(../..)

Scenario

In this lab, we will set up our MovieFun application to consume multiple datasources: one for movies (as before), and a new datasource for albums.

In order to do this, we will disable SpringBoot's auto-configuration of datasources (it works great with one datasource, but not with more).

Instead, we will supply our own database config.

This new config should enable us to continue using the J2EE `entityManager` used by the `AlbumsBean` and `MoviesBean` classes in our old app.

We will also need to use a `transactionManager` because Spring's `@Transactional` annotation will not work well with two databases.

Initial setup

In the movie fun application folder, checkout the `two-data-sources-start` tag.

```
git checkout two-data-sources-start
```

If you have work in progress from a previous lab, you can commit it to a branch if you need to save it:

```
git checkout -b my-branch
git commit -am "work in progress"
```

To restore to the original version of the tag:

```
git checkout two-data-sources-start
git reset --hard refs/tags/two-data-sources-start
git clean -df
```

Browse the code and notice the following changes: we now have an `AlbumsBean`, and the setup controller endpoint creates several `Album`s. Both the `AlbumsBean` and the `MoviesBean` use an `EntityManager`, thus they use the same database connection.

Deploy the application to PCF.

```
mvn clean package -DskipTests -Dmaven.test.skip=true
cf push moviefun -p target/moviefun.war
```

Check that the setup endpoint works as expected.

Now, let's setup the application so that the `MoviesBean` and the `AlbumsBean` use different databases.

Introducing multiple data sources

Create a new local MySQL database

```
mysql -uroot
create database albums;
```

Set up `application.yml`

Add the following to the `application.yml`.

```
moviefun:
  datasources:
    movies:
      url: jdbc:mysql://localhost:3306/movies?useSSL=false
      username: root
      password:
    albums:
      url: jdbc:mysql://localhost:3306/albums?useSSL=false
      username: root
      password:
```

Remove the existing `datasource` config and `jpa` config from `application.yml`.

Disable SpringBoot autoconfigurations for data access

SpringBoot auto-configuration is not designed for multiple datasources, so we will do the config ourselves.

Disable auto-configuration in your Application class as follows:

```
@SpringBootApplication(exclude = {
    DataSourceAutoConfiguration.class,
    HibernateJpaAutoConfiguration.class
})
```

Create DataSource configurations

Create a new `@Configuration` annotated class called `DbConfig` for your database configuration.

- Create a `@Bean` annotated `DataSource` for each database. For example:

```
@Bean
@ConfigurationProperties("moviefun.datasources.albums")
public DataSource albumsDataSource() {
    return DataSourceBuilder.create().build();
}
```

- You *could* use `@Value` to inject `url`, `username`, and `password` to configure each database instead of `@ConfigurationProperties`.

Create EntityManagerFactoryBeans configurations

To enable the `EntityManager` used by your AlbumsBean and MoviesBean classes, you will do the following:

- Declare a `@Bean` function for a `HibernateJpaVendorAdapter`. This will be used as an argument when you create your `EntityManager` with your custom configuration.
    - Set it up with `MYSQL` database type.
    - Set the platform to `"org.hibernate.dialect.MySQL5Dialect"`.
    - Enable DDL Generation.

- Declare a `@Bean` function for a `LocalContainerEntityManagerFactoryBean` for each database. The `LocalContainerEntityManagerFactoryBean` will be used to create your `EntityManagers`.
    - The function should take a data source as argument, and the jpa vendor adapter as well.
    - Set the data source
    - Set the Jpa Vendor adapter
    - Set the packages to scan to the current package (using `setPackagesToScan`)
    - Set a persistence unit name unique to each database

Set up the MoviesBean and AlbumsBean

Update the `@PersistenceContext` annotations in the `MoviesBean` and `AlbumsBean` classes with the persistence `unitName` you have chosen for each.

Now try and run the spring application.

Is the `/setup` endpoint working? What error do you see?

Create TransactionManagers configurations

To fix the `/setup` endpoint, we need to create two transaction managers, one for each database.

- Declare an `@Bean` function for a `PlatformTransactionManager` for each database
    - The function should take the matching `LocalContainerEntityManagerFactoryBean`
    - You may need to annotate the argument with `@Qualifier` to specify which one you want.
- Inject the transaction managers in the controller.
- Replace `@Transactional` with manual usage of the transaction managers **around** the creation of albums and movies in the `HomeController`.
- Remove `@Transactional` annotations in the `AlbumsBean`, and the `MoviesBean` classes.
- Recreate your local `albums` and `movies` databases (because you are now using a `JpaVendorAdapter` that is configured differently from how it was configured by Spring Boot when you ran the app earlier).

Run the app again, check that `/setup` is now working.

Deploy to Pivotal Cloud Foundry.

1. Create another MySQL service, `albums-mysql`, and bind it to the application.
2. Run `cf env moviefun` and note the `url`, `username`, and `password` for each bound MySQL service.

3. Using `cf set-env`, set the values from `Step 2` as environment variables on your app. The names for the variables should match what you created for `datasources` in your `application.yml`. For example:

```
cf set-env moviefun MOVIEFUN_DATASOURCES_MOVIES_URL [MOVIES-URL-FR
OM-STEP-2]
cf set-env moviefun MOVIEFUN_DATASOURCES_MOVIES_USERNAME [MOVIES-U
SERNAME-FROM-STEP-2]
cf set-env moviefun MOVIEFUN_DATASOURCES_MOVIES_PASSWORD [MOVIES-P
ASSWORD-FROM-STEP-2]
# ...
```

   Note how the syntax for CF environment variables, using caps and underscores, automatically maps to your Spring Configuration.

4. Deploy the `.war` file.

5. Check the app works and `/setup` is working correctly.

Add connection pooling to the data sources

- Add the HikariCP (https://github.com/brettwooldridge/HikariCP) dependency in your `pom.xml`.

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>2.5.1</version>
</dependency>
```

- Modify your `DataSource` `@Bean` functions to return a `HikariDataSource` by wrapping the existing DataSources inside a `HikariDataSource`.

  Depending on how you set up your DataSource functions initially, this step may also require you to change how you inject your datasource configuration properties.

- Check that the app still works locally.

- Deploy and ensure it still works on PCF.

Assignment

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

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

Cleanup

Before starting the next lab, unset the environment variables that were set in this lab.

You should also run `cf unbind-service moviefun albums-mysql`.