Lab 02: Create an Application

In this lab, you'll begin creating your own version of the application featured in Lab 1. This includes creating a Maven Spring Boot project and supporting GET REST calls. Test your work locally as you develop. When you're finished, deploy your project to the Oracle Application Container Cloud Service (OACCS).

Lab 2-1: Setup Required Software

Before you can build and deploy an application, we need to install Maven on Windows. The following steps show how to install the required software.

1. Download Maven (tar.gz or zip) from http://maven.apache.org/download.cgi.
2. Unzip the distribution archive to the directory in which you want to install Maven. For example, `C:\Maven\apache-maven-3.3.9\`
3. Add the bin directory of the created directory `apache-maven-3.3.9` to the `PATH` environment variable.
4. Open a Git terminal and run the command `mvn --version` to verify that Maven is correctly installed.

Lab 2-2: Set up your Project

The first step in creating an application is to set up a development space for your project.

Setup a Cloud Repository (Optional)

If you have an Oracle Cloud account, set up a cloud Git repository. A demo of these steps is provided if you do not have a cloud account.

1. Create a Developer Cloud Service project named `SpringProject`.
2. Create a Git repository in Developer Cloud Service named `SpringRepo`.
3. Copy the Git repository URL.
4. Find a local folder where you want your repo and project to be. For example: `C:\labs`.
5. Open a Git terminal on this folder and create a clone of the cloud repository. The folder `SpringRepo` should appear. This folder will also be the root directory of your project.

You're now ready to start building out your project locally.

Set up your Maven Spring Boot Project

To build out the Maven project locally, you'll need to create the directory structure and add a couple special files.

1. Within `SpringRepo`, create an EmployeeRESTApp directory (this is the project directory). Under that directory, create the following directory structure:

```
EmployeeRESTApp
└───src
    ├───main
    │   ├───java
    │   │   └───com
    │   │       └───example
    │   │           └───rest
    │   └───resources
    └───test
        └───java
            └───com
                └───example
                    └───rest
```

2. Place `pom.xml`, `deploy.xml`, and `manifest.json` in the project root directory. These files are available from the Lesson 2 page of the MOOC.

   a. `pom.xml` is slightly different from the default version provided on the Spring.io website. The Maven `exec` plugin has been added so that you can use `exec:java` to execute your application from the command line. At the top of the file, the `artifactId` and `version` values are updated to reflect the assignment.

   b. The `deploy.xml` file specifies what files should be combined to deploy your application to the Oracle Application Container Cloud Service. A deployable application archive is built automatically.

   c. The `manifest.json` file specifies metadata about your application. Your application can't be deployed without it. Packaging occurs later in this lab.

Your Maven Spring Boot project is now set up! However, there are no classes in the project. You'll begin creating these next.

Lab 2-3: Create a Simple Hello World Spring Boot Application

Adding a minimal amount of functionality to your program is an easy way to make your application testable. This lets you verify the application is set up properly and provides a stable foundation for more-complex functionality to be written later.

1. Open your Maven project in NetBeans or the IDE of your choice.
2. Create a new Java class named `App.java` in the `com.example.rest` package.
3. Replace the empty class with the following code:

```
package com.example.rest;

import java.util.Properties;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class App {

    public static final Properties myProps = new Properties();

    public static void main(String[] args) {
        // Set properties

        myProps.setProperty("server.address", "localhost");
        myProps.setProperty("server.port", "8080");

        SpringApplication app = new SpringApplication(App.class);
        app.setDefaultProperties(myProps);
        app.run(args);

    }
}
```

Note: This starts a Spring Boot application that listens locally on port 8080. You now have a running Spring Boot application, but no content for it, because we haven't added any REST controllers. This is done later. In the meantime, you can add a static webpage for testing purposes.

