims to give readers a deeper understanding of different parts of Play Framework through concise code recipes based on very common use cases and scenarios. You will be guided through the basic concepts and abstractions used in Play, and we will dive into more advanced and relevant topics such as creating RESTful APIs, using a third-party cloud storage service to store uploaded images, posting messages to external message queues, and deploying Play web applications using Docker.

By providing relevant recipes, this book hopes to equip developers with the necessary building blocks required for creating the next Facebook or SalesForce using Play Framework.

What this book covers

Chapter 1, Basics of Play Framework, introduces Play Framework and its capabilities. This chapter also introduces the essential Play components, such as the Controller and View templates. Finally, we cover how to do unit testing for a Play Framework model and controller classes.

Chapter 2, Using Controllers, discusses Play Controllers in depth. This chapter demonstrates how to use controllers with other web components such as request filters, sessions, and JSON. It also touches on how Play and Akka can be utilized together from a controller.
Chapter 3, *Leveraging Modules*, looks into utilizing official Play 2 modules as well as other third-party modules. This should help developers speed up their development by reusing and integrating existing modules.

Chapter 4, *Creating and Using Web APIs*, discusses how to create secure RESTful API endpoints with Play. This chapter also discusses how to consume other web-based APIs using the Play WS library.

Chapter 5, *Creating Plugins and Modules*, discusses how to write Play Modules and Plugins, and also tells us how to publish Play modules to a private repository on Amazon S3.

Chapter 6, *Practical Module Examples*, adds to the previous chapter regarding Play Modules and discusses more practical examples of integrating modules such as message queues and Search Services.

Chapter 7, *Deploying Play 2 Web Apps*, discusses deployment scenarios for different environments for Play web applications using tools such as Docker and Dokku.

Chapter 8, *Additional Play Information*, discusses topics relevant to developers such as integrating with IDEs and using other third-party Cloud services. This chapter also discusses how to build a Play developer environment from scratch using Vagrant.
In this chapter, we will cover the following recipes:

- Dependency injection with Spring
- Dependency injection using Guice
- Utilizing MongoDB
- Utilizing MongoDB and GridFS
- Utilizing Redis
- Integrating Play application with Amazon S3
- Integrating Play application with Typesafe Slick
- Utilizing play-mailer
- Integrating Bootstrap and WebJars

**Introduction**

In this chapter, we will look at utilizing Play and other third-party modules to address commonly required functionalities of modern web applications. As web applications and web application frameworks mature and evolve, the need for a modular and extensible system as part of the core web application framework becomes increasingly important. This is achievable and straightforward with Play Framework 2.0.

**Dependency injection with Spring**

For this recipe, we will explore how to integrate the popular Spring Framework with a Play application. We will use Spring for bean instantiation and injection using Play controllers and service classes.
Leveraging Modules

How to do it...

For Java, we need to take the following steps:

1. Run the foo_java application with Hot-Reloading enabled:
   ```
   activator "-run"
   ```

2. Declare Spring as a project dependency in build.sbt:
   ```
   "org.springframework" % "spring-context" % "3.2.2.RELEASE",
   "org.springframework" % "spring-aop" % "3.2.2.RELEASE",
   "org.springframework" % "spring-expression" % "3.2.2.RELEASE"
   ```

3. Create a new admin controller in foo_java/app/controllers/
   AdminController.java with the following code:
   ```java
   package controllers;

   import play.*;
   import play.mvc.*;
   import org.springframework.beans.factory.annotation.Autowired;
   import services.AdminService;

   @org.springframework.stereotype.Controller
   public class AdminController {
       
       @Autowired
       private AdminService adminService;

       public Result index() {
           return play.mvc.Controller.ok("This is an admin-only resource: "+ adminService.getFoo());
       }
   }
   ```

4. Create an admin service interface class in foo_java/app/services/
   AdminServices.java and a mock admin service implementation class in foo_java/app/services/AdminServiceImpl.java with the following content:
   ```java
   // AdminService.java
   package services;

   public interface AdminService {
       String getFoo();
   }

   // AdminServiceImpl.java
   ```
package services;

import org.springframework.stereotype.Service;

@Service
public class AdminServiceImpl implements AdminService {
    @Override
    public String getFoo() {
        return "foo";
    }
}

5. Add a new routes entry for the newly added action to foo_java/conf/routes:
GET /admins @controllersAdminController.index

6. Add a Global settings class to the foo_java/app/Global.java file with the following content:
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import play.Application;
import play.GlobalSettings;

    public class Global extends GlobalSettings {
        private ApplicationContext ctx;

        @Override
        public void onStart(Application app) {
            ctx = new AnnotationConfigApplicationContext(SpringConfig.class);
        }

        @Override
        public <A> A getControllerInstance(Class<A> clazz) {
            return ctx.getBean(clazz);
        }
    }

7. Add the Spring configuration class to foo_java/app/SpringConfig.java with the following content:
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

    @Configuration
@ComponentScan({"controllers", "services"})
public class SpringConfig {
}

8. Request our new route and examine the response body to confirm:

```
$ curl -v http://localhost:9000/admins
* Hostname was NOT found in DNS cache
*   Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admins HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 35
<
* Connection #0 to host localhost left intact
This is an admin-only resource: foo%
```

For Scala, we need to take the following steps:

1. Run the foo_scala application with Hot-Reloading enabled:
   activator "-run"

2. Declare Spring as a project dependency in build.sbt:
   
   "org.springframework" % "spring-context" % "3.2.2.RELEASE"

3. Create a new admin controller in foo_scala/app/controllers/
   AdminController.scala with the following content:

   ```scala
   package controllers

   import play.api.mvc.{Action, Controller}
   import services.AdminService

   class AdminController(implicit adminService: AdminService)
   extends Controller {
     def index = Action {
   ```
4. Create an admin service class in foo_scala/app/services/AdminServices.scala with the following content:

```scala
package services

class AdminService {
    def foo = "foo"
}
```

5. Add a new routes entry for the newly added action to foo_scala/conf/routes:

```scala
GET /admins                    @controllersAdminController.index
```

