(../..)

Create the Spring Boot application

During this lab, you are going to introduce Spring Boot to the Moviefun app step by step. The steps to do so will follow this pattern:

- Add Spring Boot and other dependencies to your build file
- Add a Spring Boot `Application` class
- Configure a datasource (move data configuration out of the `persistence.xml`)
- Create a controller that renders templates
- Move data access out of the view
- Configure servlets (and move configuration out of the `web.xml`)
- Clean up unused file

After each step you will check whether the app works or not. Very often it will still need fixes that the next steps will address.

Update the `pom.xml`

1. Remove the test dependency on `spring-web`
2. Set up the parent pom to include the `spring-boot-starter-parent`

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2.RELEASE</version>
</parent>
```

3. Register the `spring-boot-maven-plugin`

```
<build>
    <finalName>moviefun</finalName>
    <defaultGoal>package</defaultGoal>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

4. Add the following dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Use `search.maven.org` when you can't find the `groupId`

Create an application class

1. Create the application class in the moviefun package and annotate it with `@SpringBootApplication`.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

2. Now try and run the app (from IntelliJ a `run` button should be available or you can use `mvn spring-boot:run` from the commandline).

   Does your app work? It should not. Verify that it does not work by running the smoke tests again:

   ```
   MOVIE_FUN_URL=http://localhost:8080 mvn test
   ```

   What is the error?

Configure a datasource

For data to be persisted if there are multiple app instances or if you need to `cf restage`, you will need a database. Let's create one locally first.

1. Create a mysql database.

   ```
   mysql -uroot
   ```

   ```
   create database movies;
   ```

   If mysql is not responding, check that it is started. You can use the following command to get info on how to start the server:

   ```
   brew info mysql
   ```

2. Create the file `src/main/resources/application.yml` and add the following to it:

```
spring:
  jpa:
    generate-ddl: true
    properties.hibernate.dialect: org.hibernate.dialect.MySQL5Diale
ct
  datasource:
    url: jdbc:mysql://localhost:3306/movies?useSSL=false
    username: root
```

3. Now run the app. Did it start? Visit http://localhost:8080 (http://localhost:8080).

   Does the home page render?

4. Run the smoke tests:

   ```
   MOVIE_FUN_URL=http://localhost:8080 mvn test
   ```

   What is the error you got in the smoke tests?

Map the `index.jsp`

Right now, the `index.jsp` is not mapped appropriately to the route URL, so the homepage is not showing up. Let's fix it.

1. Configure Spring JSP rendering by adding the following to your `application.yml`:

   ```
   spring:
     mvc.view:
       prefix: /WEB-INF/
       suffix: .jsp
   ```

2. Move `index.jsp` into the `/webapp/WEB-INF` folder.

3. Create a controller class called `HomeController` and annotate it with `@Controller`.

4. In the controller create an action that renders the `index.jsp` (the controller action should return the String "index").

   Once you are done, run the smoke tests.

   ```
   MOVIE_FUN_URL=http://localhost:8080 mvn test
   ```

   What are the test failures?

Map the `setup.jsp`

Let's make the `setup` page work.

1. Add a method to the controller, map it to `/setup`, and make it render the `setup.jsp` file.
2. Change references to the `/setup.jsp` path (in the `index.jsp`) to `/setup`

   Run the app.

   Does it work? Check using the smoke tests. What error do you get?

Extract the data access

We now need to move data access from the `setup.jsp` into the controller. Any references to `MoviesBean` should be moved from the `setup.jsp` and put into the controller action.

- Make `MoviesBean` a field of your controller.
- Inject it through the constructor.
- Use it in the `setup` function to create the movies and fetch the movies.
- Assign the list of movies to the `movies` key of your model (at that point you need to add the model as an argument of your controller function)

If you run into issues with the code, scroll down to the `hints` section at the bottom of the lab to see one way to write this controller action.

Make `MoviesBean` injectable

For data access to work via the controller, we need to make the `MoviesBean` injectable.

- Replace `@Stateless` with `@Repository` on the `MoviesBean` class.
- Remove the `unitName` attribute from the `@PersistenceContext` annotation.

