Lab 10 - Leveraging Spring Cloud Connectors for Service Binding

In this lab we'll bind our RESTful web service from Lab 9 to a MySQL database and leverage Spring Cloud Connectors to easily connect to it.

NOTE The completed code for this lab can be found at `$COURSE_HOME/day_01/session_03/lab_10/complete/cities`.

Using Spring Cloud Connectors

1. Change to the lab directory (the initial state for this lab is the same as the completed state for Lab 9, so you can choose to continue with that project if you like):

   ```
   $ cd $COURSE_HOME/day_01/session_03/lab_10/initial/cities
   ```

2. At present we're still using the in-memory database. Let's connect to a MySQL database service. From the CLI, let's *create* a MySQL service instance:

   ```bash
   $ cf cs p-mysql 100mb-dev cities-db
   Creating service cities-db...
   OK
   ```

3. Next add the service to your application manifest, which will *bind* the service to our application on the next push. We’ll also add an environment variable to switch on the “cloud” profile,

   ```yml
   ---
   applications:
   - name: cities
     memory: 512M
     instances: 1
     path: build/libs/cities-0.0.1-SNAPSHOT.jar
     timeout: 180
     services:                        # Add
     - cities-db                      # these
     env:                             # four
       SPRING_PROFILES_ACTIVE: cloud  # lines
   ```

   You can also accomplish the service binding by explicitly binding the service at the command-line:

   ```bash
   $ cf bind-service cities cities-db
   Binding service cities-db to app cities...
   OK
   ```

4. Next we’ll add Spring Cloud and MySQL dependencies to our Gradle build. Comment or remove the `hsqldb` line add add the following in the `dependencies` section:

   ```groovy
   dependencies {
       // ....
       compile("org.springframework.cloud:spring-cloud-spring-service-connector:1.1.0.RELEASE")
       compile("org.springframework.cloud:spring-cloud-cloudfoundry-connector:1.1.0.RELEASE")
       runtime("'org.mariadb.jdbc:mariadb-java-client:1.1.7'
   ")
   }
   ```

Since we've added new dependencies, re-run `./gradlew idea` or `./gradlew eclipse` to have them added to the IDE classpath.

5. Next, let's create the package `org.example.cities.config` and create in that package the class `CloudDataSourceConfig`. Add the following code:

   ```
   JAVA
   @Profile("cloud")
   @Configuration
   public class CloudDataSourceConfig extends AbstractCloudConfig {
       @Bean
       public DataSource dataSource() {
           return connectionFactory().dataSource();
       }
   }
   ```

   As before, have the IDE import the appropriate dependencies.

   The `@Profile` annotation will cause this class (which becomes Spring configuration when annotated as `@Configuration`) to be added to the configuration set because of the `SPRING_PROFILES_ACTIVE` environment variable we added earlier. You can still run the application locally (with the default profile) using the embedded database.

   With this code, Spring Cloud will detect a bound service that is compatible with `DataSource`, read the credentials, and then create a `DataSource` as appropriate (it will throw an exception otherwise).

6. Add the following to `src/main/resources/application.properties` to cause Hibernate to create the database schema and import data at startup. This is done automatically for embedded databases, not for custom `DataSource`s. Other Hibernate native properties can be set in a similar fashion:

   ```
   JAVA
   spring.jpa.hibernate.ddl-auto=create
   ```

7. Build the application:

   ```bash
   $ ./gradlew assemble
   ```

8. Re-push the application:

   ```bash
   $ cf push
   ```

9. Take a look at the `env` endpoint again to see the service bound in `VCAP_SERVICES`:

   ```bash
   $ curl http://cities-colorado-contemplator.nyc.fe.gopivotal.com/env
   ...
   "VCAP_SERVICES":"{\"p-mysql\":[{\"name\":\"cities-db\",\"label\":\"p-mysql\",\"tags\":
   [\"mysql\",\"relational\"],\"plan\":\"100mb-dev\",\"credentials\":
   {\"hostname\":\"10.68.106.85\",\"port\":3306,\"name\":\"cf_7a5601f9_32e3_41e1_b523_ed5ca96418f5\",\"username\
   ":\"Vm2Z9D848eagt4rq\",\"password\":\"JgstkF06p2SOTlZg\",\"uri\":\"mysql://Vm2Z9D848eagt4rq:JgstkF06p2SOTlZg@
   10.68.106.85:3306/cf_7a5601f9_32e3_41e1_b523_ed5ca96418f5?
   reconnect=true\",\"jdbcUrl\":\"jdbc:mysql://10.68.106.85:3306/cf_7a5601f9_32e3_41e1_b523_ed5ca96418f5?
   user=Vm2Z9D848eagt4rq&password=JgstkF06p2SOTlZg\"}}]}",
   ...
   ```

   The application is now running against a MySQL database.

Customizing the `DataSource`

1. You can customize the database connection that Spring Cloud creates with a few lines of code. Change the `dataSource` method in `CloudDataSourceConfig` to add some pooling and connection configuration:

```java
@Bean
public DataSource dataSource() {
    PooledServiceConnectorConfig.PoolConfig poolConfig =
            new PooledServiceConnectorConfig.PoolConfig(5, 200);

    DataSourceConfig.ConnectionConfig connectionConfig =
            new DataSourceConfig.ConnectionConfig("characterEncoding=UTF-8");
    DataSourceConfig serviceConfig = new DataSourceConfig(poolConfig, connectionConfig);

    return connectionFactory().dataSource("cities-db", serviceConfig);
}
```

2. Build the application:

```bash
$ ./gradlew assemble
```

3. Re-push the application:

```bash
$ cf push
```

Last updated 2015-06-14 20:19:26 EDT