6. Add a Global settings class to the foo_scala/app/Global.scala with the following content:

```scala
import org.springframework.context.ApplicationContext
import org.springframework.context.annotation.AnnotationConfigApplicationContext

object Global extends play.api.GlobalSettings {
    private val ctx: ApplicationContext = new AnnotationConfigApplicationContext(classOf[SpringConfig])

    override def getControllerInstance[A](clazz: Class[A]): A = {
        return ctx.getBean(clazz)
    }
}
```

7. Add the Spring configuration class to foo_scala/app/SpringConfig.scala with the following content:

```scala
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Bean
import controllers._
import services._

@Configuration
class SpringConfig {
    @Bean
```
Leveraging Modules

```scala
implicit def adminService: AdminService = new AdminService

@Bean
def adminController: AdminController = new AdminController
}
```

8. Request our new route and examine the response body to confirm:

```bash
$ curl -v http://0.0.0.0:9000/admins
* Hostname was NOT found in DNS cache
* Trying 0.0.0.0...
* Connected to 0.0.0.0 (127.0.0.1) port 9000 (#0)
> GET /admins HTTP/1.1
> User-Agent: curl/7.37.1
> Host: 0.0.0.0:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 35
<
* Connection #0 to host 0.0.0.0 left intact
This is an admin-only resource: foo%
```

How it works...

In this recipe, we configured our Play application to utilize Dependency injection in our controllers and service classes using Spring. We configured Spring in the Global settings file and loaded the SpringConfig class, which will contain our Spring-specific configurations.

Dependency injection using Guice

For this recipe, we will explore how to integrate Google Guice with a Play application. We will use Guice for bean instantiation and injection using Play controllers and service classes.

How to do it...

For Java, we need to take the following steps:

1. Run the foo_java application with Hot-Reloading enabled:
   ```bash
   activator "-run"
   ```
2. Declare the guice module as a project dependency in build.sbt:
"com.google.inject" % "guice" % "3.0"

3. Configure Guice by modifying the contents of the Global settings class:
   import com.google.inject.AbstractModule;
   import com.google.inject.Guice;
   import com.google.inject.Injector;
   import com.google.inject.Singleton;
   import play.GlobalSettings;
   import services.*;

   public class Global extends GlobalSettings {
     private Injector injector = Guice.createInjector(new AbstractModule() {
       @Override
       protected void configure() {
         bind(CategoryService.class).to(
             CategoryServiceImpl.class).in(Singleton.class);
       }
     });

     @Override
     public <T> T getControllerInstance(Class<T> clazz) {
       return injector.getInstance(clazz);
     }
   }

4. Create a category controller in foo_java/app/controllers/CategoryController.java by adding the following content:
   package controllers;

   import com.google.inject.Inject;
   import play.libs.Json;
   import play.mvc.Controller;
   import play.mvc.Result;
   import services.CategoryService;

   public class CategoryController extends Controller {
     @Inject
     private CategoryService categoryService;

     public Result index() {
       return ok(Json.toJson(categoryService.list()));
     }
   }
5. Create a category service interface in foo_java/app/services/CategoryService.java by adding the following content:

```java
package services;

import java.util.List;

public interface CategoryService {
    List<String> list();
}
```

6. Create a category service implementation class in foo_java/app/services/CategoryServiceImpl.java by adding the following content:

```java
package services;

import java.util.Arrays;
import java.util.List;

public class CategoryServiceImpl implements CategoryService {
    @Override
    public List<String> list() {
        return Arrays.asList("Manager", "Employee", "Contractor");
    }
}
```

7. Add a new routes entry for the newly added action to foo_java/conf/routes:

```
GET /categories @controllers.CategoryController.index
```

8. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/categories
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /categories HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> < HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 35
<
* Connection #0 to host localhost left intact
["Manager","Employee","Contractor"]
```
For Scala, we need to take the following steps:

1. Run the foo_scala application with Hot-Reloading enabled:
   ```scala
   activator "-run"
   ```

2. Declare the secure_social module as a project dependency in build.sbt:
   ```scala
   "com.google.inject" % "guice" % "3.0"
   ```

3. Configure Guice by modifying the contents of the Global settings class:
   ```scala
   import com.google.inject.{Guice, AbstractModule}
   import play.api.GlobalSettings
   import services._

   object Global extends GlobalSettings {
     val injector = Guice.createInjector(new AbstractModule {
       protected def configure() {
         bind(classOf[CategoryService]).to(classOf[CategoryServiceImpl])
       }
     })
   }

   override def getControllerInstance[A](controllerClass: Class[A]): A = {
     injector.getInstance(controllerClass)
   }
   }
   ```

4. Create a category controller in foo_scala/app/controllers/CategoryController.scala by adding the following content:
   ```scala
   package controllers

   import play.api.mvc._
   import play.api.libs.json.Json._
   import com.google.inject.
   import services._

   @@Singleton
   class CategoryController @Inject()(categoryService: CategoryService) extends Controller {
     def index = Action {
       Ok(toJson(categoryService.list))
     }
   }
   ```
Leveraging Modules

5. Create a category service in `foo_scala/app/services/CategoryService.scala` by adding the following content:

```scala
package services

trait CategoryService {
  def list: Seq[String]
}

class CategoryServiceImpl extends CategoryService {
  override def list: Seq[String] = Seq("Manager", "Employee", "Contractor")
}
```

6. Add a new routes entry for the newly added action to `foo_scala/conf/routes`:

```
GET /categories    @controllers.CategoryController.index
```

7. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/categories
* Hostname was NOT found in DNS cache
*   Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /categories HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 35
<
* Connection #0 to host localhost left intact
["Manager","Employee","Contractor"]
```

How it works...

In this recipe, we configured our Play application to utilize Dependency injection in our controllers and service classes using Google Guice. We configured Guice in the Global settings file, which will contain our Guice-specific configurations.
Utilizing MongoDB

For this recipe, we will explore how to utilize MongoDB, the popular NoSQL library, within a Play application. MongoDB is one of the most widely-used NoSQL databases, and it most certainly has been a viable option as a datastore for many modern web applications. We will be using the Scala module, play-plugins-salat, which is an Object relation mapping tool that uses the official MongoDB Scala driver Casbah. This will be a Scala-only recipe.

For more information about Casbah, please refer to https://github.com/mongodb/casbah.

