## **Lab 2 - Spring Boot**

* In this exercise you will create a simple Spring Boot application. If you feel you already have a good understanding of Spring Boot you can skip this exercise. 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 stable releases of Boot and Java. These instructions were originally tested with Java 1.8, Boot 1.2.3.
   * 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, JDBC, JPA, HSQLDB, Actuator**.
2. Create a new Controller in the base package:
   * Name the controller anything you like.
   * Annotate the Controller with @Controller.
3. 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 **@RequestMapping("/")**
   * 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.
   * 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.**
7. 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 mascotte (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 @RequestMapping("/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:
4. @RequestMapping("/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;  
    }
5. 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.
   * Be sure to create this as an Interface, not a Class!
3. 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").
4. 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 save() method on the repository. Also, remove any values set for the team IDs. Example code:
5. 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.save(list);  
    }
6. Return to the TeamController. Use @Autowired to dependency inject a TeamRepository variable. Name the variable anything you like (may I suggest: "teamRepository").
7. Alter the logic in your controller method to simply return the result of the repository's findAll() method:
8. @RequestMapping("/teams")  
    public Iterable<Team> getTeams() {  
    return teamRepository.findAll();  
    }
9. 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 (Optional)- 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 @RequestMapping 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 findOne() method.
2. 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.
4. Save all work. Restart the application. Open <http://localhost:8080/teams> to see the players.

**Part 6 - 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. (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.
6. If you have reached this point, Congratulations, you have finished the exercise!

**Part 7 (Optional) - 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:
   * [/info](http://localhost:8080/info)
   * [/beans](http://localhost:8080/beans)
   * [/configprops](http://localhost:8080/configprops)
   * [/autoconfig](http://localhost:8080/autoconfig)
   * [/health](http://localhost:8080/health)
2. Explore [/mappings](http://localhost:8080/mappings). Does it show you any other useful endpoints you would like to try?

Tips: When running in Eclipse or STS, you can get automatic hot-swapping of many code changes if you "debug" the application rather than "run" it.