Openshift 4 Workshop JavaEE8 (Part 2 of 2)

Background

Basics of Java Persistence

What is Java Persistence API?

The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform. Java Persistence API removes a lot of boilerplate code and helps bridge the differences between the object modules (used in Object-oriented languages like Java) and the relational model (used in relational databases like PostgreSQL®, MySQL®, Oracle® DB, etc.). JPA is commonly used in both Java EE and other frameworks like Spring Data.
The JPA implementation in JBoss EAP is called Hibernate, which is a Red Hat lead open source project. Hibernate is probably the most popular JPA implementation known for its stability and performance.

Enable JPA in JBoss EAP Java EE sometimes receives criticism for being too big, making application servers slow and requiring too much resources, but to be fair Java EE is only a specification of different API's and how they should work. How these specifications are implemented are left to the application server, and while there are application servers that are slow and requires a lot of resources, JBoss EAP is a modern application server based on a modular architecture using subsystems to implement Java EE technologies. Many of these modules are only started when an application makes use of them. So if you are only using parts of all the features in Java EE, JBoss EAP will not waste resources like memory and CPU with unused subsystems.

1. Create a persistence.xml. The way you enable JPA in JBoss EAP merely is by putting a persistence.xml file on the classpath. The persistence.xml file is anyway required by the JPA specification and holds the base configuration, for example, which datasource to use.

Our first step is to create a persistence.xml file in src/main/resources/META-INF.

Copy this in the persistence.xml

<?xml version="1.0" encoding="UTF-8"?>

```
<persistence version="2.1"
          xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="
       http://xmlns.jcp.org/xml/ns/persistence
       http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
   <persistence-unit name="primary">
       <jta-data-source>java:jboss/datasources/WeatherDS</jta-data-source>
       <properties>
          <!-- Properties for Hibernate -->
          <property name="hibernate.hbm2ddl.auto" value="create-drop" />
          <property name="hibernate.show_sql" value="true" />
       </properties>
   </persistence-unit>
</persistence>
```

Note that we are giving the persistence unit the name primary and instructing it to use the JTA datasource called **java:jboss/datasources/WeatherDS**. There are also two hibernate specific properties configured here. The first is **hibernate.hbm2ddl.auto** which is set to **create-drop**. This tells Hibernate to create the necessary database schema when the application is deployed and drop (.e.g. delete) the schema when it's undeployed. This assumes that we are starting from an empty database and will delete the content of the database when undeployed. This is of course not a setting we want to have in production. A common setting is to use verify, which will check that the database schema matches the object model and otherwise fail to deploy. However, for the convenience of this scenario, we will leave it at create-drop for now. The second property **hibernate.show_sql** and will configure hibernate to log all SQL queries in the JBoss EAP server.log.

2. **Create a datasource**

Next step is to create the datasource. Usually this is done by configuring JBoss EAP using web-console, CLI scripts, or similar. However, as you will see later in this scenario, we can do this in OpenShift by just specifying environment variables pointing to a database. At this stage, we haven't created a database yet, but for test and development purposes we can use an in-memory database like H2. We can define it as follows.

Open the src/main/webapp/WEB-INF/weather-ds.xml file and click **Copy To Editor** to insert add the following content to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources xmlns="http://www.jboss.org/ironjacamar/schema"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.jboss.org/ironjacamar/schema
http://docs.jboss.org/ironjacamar/schema/datasources_1_0.xsd">
   <!-- The datasource is bound into JNDI at this location. We reference
      this in META-INF/persistence.xml -->
   <datasource jndi-name="java:jboss/datasources/WeatherDS"
               pool-name="weather" enabled="true"
               use-java-context="true">

<connection-url>jdbc:h2:mem:weather;DB_CLOSE_ON_EXIT=FALSE;DB_CLOSE_DELAY=-1</connection-url>
      <driver>h2</driver>
      <security>
         <user-name>sa</user-name>
         <password>sa</password>
      </security>
   </datasource>
</datasources>
```

3. Add JPA annotation to the data model

JPA is a very non-intrusive framework, and all we need to do is to add a couple of annotation to the objects in the domain model.

Let's start with the City class. **Open src/main/java/com/redhat/example/weather/City.java**

First, **we need to tell JPA that this class is an Entity**, meaning that it needs to be able to persist to the database. We do this by adding the **@Entity** annotation. Do that manually aftert the comment //TODO: Add Entity annotation

```
@Entity
public class City{
```

Then, we need to indicate which field is to be considered the **primary key** for the object. We do this by adding annotation **@Id to the field private String id**.

```
@Id
private String id;
```

That's it! We are now ready to retrieve object of the type City. However, We also have a data model for Country that holds a list of Cities. Let's go ahead and annotate that as well.

Open **src/main/java/com/redhat/example/weather/Country.java**

Add @Entity annotation to the class definition.

```
@Entity
public class Country
```

Then add @Id annotation to public String id;

```
@Id
private String id;
```

Now, we also need to tell JPA that the private List<City> cities; are related to each other. The relationship is what DB's calls One-To-Many, meaning that one single Country object has zero, one or more related City objects. In JPA we can accomplish this by adding @OneToMany annotation to it.

```
@OneToMany(fetch = FetchType.EAGER)
private List<City> cities;
```

4. Update the REST Service with content from the database

To find or persist the object in the database JPA provides an API object called EntityManager. Getting an EntityManager is fairly simple since all we have to do is to inject it using @PersistenceContext.

Open src/main/java/com/redhat/example/weather/WeatherService.java

At the TODO comment inject the entity manager like this:

```
@PersistenceContext(unitName = "primary")
EntityManager em;
```

Note, that we also set the unitName to "primary". That is actually not necessary since we only have a single persistence unit, JPA would use that one, but as a best practice it's always good to refer to named persistence units.

Now we can use the EntityManager to collect Country object. There are many different API calls we can do including creating our own query, however the fastest and easiest way is to use a method called find(Class returnType,Object key) like this:

```
public Country find(Class returnType,Object key) {
   return em.find(Country.class,"en");
   /* TODO: Replace with dynamic call to get selected language*/
}
```

At the moment the country id is hardcoded as "en", but in the next scenario we will learn how to use CDI to let the user select which country they want to display

Deploy the updates
We are now ready to test our application in OpenShift.
First, build the application and verify that we do not have any compilation issues.

mvn clean package

This will produce a WAR file called ROOT.war under the target directory.
Next, build a container by starting an OpenShift S2I build and provide the WAR file as input.

oc start-build weather-app --from-file=target/ROOT.war --wait

When the build has finished, you can test the REST endpoint directly using for example curl.

curl -i -H "Accept:application/json" <Weather Route>/api/weather

Note: that it might take a couple of seconds for the application to start so if the command fails at first, please try again.

There is more, but We will not have time for this, such as adding an external database, see -> https://learn.openshift.com/middleware/middleware-javaee8/