How to do it...

Let's take the following steps:

1. Run the foo_scala application with Hot-Reloading enabled:
   
   activator "-run"

2. Declare play-plugins-salat as a project dependency in build.sbt:
   
   "se.radley" %% "play-plugins-salat" % "1.5.0"

3. Add additional salat and MongoDB directives to build.sbt:
   
   import play.PlayImport.PlayKeys._
   import play.twirl.sbt.Import.TwirlKeys
   
   routesImport += "se.radley.plugin.salat.Binders._"
   TwirlKeys.templateImports += "org.bson.types.ObjectId"

4. Declare the salat plugin in foo_scala/conf/play.plugins:
   
   500:se.radley.plugin.salat.SalatPlugin

5. Declare the MongoDB instance information in foo_scala/conf/application.conf:
   
   mongodb.default.db = "cookbookdb"

6. Modify foo_scala/app/controllers/WarehouseController.scala by adding the following content:
   
   package controllers
   
   import models._
   import play.api.libs.json._
   import play.api.mvc.{BodyParsers, Action, Controller}
import se.radley.plugin.salat.Binders.ObjectId

object WarehouseController extends Controller {
  implicit val objectIdReads = se.radley.plugin.salat.Binders.objectIdReads
  implicit val objectIdWrites = se.radley.plugin.salat.Binders.objectIdWrites
  implicit val warehouseWrites = Json.writes[Warehouse]
  implicit val warehouseReads = Json.reads[Warehouse]

  def index = Action {
    val list = Warehouse.list
    Ok(Json.toJson(list))
  }

  def create = Action(BodyParsers.parse.json) { implicit request =>
    val post = request.body.validate[Warehouse]
    post.fold(
      errors => {
        BadRequest(Json.obj("status" -> "error", "message" -> JsError.toFlatJson(errors)))
      },
      warehouse => {
        Warehouse.create(warehouse)
        Created(Json.toJson(warehouse))
      })
  }
}

7. Add new routes for the newly added action to foo_scala/conf/routes:
   GET /warehouses controllers.WarehouseController.index
   POST /warehouses controllers.WarehouseController.create

8. Add the collection mapping for the warehouse model to foo_scala/app/models/Warehouse.scala:

   package models

   import play.api.Play.current
   import com.mongodb.casbah.commons.MongoDBObject
   import com.novus.salat.dao._
   import se.radley.plugin.salat._
   import se.radley.plugin.salat.Binders._
import mongoContext._

case class Warehouse(id: Option[ObjectId] = Some(new ObjectId), name: String, location: String)

object Warehouse extends ModelCompanion[Warehouse, ObjectId] {
  val dao = new SalatDAO[Warehouse, ObjectId](collection = mongoCollection("warehouses")) {}

  def list = dao.find(ref = MongoDBObject()).toList
  def create(w: Warehouse) = dao.save(w)
}

9. Add the Mongo context to foo_scala/app/models/mongoContext.scala:

   package models

   import com.novus.salat.dao._
   import com.novus.salat.annotations._
   import com.mongodb.casbah.Imports._
   import play.api.Play
   import play.api.Play.current

   package object mongoContext {
     implicit val context = {
       val context = new Context {
         val name = "global"
         override val typeHintStrategy = StringTypeHintStrategy(when = TypeHintFrequency.WhenNecessary, typeHint = "_t")
       }
       context.registerGlobalKeyOverride(remapThis = "id", toThisInstead = "_id")
       context.registerClassLoader(Play.classloader)
       context
     }
   }

10. Add a new warehouse record by accessing the warehouse post endpoint using curl:
    $ curl -v -X POST http://localhost:9000/warehouses --header "Content-type: application/json" --data '{"name":"Warehouse A", "location":"Springfield"}"
        * Hostname was NOT found in DNS cache
        * Trying ::1...
Leveraging Modules

* Connected to localhost (::1) port 9000 (#0)
  > POST /warehouses HTTP/1.1
  > User-Agent: curl/7.37.1
  > Host: localhost:9000
  > Accept: */*
  > Content-type: application/json
  > Content-Length: 48
  >
  * upload completely sent off: 48 out of 48 bytes
  < HTTP/1.1 201 Created
  < Content-Type: application/json; charset=utf-8
  < Content-Length: 47
  <
  * Connection #0 to host localhost left intact
  {"name":"Warehouse A","location":"Springfield"}

11. View all warehouse records by accessing the warehouse index endpoint using curl:

$ curl -v http://localhost:9000/warehouses
  * Hostname was NOT found in DNS cache
  * Trying ::1...
  * Connected to localhost (::1) port 9000 (#0)
  > GET /warehouses HTTP/1.1
  > User-Agent: curl/7.37.1
  > Host: localhost:9000
  > Accept: */*
  >
  < HTTP/1.1 200 OK
  < Content-Type: application/json; charset=utf-8
  < Content-Length: 241
  <
  * Connection #0 to host localhost left intact
  [{"id":"5490fde9e0820cf6df38584c","name":"Warehouse A","location":"Springfield"}]%
In this recipe, we created a new URL route and action that will insert and retrieve warehouse records from a MongoDB instance. We used the Play module play-plugins-salat and configured the connection in `fooScala/conf/application.conf`. We then mapped our MongoDB collection in the warehouse model class:

```scala
case class Warehouse(id: Option[ObjectId] = Some(new ObjectId),
                    name: String, location: String)
```

Next, we invoked the appropriate warehouse companion object methods from the warehouse controller:

```scala
val list = Warehouse.list
Warehouse.create(warehouse)
```

We also declared our JSON binders for the warehouse model and MongoDB's `ObjectId` in the warehouse controller:

```scala
implicit val objectIdReads = se.radley.plugin.salat.Binders.objectIdReads
implicit val objectIdWrites = se.radley.plugin.salat.Binders.objectIdWrites
implicit val warehouseWrites = Json.writes[Warehouse]
implicit val warehouseReads = Json.reads[Warehouse]
```

For this recipe, we will explore how to store and deliver files with Play applications by using MongoDB and GridFS. We will continue by adding to the previous recipe. As with the previous recipe, this recipe will be Scala only.