4. In your IDE, click the Files tab.
5. Navigate to your project's `src\main\resources` directory.
6. Create a subdirectory named `public`.
7. In the `public` directory, create an `index.html` file.
8. Put the following text into that file:

```
<html>
<head>
<title>Hello Home Page</title>
</head>
<body>
<h2>Hello Home Page</h2>
<p>Hello World!</p>
</body>
</html>
```

9. Save your work.

Your application now has the minimal functionality it needs to be testable.

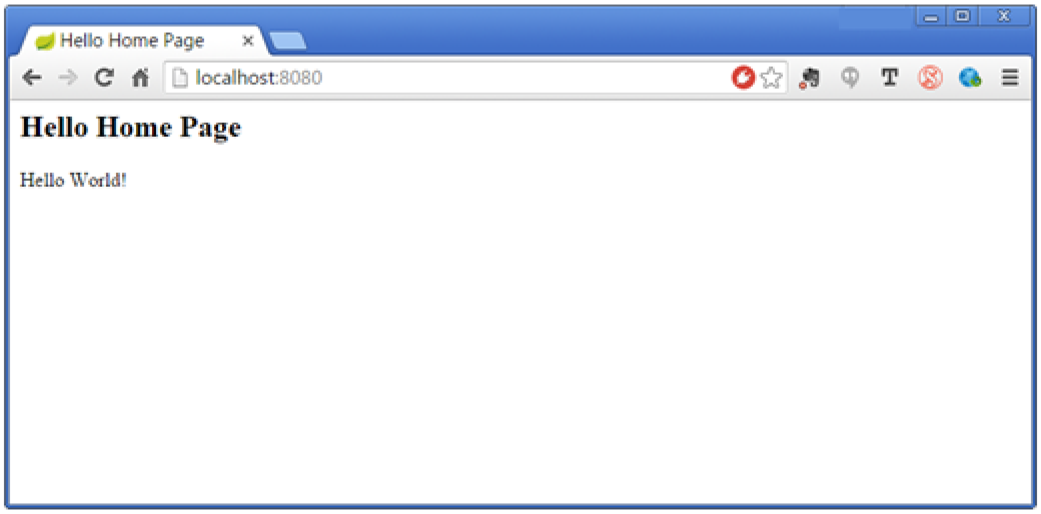
Lab 2-4: Test your Application Locally

It's preferable to test your application locally as you develop, rather than to upload and test on Oracle Application Container Cloud Service. Local testing is faster and more convenient for a developer. It also avoids exposing potentially buggy code to others, or drawing resources away from production processes.

1. Open a command prompt on your application's root directory. You'll need to test your application with a command prompt window because you cannot connect to servers started from Git Bash.
2. Compile your project: `mvn clean compile`

This creates the `target` folder in the project's root directory. You'll notice this folder doesn't contain a Jar. That file gets created through a different command, shown later.

3. Run your project: `mvn exec:java`
4. Open a browser and connect to http://localhost:8080. If you see a window that looks like the following screenshot, then your Spring Boot application is now running!



Hello Home Page

Hello World!

5. Press `Ctrl + C` in the command prompt to stop your program.
6. Alternatively, test your work by typing: `mvn clean package`

   This packages the final deliverables in the `target` folder, including a Jar and Zip file. You'll need the Zip later in this lab.
7. Run the newly created Jar:

   ```
   java -jar target/EmployeeRestApp-1.0.jar
   ```
8. Test your application again. Open a browser and connect to http://localhost:8080.

Lab 2-5: Add GET Methods to the Spring Boot REST Application

A lot more work is needed to build out your application. We've provided a lot of this code for you. We want you to focus on writing RESTful microservices, and not on writing object classes or lambda expressions. A couple completed GET methods are included. Study these methods to figure out how to write your own GET methods in the `EmployeeController` class.

