## **Section-End Project - Lesson 4 - Spring Boot Application**

* In this exercise you will create a simple Spring Boot application. All other exercises in this course will assume you know Spring Boot.

****Part 1 - Simple Web Application****

1. Using either Spring Tool Suite, or [https://start.spring.io](https://start.spring.io/), create a new Spring Boot Project.

* Use either Maven or Gradle (if you have it installed). All lab instructions are based on Maven.
* Use the latest release of Spring Boot and specifically Java 11.
* Use JAR packaging for now, unless you prefer WAR and already have a local server (like Tomcat) installed and ready to run it.
* Use any values you like for group, artifact, package, description, etc.
* Select the following dependencies: Web, Thymeleaf, JPA, HSQLDB, and Actuator.

1. Create a new Controller in the base package:

* Name the controller anything you like.
* Annotate the Controller with @Controller.

1. Create a new method in the controller:

* Name the method anything you like. Have it return a String. No parameters needed.
* Annotate the method with @GetMapping("/")
* Have the method return the String value "hello".

4. If not already present, create a new folder under src/main/resources called "templates"

5. Create a new file in the templates folder called "hello.html". Place the words "Hello from Thymeleaf" (or any markup you like) within the file.

6. Save all your work. Run your application.

* If you are working in Spring Tool Suite, simply right-click on the application / Run As / Spring Boot App or run it from the Boot Dashboard..
* If you are working in another IDE, Run the main method found within the main Application class.
* If you wish to run from a command prompt, from the application's root folder run mvn spring-boot:run.

1. Open a browser to <http://localhost:8080/>. You should see your web page.

****Part 2 - Return a RESTful Response****

1. Create a new Java class called "Team" in the base package. Give it a Long field for id, and String fields for name, location, and mascot (or whatever other properties you like). Generate "getters and setters" for all fields. Save your work.
2. Create a new Controller called "TeamController". Annotate it with @RestController.
3. Create a new method in the TeamController.

* Name the method "getTeams". Have it return a List of Team objects.
* Annotate the method with @GetMapping("/teams")
* Have the method create a List of Team objects. Create one or more Team objects and add it to the list. Set the teams' properties to whatever values you like, and return the List. Sample:

@GetMapping("/teams")

public List<Team> getTeams() {

List<Team> list = new ArrayList<>();

Team team = new Team();

team.setId(0l);

team.setLocation("Harlem");

team.setName("Globetrotters");

list.add(team);

team = new Team();

team.setId(1l);

team.setLocation("Washington");

team.setName("Generals");

list.add(team);

return list;

}

1. Save all work. Stop the application if it is already running, and start it again. Open <http://localhost:8080/teams>. You should see a JSON response with your teams' data.

****Part 3 - Create Spring Data JPA Repositories****

1. Return to the Team class. Add required JPA annotations: The class must be annotated with @Entity, the id must be annotated with @Id and @GeneratedValue.
2. Create a new Interface called "TeamRepository". Have it extend CrudRepository<Team,Long>.

* Be sure to create this as an Interface, not a Class!

1. Open the application's main configuration / launch class (the one annotated with @SpringBootApplication). Use @Autowired to dependency inject a member variable of type TeamRepository. Name the variable anything you like (may I suggest: "teamRepository").
2. Add some logic to initially populate the database: Add a public void init() method. Annotate it with @PostConstruct. Cut and paste the team-creation code from you controller to this method, except rather than returning a List of Teams, call the saveAll() method on the repository. Also, remove any values set for the team IDs. Example code:

public void init() {

List<Team> list = new ArrayList<>();

Team team = new Team();

team.setLocation("Harlem");

team.setName("Globetrotters");

list.add(team);

team = new Team();

team.setLocation("Washington");

team.setName("Generals");

list.add(team);

teamRepository.saveAll(list);

}

1. Return to the TeamController. Use @Autowired to dependency inject a TeamRepository variable. Name the variable anything you like (may I suggest: "teamRepository").
2. Alter the logic in your controller method to simply return the result of the repository's findAll() method:

@GetMapping("/teams")

public Iterable<Team> getTeams() {

return teamRepository.findAll();

}

1. Add this property to application.properties to control how the JPA implementation (Hibernate) should handle the DB schema:

spring.jpa.hibernate.ddl-auto = update

1. Save all work. Stop the application if it is already running, and start it again. Open <http://localhost:8080/teams>. You should see a JSON response with your teams' data.

****Part 4 - Create a Single Team endpoint****

1. Return to the TeamController and add a method that returns a single Team given an ID.

* Name the method anything you like. Suggestion: getTeam.
* Return type should be a Team.
* Use a @GetMapping annotation to map this method to the "/teams/{id}" pattern.
* Define a parameter named "id" of type Long annotated with @PathVariable.
* Logic: return the result of the teamRepository's findById(id).get() method. (The findById() returns a Java 8 "Optional", and the get() simply returns the actual Team.

1. Save all work. Stop the application if it is already running, and start it again. Use <http://localhost:8080/teams> to note the generated ID values for each Team. Then use URLs such as <http://localhost:8080/teams/1> or <http://localhost:8080/teams/2> to get results for the individual teams.