Let's take the following steps:

1. Run the `fooScala` application with Hot-Reloading enabled:
   ```bash
   activator "-run"
   ```

2. Modify `foo Scala/app/controllers/WarehouseController.scala` by adding the following content:
   ```scala
   import java.text.SimpleDateFormat
   import play.api.libs.iteratee.Enumerator
   
   def upload = Action(parse.multipartFormData) { request =>
   ```
request.body.file("asset") match {
  case Some(asset) => {
    val gridFs = Warehouse.assets
    val uploadedAsset = gridFs.createFile(asset.ref.file)
    uploadedAsset.filename = asset.filename
    uploadedAsset.save()

    Ok("Asset is available at http://localhost:9000/
    warehouses/assets/%s".format(uploadedAsset.id))
  }
  case None => {
    BadRequest
  }
}

def retrieveFile(id: ObjectId) = Action {
  import com.mongodb.casbah.Implicits._
  import play.api.libs.concurrent.Execution.Implicits._
  val gridFs = Warehouse.assets

  gridFs.findOne(Map("_id" -> id)) match {
    case Some(f) => Result(
      ResponseHeader(OK, Map(
        CONTENT_LENGTH -> f.length.toString,
        CONTENT_TYPE -> f.contentType.getOrElse(BINARY),
        DATE -> new SimpleDateFormat("EEE, dd MMM yyyy
        HH:mm:ss 'GMT'", java.util.Locale.US).format(f.uploadDate)
      )),
      Enumerator.fromStream(f.inputStream)
    )
    case None => NotFound
  }
}

3. Add new routes for the newly added action to foo_scala/conf/routes:

   POST /warehouses/ assets/upload controllers.WarehouseController.
   upload
   GET /warehouses/ assets/:id controllers.WarehouseController.
   retrieveFile(id: ObjectId)
4. Modify the collection mapping for the warehouse model in `foo_scala/app/models/Warehouse.scala`:

```scala
val assets = gridFS("assets")

def upload(asset: File) = {
  assets.createFile(asset)
}

def retrieve(filename: String) = {
  assets.find(filename)
}
```

5. Upload a new warehouse asset file by accessing the warehouse upload endpoint using `curl`:

```
$ curl -v http://localhost:9000/warehouses/assets/upload -F "asset=@/tmp/1.jpg"
   * Hostname was NOT found in DNS cache
   * Trying ::1...
   * Connected to localhost (::1) port 9000 (#0)
   > POST /warehouses/assets/upload HTTP/1.1
   > User-Agent: curl/7.37.1
   > Host: localhost:9000
   > Accept: */*
   > Content-Length: 13583
   > Expect: 100-continue
   > Content-Type: multipart/form-data; boundary=------------------------4a001bdeff39c089

   >
   < HTTP/1.1 100 Continue
   < HTTP/1.1 200 OK
   < Content-Type: text/plain; charset=utf-8
   < Content-Length: 86
   <
   * Connection #0 to host localhost left intact

   Asset is available at http://localhost:9000/warehouses/assets/549121fbe082fc374fa6cb63%
```
6. Verify that our file delivery URL route is working by accessing the URL in a web browser that is part of the output of the previous step:

http://localhost:9000/warehouses/assets/549121fbe082fc374fa6cb63

**How it works...**

In this recipe, we created new URL routes and actions that will be used to upload and retrieve warehouse asset files in a MongoDB instance using GridFS. We added the GridFS reference to our collection mapping file in `foo_scala/app/models/Warehouse.scala`:

```scala
val assets = gridFS("assets")
```

We then added the respective methods for file upload and retrieval:

```scala
def upload(asset: File) = {
  assets.createFile(asset)
}
def retrieve(filename: String) = {
  assets.find(filename)
}
```

Next, we created the actions in `foo_scala/app/controllers/WarehouseController.scala`, which will handle the actual file upload and retrieval requests.

**Utilizing Redis**

For this recipe, we will explore how Play applications integrate with Redis using Play cache. Redis is a widely used key-value database, usually utilized as an intermediary object cache for modern web applications. This recipe requires a running Redis instance that our Play 2 web application can interface with.

**How to do it...**

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:
   ```bash
   activator "-run"
   ```

2. Declare Redis as a project dependency in `build.sbt`:
   ```scala
   "com.typesafe.play.plugins" %% "play-plugins-redis" % "2.3.1"
   ```
3. Declare the repository hosting Sedis, a library dependency of play-plugins-redis in build.sbt:

   resolvers += "Sedis repository" at "http://pk11-scratch.googlecode.com/svn/trunk/"

4. Enable the play-mailer plugin by declaring it in foo_java/conf/play.plugins:

   550:com.typesafe.plugin.RedisPlugin

5. Specify your Redis host information in foo_java/conf/application.conf:

   ehcacheplugin=disabled
   redis.uri="redis://127.0.0.1:6379"

6. Modify foo java/app/controllers/Application.java by adding the following code:

   import play.cache.*;

   public static Result displayFromCache() {
     final String key = "myKey";
     String value = (String) Cache.get(key);

     if (value != null && value.trim().length() > 0) {
       return ok("Retrieved from Cache: " + value);
     } else {
       Cache.set(key, "Let's Play with Redis!");
       return ok("Setting key value in the cache");
     }
   }

7. Add a new routes entry for the newly added action to foo_java/conf/routes:

   GET /cache controllers.Application.displayFromCache

8. Request our new route and examine the response body to confirm that our displayFromCache action is setting the key value for the first time:

   $ curl -v http://localhost:9000/cache
   * Hostname was NOT found in DNS cache
   * Trying 127.0.0.1...
   * Connected to localhost (127.0.0.1) port 9000 (#0)
   > GET /cache HTTP/1.1
   > User-Agent: curl/7.37.1
   > Host: localhost:9000
Leveraging Modules

> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 30
<
* Connection #0 to host localhost left intact
Setting key value in the cache%

9. Request the /cache route again to be able to view the value of the cache key:

$ curl -v http://localhost:9000/cache
* Hostname was NOT found in DNS cache
  Trying 127.0.0.1...
  * Connected to localhost (127.0.0.1) port 9000 (#0)
> GET /cache HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 43
<
* Connection #0 to host localhost left intact
Retrieved from Cache: Let's Play with Redis!%