1. Download `lab2-5.zip` from the Lesson 2 page. Incorporate the resources into their respective folders.
   a. The `Source Packages` folder contains six files, including an updated version of `App.java`. Insert these in your project's `src/main/java/com/example/rest` folder.
   b. The `Test Packages` folder contains `AppTest.java`. Insert this in you project's `src/test/java/com/example/rest` folder.
   c. The Other Sources folder contains four files, including an updated version of `index.html`. Insert these in your project's `src/main/resources/public` folder.
2. You may want to review the `Source Packages` code as you complete the next step. As you review, you might make the following observations:
   a. `Employee` fields are `final`. Your data will be deployed to a Tomcat server, which is inherently multithreaded. That means multiple clients might try to change your data at the same time. To prevent anything weird from happening, the data needs to be immutable. Immutable

means read-only. By setting variables to `final`, once initialized, the field values cannot be changed. Thus the data is read-only. Read-only data is inherently thread-safe.

b. `MockEmployeeList` uses `CopyOnWriteArrayList` instead of `ArrayList`. `CopyOnWriteArrayList` is a thread-safe implementation for `ArrayList`. As stated before, because your application will run in a multi-threaded environment, a thread-safe data structure is required.

3. In your IDE, open `EmployeeController.java` and create the following:

   a. A `GET` method handler for a `getByLastName` method. It returns all matching employees whose last name is contained in the specified in the URL. For example: `http://localhost:8080/employees/lastname/Jast` or `http://localhost:8080/employees/lastname/J`

   b. A `GET` method handler for a `getByTitle` method. It returns all matching employees whose title is contained in the URL. For example: `http://localhost:8080/employees/title/National Data Strategist`

   c. A `GET` method handler for a `getByDept` method. It returns all matching employees whose department is contained in the URL. For example: `http://localhost:8080/employees/department/Mobility`

   In all cases, if the employee does not exist, return a "404 not found" error message.

Note: The `@RestController` annotation identifies the class as RESTful. Spring Boot looks for `@RequestMapping` annotated methods as request handlers for the application. Spring Boot uses the `ResponseEntity` class to set HTTP response codes. The class uses an unbounded generic `<?>` so that any class may be used with it.

4. Test your code locally. The search criteria are case-sensitive. When you're satisfied, move on to the next part of the lab.

Lab 2-6: Commit your Work to Git (Optional)

If you have an Oracle Cloud account, you can commit your source code to your cloud repository. If you do not, a demo is provided showing each of the steps.

1. The `target` folder contains files that are auto-generated. There's no need to commit these files. Create a `.gitignore` file which contains a single line:

   ```
   **/target/**
   ```

   This tells Git to ignore any `target` directory in your repository. We do not need to version the application binaries.

2. Open a Git terminal on `SpringRepo`.

3. Set global name and email. Setting these values avoids a warning the first time you do a commit.

   ```
   a.git config --global user.name "Your Name"
   b.git config --global user.email you@example.com
   ```

4. Add your files to the repository: `git add .`

5. Ensure that mostly source code, and not auto-generated files from the `target` folder, will be committed.
    a. Check which files are marked to commit: `git status`
    b. If necessary, remove everything from the current `add` queue: `git reset`
6. Commit your work:

   ```
   git commit -m 'Repo setup for project'
   ```
7. Push your updates to the cloud: `git push origin master`

Your source code is now successfully backed up on the Developer Cloud Service repository.

Lab 2-7: Deploy and Test your Application in Oracle Application Container Cloud Service (Optional)

If you have a cloud account, you'll typically deploy your application to Oracle Application Container Cloud Service whenever you want to publish a major software update. The steps are summarized below. A demo of these steps is provided as well.

1. Make your application deployable on Oracle Application Container Cloud as described in the lecture. This will require some changes to `App.java`.
2. Build an up to date application archive by typing `mvn clean package`. This produces two versions of the application archive in the `target` folder: a `.zip` and `.tag.gz`.
3. Upload either the `.zip` or `.tag.gz` to Oracle Application Container Cloud Service.

4. Test your application in Oracle Application Container Cloud Service.