Run the application. Does it work? Check with the smoke tests.

What have the changes you enacted in this section accomplished?

Transforming `setup.jsp`

After removing references to `MoviesBean` in the JSP, you will need to change how movies are rendered.

Replace the old way of iterating over the collection in the `setup.jsp` file with the following:

```
<c:forEach items="${requestScope.movies}" var="movie">
  <tr>
    <td>${ movie.title }</td>
    <td>${ movie.director }</td>
    <td>${ movie.genre }</td>
  </tr>
</c:forEach>
```

Now move the `setup.jsp` file into the `WEB-INF` folder.

Make the controller action transactional

For the `MoviesBean#addMovie` function to work, we need to make the calling function transactional.

Add the `@Transactional` annotation to your controller function.

Try running the app again. Run the smoke tests.

At this point you should now have both `/` and `/setup` working in your application.

Update `SmokeTest.java`

Make sure the `setupPage` url is changed from `/setup.jsp` to `/setup`.

Register the Web Servlet

From your running app, try visiting the `/moviefun` endpoint. What happens?

For the `/moviefun` endpoint to work we need to register the web servlet. This was done in the `web.xml` file, but we should move that configuration to our new Spring Boot `Application` class.

1. We should declare a `ServletRegistrationBean` in our application configuration.
2. To do so, define a function in the `Application` class annotated with `@Bean`, returning the `ServletRegistrationBean`. Pass the `ActionServlet` to the function as an argument so that Spring instantiates it for you.
3. Annotate the `ActionServlet` class with `@Component` so it is available to your `ServletRegistrationBean`.
4. Note that the `ActionServlet` uses the `MoviesBean#addMovie` function. In order for this to work, you will need to move `@Transactional` from its previous location to the `addMovie` function of `MoviesBean`.

For an example of how we've written this function, scroll down to the hints at the bottom of this lab.

Try running the app. Run the smoke tests.

Everything should now work! (if it does not, read your error messages).

Wrap Up

We no longer need the `web.xml` and `persistence.xml` files. Feel free to delete them.

Push the app to Pivotal Cloud Foundry and run the smoke tests against it. You can now push with the java buildpack:

```
cf push moviefun -p target/moviefun.war -b https://github.com/cloudfou
ndry/java-buildpack.git
```

Check that `/setup` is working in your browser.

Run `cf restage moviefun`. Check that your data is persisted.

Assignment

Once you are done with the lab and the application is deployed and working on PCF, you can submit the assignment using the `submitReplatformingSpringBootification` gradle task. It requires you to provide the `movieFunUrl` project property. For example:

```
cd ~/workspace/assignment-submission
./gradlew submitReplatformingSpringBootification -PmovieFunUrl=http://
my-movie-fun.cfapps.io
```

Hints

Example `setup` Controller Action

In the `HomeController` class:

```
@GetMapping("/setup")
public String setup(Map<String, Object> model) {
    moviesBean.addMovie(new Movie("Wedding Crashers", "David Dobkin",
"Comedy", 7, 2005));
    moviesBean.addMovie(new Movie("Starsky & Hutch", "Todd Phillips",
"Action", 6, 2004));
    moviesBean.addMovie(new Movie("Shanghai Knights", "David Dobkin",
"Action", 6, 2003));
    moviesBean.addMovie(new Movie("I-Spy", "Betty Thomas", "Adventure"
, 5, 2002));
    moviesBean.addMovie(new Movie("The Royal Tenenbaums", "Wes Anderso
n", "Comedy", 8, 2001));
    moviesBean.addMovie(new Movie("Zoolander", "Ben Stiller", "Comedy"
, 6, 2001));
    moviesBean.addMovie(new Movie("Shanghai Noon", "Tom Dey", "Comedy"
, 7, 2000));

    model.put("movies", moviesBean.getMovies());
    return "setup";
}
```

Example `ServletRegistrationBean` Declaration

In the `Application` class:

```
@Bean
public ServletRegistrationBean servletRegistrationBean(ActionServlet a
ctionServlet){
    return new ServletRegistrationBean(actionServlet, "/moviefun/*");
}
```