For Scala, we need to take the following steps:

1. Run the foo_scala application with Hot-Reloading enabled:
   activator "-run"

2. Declare Redis as a project dependency in build.sbt:
   "com.typesafe.play.plugins" %% "play-plugins-redis" % "2.3.1"

3. Declare the repository hosting Sedis, a library dependency of play-plugins-redis, in build.sbt:
   resolvers += "Sedis repository" at "http://pk11-scratch.googlecode.com/svn/trunk/"
4. Enable the play-mailer plugin by declaring it in `foo_scala/conf/play.plugins`:
   ```scala
   550:com.typesafe.plugin.RedisPlugin
   ```

5. Specify your Redis host information in `foo_scala/conf/application.conf`:
   ```properties
   ehcache.plugin=disabled
   redis.uri="redis://127.0.0.1:6379"
   ```

6. Modify `foo_scala/app/controllers/Application.scala` by adding the following code:
   ```scala
   import scala.util.Random
   import play.api.cache._
   import play.api.Play.current

   def displayFromCache = Action {
     val key = "myKey"
     Cache.getAs[String](key) match {
       case Some(myKey) => {
         Ok("Retrieved from Cache: %s".format(myKey))
       }
       case None => {
         Cache.set(key, "Let's Play with Redis!")
         Ok("Setting key value in the cache")
       }
     }
   }
   ```

7. Add a new routes entry for the newly added action to `foo_scala/conf/routes`:
   ```routes
   GET /cache controllers.Application.displayFromCache
   ```

8. Request our new route and examine the response body to confirm that our `displayFromCache` action is setting the key value for the first time:
   ```bash
   $ curl -v http://localhost:9000/cache
   * Hostname was NOT found in DNS cache
   * Trying ::1...
   * Connected to localhost (::1) port 9000 (#0)
   > GET /cache HTTP/1.1
   > User-Agent: curl/7.37.1
   > Host: localhost:9000
   > Accept: */*
   >
   < HTTP/1.1 200 OK
Leveraging Modules

9. Request the /cache route again to be able to view the value of the cache key:

   $ curl -v http://localhost:9000/cache
   * Hostname was NOT found in DNS cache
   * Trying 127.0.0.1...
   * Connected to localhost (127.0.0.1) port 9000 (#0)
   > GET /cache HTTP/1.1
   > User-Agent: curl/7.37.1
   > Host: localhost:9000
   > Accept: */*
   >
   < HTTP/1.1 200 OK
   < Content-Type: text/plain; charset=utf-8
   < Content-Length: 43
   <
   * Connection #0 to host localhost left intact
   Retrieved from Cache: Let's Play with Redis!

How it works...

In this recipe, we created a new URL route and action that will interact with our Redis instance. To follow on with this recipe, you will need the following running Redis instance to connect to:

   $ redis-server /usr/local/etc/redis.conf
   [2407] 14 Apr 12:44:20.623 * Increased maximum number of open files to 10032 (it was originally set to 2560).
   Redo 2.8.17 (00000000/0)
   64 bit
   Running in stand alone mode
For more information about installing and running a Redis server, please refer to http://redis.io/topics/quickstart.

We configured the Play Redis module by declaring the necessary dependencies and repository settings in build.sbt. We then configured the connection to the Redis instance in conf/application.conf. Finally, we loaded the Redis play-plugin in conf/play.plugins:

```
550:com.typesafe.plugin.RedisPlugin
```

The displayFromCache action, when invoked, has two distinct functions. First, it attempts to retrieve a value from the cache. If it is able to retrieve a value from the Redis cache, it prints the contents of the value in the response body. If it is unable to retrieve a value from the Redis cache, it sets a random string value to the key and prints a status message in the response body.

We then used curl to test out this new route and accessed the route twice; the action printed out two different messages in the response body.

**Integrating Play application with Amazon S3**

For this recipe, we will explore how Play applications can upload files directly to Amazon Web Services (AWS) S3, a popular cloud storage solution.

For more information about S3, please refer to http://aws.amazon.com/s3/.
How to do it...

For Java, we need to take the following steps:

1. Run the foo_java application with Hot-Reloading enabled:
   ```
   activator "-run"
   ```

2. Declare play-s3 as a project dependency in build.sbt:
   ```
   "com.amazonaws" % "aws-java-sdk" % "1.3.11"
   ```

3. Specify your AWS credentials in foo_java/conf/application.conf:
   ```
   aws.accessKeyId="YOUR S3 ACCESS KEY"
   aws.secretKey="YOUR S3 SECRET KEY"
   fooscala.s3.bucketName="YOUR S3 BUCKET NAME"
   ```

4. Modify foo_java/app/controllers/Application.java by adding the following code:
   ```java
   import com.amazonaws.auth.*;
   import com.amazonaws.services.s3.*;
   import com.amazonaws.services.s3.model.*;

   public static Result s3Upload() {
     return ok(views.html.s3.render());
   }

   public static Result submitS3Upload() {
     Http.MultipartFormData body = request().body().asMultipartFormData();
     Http.MultipartFormData.FilePart profileImage = body.getFile("profile");

     if (profileImage != null) {
       try {
         File file = profileImage.getFile();
         String filename = profileImage.getFilename();

         String accessKey = Play.application().configuration().getString("aws.accessKeyId");
         String secret = Play.application().configuration().getString("aws.secretKey");
         String bucketName = Play.application().configuration().getString("fooscala.s3.bucketName");

         try {
           ```
AWSCredentials awsCredentials = new BasicAWSCredentials(accessKey, secret);
AmazonS3 s3Client = new AmazonS3Client(awsCredentials);
AccessControlList acl = new AccessControlList();
acl.grantPermission(GroupGrantee.AllUsers, Permission.Read);
s3Client.createBucket(bucketName);
s3Client.putObject(new PutObjectRequest(bucketName, filename, file).
    withAccessControlList(acl));

String img = "http://" + bucketName + ".s3.amazonaws.com/" + filename;
return ok("Image uploaded: " + img);

5. Add a new routes entry for the newly added action to foo_java/conf/routes:
   GET     /s3_upload      controllers.Application.s3Upload
   POST    /s3_upload      controllers.Application.submitS3Upload