****Part 5 - Add Players****

1. Add a new class named Player. Add fields for id, name, and position. The id should be Long, and other fields can be Strings. Generate getters / setters for each field. Add an @Entity annotation on the class, and @Id and @GeneratedValue annotations on the id. You may wish to add a custom constructor to make it easy to create a Player object by supplying name and position Strings. (If you do this, be sure to retain a default constructor). Save your work.
2. Open the Team class. Add a Set of Player objects named players. Generate getters and setters. Annotate the set with @OneToMany(cascade=CascadeType.ALL) and @JoinColumn(name="teamId"). You may wish to add a custom constructor to make it easy to create a Team object by supplying name, location, and Set of Players. (If you do this, be sure to retain a default constructor). Save your work.
3. Return to application's main configuration / launch class and alter the team population logic to add some players to each team. Here is an example implementation:

@PostConstruct

public void init() {

List<Team> list = new ArrayList<>();

Set<Player> set = new HashSet<>();

set.add(new Player("Big Easy", "Showman"));

set.add(new Player("Buckets", "Guard"));

set.add(new Player("Dizzy", "Guard"));

list.add(new Team("Harlem", "Globetrotters", set));

list.add(new Team("Washington","Generals",null));

teamRepository.saveAll(list);

}

1. Save all work. Restart the application. Open <http://localhost:8080/teams> to see the players.

****Part 6 - Add XML Support****

1. When possible, REST applications should return the content type requested by the client. Spring Boot can easily support formats like XML based on dependencies (JAXB 2, Jackson) on the classpath. Unfortunately, versions of Java after 8.x no longer automatically include JAXB 2, so XML support requires a bit of effort on our part by adding either JAXB 2 OR Jackson dependencies. Open your POM and add dependencies for EITHER Jackson (recommended):

<dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</dependency>

OR JAXB 2:

<dependency>

<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

</dependency>

<dependency>

<groupId>org.glassfish.jaxb</groupId>

<artifactId>jaxb-runtime</artifactId>

<version>2.3.0</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>javax.activation</groupId>

<artifactId>javax.activation-api</artifactId>

<version>1.2.0</version>

</dependency>

1. If using JAXB2, you must also add an @XmlRootElement annotation to your Teams class.
2. Add these properties to your application.properties file to allow control of the desired format through either an extension/suffix or query parameter:

# URL ending with ?format=xml can specify requested media type:

spring.mvc.contentnegotiation.favor-parameter=true

# URL ending with .xml or .json can specify the requested media type:

spring.mvc.contentnegotiation.favor-path-extension=true

spring.mvc.pathmatch.use-suffix-pattern=true

1. Save all work. Restart the application. Open <http://localhost:8080/teams/1?format=xml> or <http://localhost:8080/teams/1.xml> to see the results in XML. Use <http://localhost:8080/teams/1?format=json> or <http://localhost:8080/teams/1.json> to see the results in JSON.

* Note that not all browsers display JSON & XML nicely, consider shopping for some plugins.
* Note that JAXB 2 will have issues with the /teams endpoint, since the type returned is not annotated with @XmlRootElement, one of the reasons many prefer Jackson.
* Note that the Accept header (Accept=text/xml or Accept=application/json) can also be used to specify the desired content type - it's just a bit more work to set from the browser. Use of an extension basis for content negotiation (i.e. .xml, .json) has fallen out of favor recently, see [suffix-based parameter matching](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-ann-requestmapping-suffix-pattern-match).

****Part 7 - Add Spring Data REST****

1. Open the project's POM. Add a dependency for group org.springframework.boot and artifact spring-boot-starter-data-rest. Save your work.
2. Open TeamRepository. Add a @RestResource(path="teams", rel="team") annotation to the interface.
3. Create a new Interface called "PlayerRepository". Have it extend CrudRepository<Player,Long>. (Be sure to create this as an Interface, not a Class)! Add a @RestResource(path="players", rel="player") annotation to the interface.
4. Open TeamController. Comment out the @RestController annotation on the class. (We will be using Spring Data REST to implement the controller, so we don't want this existing controller to interfere).
5. Save all work. Restart the application. Open <http://localhost:8080/teams> to see the players. Note that (depending on the browser you are using) you can navigate the links for players and teams.

* Note that Spring Data REST does not presently support XML, so the format=xml or .xml will have no effect (unfortunately).

If you have reached this point, Congratulations, you have finished the exercise!

****Part 8 - Explore the Actuator Endpoints****

1. One of the dependencies we specified was Actuator. It automatically adds some useful endpoints to our web application. Open the following with a browser:

* [/actuator/info](http://localhost:8080/actuator/info)
* [/actuator/health](http://localhost:8080/actuator/health)

1. Notice that some other actuator endpoints are not enabled by default. Try the following - they won't work, but take a close look at the reason why - exposing these could be a security risk:

* [/actuator/beans](http://localhost:8080/actuator/beans)
* [/actuator/configprops](http://localhost:8080/actuator/configprops)
* [/actuator/autoconfig](http://localhost:8080/actuator/env)

1. Enable these actuator endpoints by adding the following setting to your application.properties (save your work and restart):

* management.endpoints.web.exposure.include=beans,configprops,mappings,env

1. Explore [/actuator/mappings](http://localhost:8080/actuator/mappings). This is a useful one for debugging web applications. Search through and see where the @GetMappings you set earlier are setup.

****Part 9 - DevTools****

1. Often while developing we need to run an application, then make some changes, then restart. The Spring Boot "DevTools" dependency can make things easier by automatically restarting when changes are detected (though it is not as full-featured as other tools like JRebel). Add the following dependency:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

1. While your application is running, make a small, innocent change to some code (like a comment or spacing). Observe that depending on the change, DevTools will restart your application.

Tips:

* When running in Eclipse or STS, you can get automatic hot-swapping of many code changes without DevTools if you "debug" the application rather than "run" it.
* If you encounter unusual issues in the upcoming labs, it may be useful to remove the DevTools dependency to see if the problem persists.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*