6. Add the S3 file upload submission View template to foo_java/app/views/s3.scala.html:
   @helper.form(action = routes.Application.submitS3Upload,
        'enctype -> "multipart/form-data") {
       <input type="file" name="profile">
       <p>
       <input type="submit">
    </p>
   }
Leveraging Modules

For Scala, we need to take the following steps:

1. Run the foo_scala application with Hot-Reloading enabled:
   ```
   activator "-run"
   ```

2. Declare play-s3 as a project dependency in build.sbt:
   ```
   "nl.rhinofly" %% "play-s3" % "5.0.2",
   ```

3. Declare the custom repository where the play-s3 module is hosted:
   ```
   resolvers += "Rhinofly Internal Repository" at "http://maven-repository.rhinofly.net:8081/artifactory/libs-release-local"
   ```

4. Specify your AWS credentials in foo_scala/conf/application.conf:
   ```
   aws.accessKeyId="YOUR S3 ACCESS KEY"
   aws.secretKey="YOUR S3 SECRET KEY"
   fooscala.s3.bucketName="YOUR S3 BUCKET NAME"
   ```

5. Modify foo_scala/app/controllers/Application.scala by adding the following code:
   ```scala
   import play.api.Play.current
   import fly.play.s3._
   def s3Upload = Action {
     Ok(s3())
   }

   def submitS3Upload = Action(parse.multipartFormData) {
     request =>
       import play.api.Play

       request.body.file("profile") match {
         case Some(profileImage) => {
           val bucketName = Play.current.configuration.getString("fooscala.s3.bucketName").get
           val bucket = S3(bucketName)

           val filename = profileImage.filename
           val contentType = profileImage.content
           val byteArray = Files.toByteArray(profileImage.ref.
           file)

           val result = bucket.add(BucketFile(filename, contentType.get, byteArray, Option(PUBLIC_READ), None))
           val future = Await.result(result, 10 seconds)
         }
       }
   }
   ```
6. Add a new routes entry for the newly added action to `foo_scala/conf/routes`:

   GET /s3_upload controllers.Application.s3Upload
   POST /s3_upload controllers.Application.submitS3Upload

Add the s3 file upload submission view template in `foo_scala/app/views/s3.scala.html`:

   @helper.form(action = routes.Application.submitS3Upload, 
                'enctype -> "multipart/form-data") {
      <input type="file" name="profile">
      <p>
      <input type="submit">
      </p>
   }

How it works...

In this recipe, we created a new URL route and action that received the uploaded file. We then pushed this file to Amazon S3 using the RhinoFly S3 module by supplying the S3 access and secret keys in `conf/application.conf`. We also specified our S3 bucket name in `conf/application.conf` for future use. We are able to retrieve this value by using Play's configuration API:

   val bucketName = Play.current.configuration.getString("fooscala.s3.bucketName").get

We then printed the location of the uploaded file to the response body for easy verification:

   Ok("Image uploaded to: http://%s.s3.amazonaws.com/%s".format(bucketName, filename))

You should now see the response in the web browser with the following text:

   Image uploaded to: http://<YOUR_BUCKET_NAME>.s3.amazonaws.com/<FILENAME>
Leveraging Modules

You can also use the AWS Management Console to verify the file upload in the S3 section:

Integrating with Play application Typesafe Slick

In this recipe, we will explore how we can integrate Typesafe Slick with Play applications using the play-slick module. Typesafe Slick is a relational mapping tool built on Scala and is handy for managing database objects like native Scala types.

How to do it...

We need to take the following steps:

1. Run the foo_scala application with Hot-Reloading enabled:
   
```bash
   activator "-run"
   ```

2. Declare play-slick as a project dependency in build.sbt:
   
```
   "com.typesafe.play" %% "play-slick" % "0.8.1"
   ```
3. Specify your database host information in foo_scala/conf/application.conf:
   
   ```
   db.default.driver=org.h2.Driver
   db.default.url="jdbc:h2:mem:play"
   db.default.user=sa
   db.default.password=""
   slick.default="models.*"
   ```

4. Create the new supplier controller in foo_scala/app/controllers/SupplierController.scala with the following content:

   ```scala
   package controllers

   import play.api.mvc.{BodyParsers, Controller}
   import play.api.db.slick._
   import play.api.libs.json._
   import play.api.Play.current
   import models._

   object SupplierController extends Controller {
     implicit val supplierWrites = Json.writes[Supplier]
     implicit val supplierReads = Json.reads[Supplier]

     def index = DBAction { implicit rs =>
       Ok(Json.toJson(Suppliers.list))
     }

     def create = DBAction(BodyParsers.parse.json) { implicit rs =>
       val post = rs.request.body.validate[Supplier]
       post.fold(
         errors => {
           BadRequest(Json.obj("status" ->"error", "message" -> JsError.toFlatJson(errors)))
       },
       supplier => {
         Suppliers.create(supplier)
         Created(Json.toJson(supplier))
       }
     }
   }
   ```
5. Create a Slick mapping for suppliers in the foo_scala/app/models/Suppliers.scala file with the following content:

```scala
package models

import scala.slick.driver.H2Driver.simple._
import scala.slick.lifted.Tag

case class Supplier(id: Option[Int], name: String, contactNo: String)

class Suppliers(tag: Tag) extends Table[Supplier](tag, "SUPPLIERS") {
  def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
  def name = column[String]("NAME")
  def contactNo = column[String]("CONTACT_NO")

  def * = (id.?, name, contactNo) <> (Supplier.tupled, Supplier.unapply)
}

object Suppliers {
  val suppliers = TableQuery[Suppliers]

  def list(implicit s: Session) = suppliers.sortBy(m => m.name.asc).list
  def create(supplier: Supplier)(implicit s: Session) = suppliers.insert(supplier)
}
```

6. Add the new routes for the Supplier controller in foo_scala/conf/routes:

```scala
GET  /suppliers controllers.SupplierController.index
POST  /suppliers controllers.SupplierController.create
```

7. Request the new Post route and examine the response headers and body to confirm that the record was inserted in the database:

```
$ curl -v -X POST http://localhost:9000/suppliers --header "Content-type: application/json" --data '{"name":"Ned Flanders", "contactNo":"555-1234"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /suppliers HTTP/1.1
> User-Agent: curl/7.37.1
```
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 47
>
* upload completely sent off: 47 out of 47 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 46
<
* Connection #0 to host localhost left intact
{"name":"Ned Flanders","contactNo":"555-1234"}

8. Request the listing route and verify that it is, in fact, returning records from the database:

```bash
$ curl -v http://localhost:9000/suppliers
* Hostname was NOT found in DNS cache
  * Trying ::1...
  * Connected to localhost (::1) port 9000 (#0)
  > GET /suppliers HTTP/1.1
  > User-Agent: curl/7.37.1
  > Host: localhost:9000
  > Accept: */*
  >
  < HTTP/1.1 200 OK
  < Content-Type: application/json; charset=utf-8
  < Content-Length: 110
  <
  * Connection #0 to host localhost left intact

[{{"id":1,"name":"Maud Flanders","contactNo":"555-1234"},{"id":2,"name":"Ned Flanders","contactNo":"712-1234"}]}
```

**How it works...**

In this recipe, we created a new URL route and action that will create and retrieve suppliers from the H2 database. We used Typesafe Slick as the relational mapping tool for creating queries and inserts. We started by declaring the required dependencies in `build.sbt`. Next, we defined the mapping properties for suppliers in `foo_scala/app/models/Supplier.scala`. 
In the mapping file, we declared our case class supplier. We also declared our Slick table mapping class. Lastly, we added our suppliers object class that should ideally contain all the required functions for data insertion and querying. We added the appropriate routes to the conf/routes file and ran the database evolution. This allows Slick to automatically manage table creation and column syncing. To test our implementation, we used curl to request our POST and GET endpoints to be able to view the response headers and body.

Utilizing play-mailer

For this recipe, we will explore how Play applications can send e-mails. We will use the Play module play-mailer to achieve this. We will be utilizing Mandrill, a cloud e-mailer service, to send out e-mails. For more information about Mandrill, please refer to https://mandrill.com/.

How to do it...

For Java, we need to take the following steps:

1. Run the foo_java application with Hot-Reloading enabled:
   activator "-run"

2. Declare play-mailer as a project dependency in build.sbt:
   "com.typesafe.play.plugins" %% "play-plugins-mailer" % "2.3.1"

3. Enable the play-mailer plugin by declaring it in foo_java/conf/play.plugins:
   1500:com.typesafe.plugin.CommonsMailerPlugin

4. Specify your smtp host information in foo_java/conf/application.conf:
   smtp.host=smtp.mandrillapp.com
   smtp.port=25
   smtp.user="YOUR OWN USER HERE"
   smtp.password="YOUR OWN PASSWORD HERE"
   smtp.mock=true

5. Modify foo_java/app/controllers/Application.java by adding the following code:
   import play.libs.F;
   import play.libs.F.Function;
   import play.libs.F.Promise;
   import com.typesafe.plugin.*/;

   public static Promise<Result> emailSender() {
     Promise<Boolean> emailResult = Promise.promise{
       new F.Function0<Boolean>() {
         public Boolean apply() {
           // Your email sending logic here
           return true; // Replace with actual logic
         }
       }
     }
@Override
public Boolean apply() throws Throwable {
    try {
        MailerAPI mail = play.Play.application().
            plugin(MailerPlugin.class).email();
        mail.setSubject("mailer");
        mail.setRecipient("ginduc@dynamicobjx.com");
        mail.setFrom("Play Cookbook <noreply@email.com>");
        mail.send("text");

        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
}

return emailResult.map(
    new Function<Boolean, Result>() {
        @Override
        public Result apply(Boolean sent) throws Throwable {
            if (sent) {
                return ok("Email sent!");
            } else {
                return ok("Email was not sent!");
            }
        }
    });

return emailResult.map(
    new Function<Boolean, Result>() {
        @Override
        public Result apply(Boolean sent) throws Throwable {
            if (sent) {
                return ok("Email sent!");
            } else {
                return ok("Email was not sent!");
            }
        }
    });

6. Add a new routes entry for the newly added action to foo_java/conf/routes:
   POST /send_email controllers.Application.emailSender

7. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

   $ curl -v -X POST http://localhost:9000/send_email
   * Hostname was NOT found in DNS cache
   * Trying ::1...
Leveraging Modules

* Connected to localhost (::1) port 9000 (#0)
  > POST /send_email HTTP/1.1
  > User-Agent: curl/7.37.1
  > Host: localhost:9000
  > Accept: */*
  >
  < HTTP/1.1 200 OK
  < Content-Type: text/plain; charset=utf-8
  < Content-Length: 11
  <

* Connection #0 to host localhost left intact
  Email sent!%

For Scala, we need to take the following steps:

1. Run the foo_scala application with Hot-Reloading enabled:
   activator "-run"

2. Declare play-mailer as a project dependency in build.sbt:
   "com.typesafe.play.plugins" %% "play-plugins-mailer" % "2.3.1"

3. Enable the play-mailer plugin by declaring it in foo_scala/conf/play.plugins:
   1500:com.typesafe.plugin.CommonsMailerPlugin

4. Specify your smtp host information in foo_scala/conf/application.conf:
   smtp.host=smtp.mandrillapp.com
   smtp.port=25
   smtp.user="YOUR OWN USER HERE"
   smtp.password="YOUR OWN PASSWORD HERE"
   smtp.mock=true

5. Modify foo_scala/app/controllers/Application.scala by adding the following action:

   import scala.concurrent._
   import com.typesafe.plugin._
   import play.api.libs.concurrent.Execution.Implicits._
   import play.api.Play.current

   def emailSender = Action.async {
     sendEmail.map { messageId =>
       Ok("Sent email with Message ID: " + messageId)
     }
   }
def sendEmail = Future {
    val mail = use[MailerPlugin].email

    mail.setSubject("Play mailer")
    mail.setRecipient("ginduc@dynamicobjx.com")
    mail.setFrom("Play Cookbook <noreply@email.com>")
    mail.send("text")
}

6. Add a new routes entry for the newly added action to foo_scala/conf/routes:
   POST /send_email    controllers.Application.emailSender

7. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

   $ curl -v -X POST http://localhost:9000/send_email
   * Hostname was NOT found in DNS cache
   *   Trying ::1...
   * Connected to localhost (::1) port 9000 (#0)
   > POST /send_email HTTP/1.1
   > User-Agent: curl/7.37.1
   > Host: localhost:9000
   > Accept: */*
   >
   < HTTP/1.1 200 OK
   < Content-Type: text/plain; charset=utf-8
   < Content-Length: 30
   <
   *
   Sent email with Message ID: ()% 

How it works...

In this recipe, we created a new URL route and action that will invoke our newly added sendMail function. We declared the module dependency in foo_scala/build.sbt and specified our smtp server settings in foo_scala/conf/application.conf. After this, we invoked the URL route using curl in the terminal to test out our e-mail sender. You should now receive the e-mail in your e-mail client software.
Integrating Bootstrap and WebJars

For this recipe, we will explore how we can integrate and utilize the popular frontend framework Bootstrap with a Play 2 web application. We will integrate Bootstrap using WebJars, which is a tool to package frontend libraries into JAR files that can then be easily managed (in our case, by sbt).

How to do it...

For Java, we need to take the following steps:

1. Run the foo_java application with Hot-Reloading enabled:
   
   ```bash
   activator ~run
   ```

2. Declare Bootstrap and WebJars as a project dependency in `build.sbt`:
   
   ```scala
   "org.webjars" % "bootstrap" % "3.3.1",
   "org.webjars" %% "webjars-play" % "2.3.0"
   ```

3. Modify `foo_java/app/controllers/Application.java` by adding the following code:
   
   ```java
   public static Result bootstrapped() {
      return ok(views.html.bootstrapped.render());
   }
   ```

4. Add the new route entries to `foo_java/conf/routes`:
   
   ```scala
   GET     /webjars/*file        controllers.WebJarAssets.at(file)
   GET     /bootstrapped         controllers.Application.bootstrapped
   ```

5. Create the new layout View template in `foo_java/app/views/mainLayout.scala.html` with the following content:
   
   ```html
   @title(String)(content: Html)<!DOCTYPE html>
   <html>
   <head>
   <meta charset="utf-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <meta name="description" content="">
   <meta name="author" content="">
   <title>@title</title>
   ```
6. Create the Bootstrapped View template in `foo_java/app/views/bootstrapped.scala.html` with the following content:

```scala
@mainLayout("Bootstrapped") {
  <div class="hero-unit">
    <h1>Hello, world!</h1>
    <p>This is a template for a simple marketing or informational website. It includes a large callout called the hero unit and three supporting pieces of content. Use it as a starting point to create something more unique.</p>
    <a href="#" class="btn btn-primary btn-large">Learn more &raquo;</a>
  </div>
}
```
7. Request our new Bootstrapped route (http://localhost:9000/bootstrapped) using a web browser and examine the rendered page using a Bootstrap template:

![Image of Bootstrap template](image.png)

**Hello, world!**

This is a template for a simple marketing or informational website. It includes a large callout called the hero unit and three supporting pieces of content. Use it as a starting point to create something more unique.

Learn more -

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:
   
   ```bash
   activator "-run"
   ```

2. Declare Bootstrap and WebJars as a project dependency in `build.sbt`:
   
   ```scala
   "org.webjars" %% "bootstrap" % "3.3.1",
   "org.webjars" %% "webjars-play" % "2.3.0"
   ```

3. Modify `foo_scala/app/controllers/Application.scala` by adding the following action:

   ```scala
   def bootstrapped = Action {
     Ok(views.html.bootstrapped())
   }
   ```
Leveraging Modules

4. Add the new route entries to foo-scala/conf/routes:

   GET       /webjars/*file        controllers.WebJarAssets.at(file)
   GET       /bootstrapped         controllers.Application.bootstrapped

5. Create the new layout view template in foo-scala/app/views/
   mainLayout.scala.html with the following content:

   ```html
   @(title: String)(content: Html)<!DOCTYPE html>
   <html>
   <head>
     <meta charset="utf-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-scale=1">
     <meta name="description" content="">
     <meta name="author" content="">
     <title>@title</title>
     <link rel="shortcut icon" type="image/png" href='@routes.Assets.at("images/favicon.png")'>
     <link rel="stylesheet" media="screen" href='@routes.WebJarAssets.at(WebJarAssets.locate("css/bootstrap.min.css"))' />
     <link rel="stylesheet" media="screen" href='@routes.Assets.at("stylesheets/app.css")'/>
     <style>
       body {
         padding-top: 50px;
       }
     </style>
   </head>
   <body>
   <nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">
     <div class="container-fluid">
       <div class="navbar-header">
         <button type="button" class="navbar-toggle collapsed"
         data-toggle="collapse" data-target="#navbar" aria-expanded="false"
         aria-controls="navbar">
           <span class="sr-only">Toggle navigation</span>
           <span class="icon-bar"></span>
           <span class="icon-bar"></span>
           <span class="icon-bar"></span>
         </button>
         <a class="navbar-brand" href="#">Admin</a>
       </div>
   ```
6. Create the bootstrapped View template in `foo_scala/app/views/bootstrapped.scala.html` with the following content:

```scala
@mainLayout("Bootstrapped") {
  <div class="hero-unit">
    <h1>Hello, world!</h1>
    <p>This is a template for a simple marketing or informational website. It includes a large callout called the hero unit and three supporting pieces of content. Use it as a starting point to create something more unique.</p>
    <a href="#" class="btn btn-primary btn-large">Learn more &raquo;</a>
  </div>
}
```

7. Request our new bootstrapped route (`http://localhost:9000/bootstrapped`) using a web browser and examine the rendered page using a Bootstrap template:

![Bootstrapped page screenshot](image)
How it works...

In this recipe, instead of downloading Bootstrap separately and managing different versions manually, we used the Play module and WebJars, declaring Bootstrap as a frontend dependency in `build.sbt`. We created the new View templates containing the Bootstrap template. We then created a new URL route that will utilize these new Bootstrap-based views.
Where to buy this book

You can buy Play Framework Cookbook Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.