**Hands On Lab 1**

**Build a Spring Boot REST API with Java**

[Introduction 1](#_Toc5094)

[Requirements 2](#_Toc1878)

[Initializing a Spring Boot Project 2](#_Toc27806)

[Using Spring Initializr 2](#_Toc30062)

[Using Spring CLI 4](#_Toc2735)

[Connecting Spring Boot to the Database 5](#_Toc28365)

[Create the PostgreSQL database 6](#_Toc32258)

[Domain Model - Creating a User Model 7](#_Toc26661)

[Persistence Layer - Creating Repository Classes 8](#_Toc17193)

[Business Layer - Creating a Controller 9](#_Toc962)

[Compile, Build, and Run 12](#_Toc13116)

[Testing the APIs 13](#_Toc24176)

[Conclusion 15](#_Toc18332)

# Introduction

****REST**** stands for REpresentational State Transfer, a standardized approach to building web services.

A ****REST API**** is an intermediary Application Programming Interface that enables two applications to communicate with each other over HTTP, much like how servers communicate to browsers.

RESTful is the most common approach for building web services because of how easy it is to learn and build.

Let's say you order something from a fast-food restaurant and the cashier requests the information needed from you to process your order. After it's been processed, they hand you the order you requested for. This transaction is a real-life example of how REST API works.

In this tutorial, we'll go over how to build a REST API in Java with Spring Boot. It'll accept POST and GET request payloads to view and add entries from an entity - User.

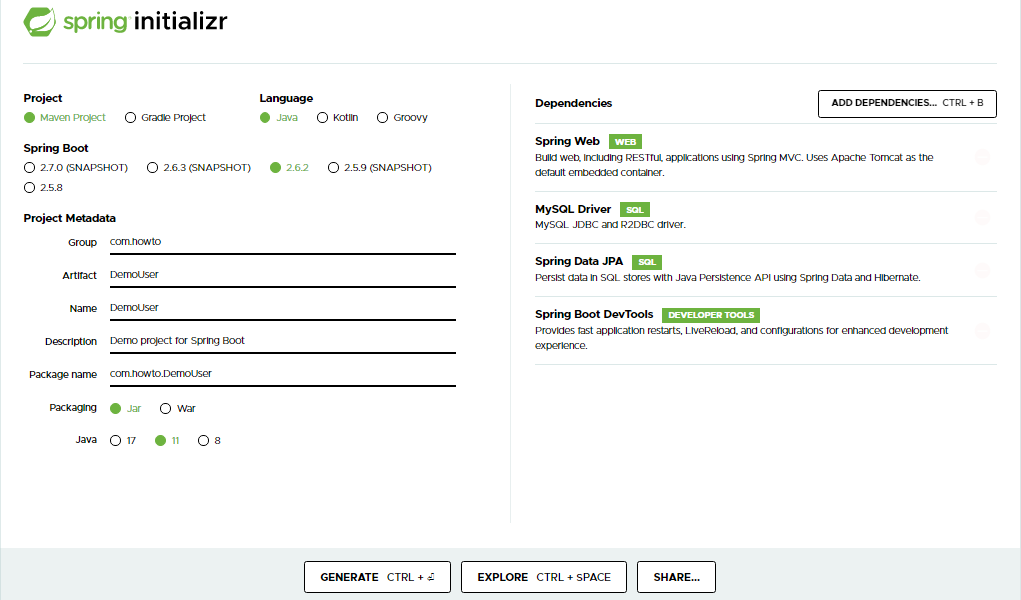
### **Requirements**

* IDE or text editor
* [JDK 1](https://www.oracle.com/ph/java/technologies/javase/javase-jdk8-downloads.html" \t "https://stackabuse.com/build-a-spring-boot-rest-api-with-java-full-guide/_blank)1
* [Maven](https://archive.apache.org/dist/maven/maven-3/" \t "https://stackabuse.com/build-a-spring-boot-rest-api-with-java-full-guide/_blank) 3+

# Initializing a Spring Boot Project

### **Using Spring Initializr**

One easy way you can initialize a new Spring Boot project is by using [Spring Initializr](https://start.spring.io/" \t "https://stackabuse.com/build-a-spring-boot-rest-api-with-java-full-guide/_blank), which automatically generates a skeleton Spring Boot project for you:



We'll add a few dependencies here as well, as we'll want to use them in our project:

* ****Spring Web**** - To include Spring MVC and embedded Tomcat into your project
* ****Spring Data JPA**** - Java Persistence API and Hibernate
* ****Spring Boot DevTools**** - Very useful development tools
* ****PostgreSQL Driver**** - JDBC Driver (Can be any DB you'd like to use)

Afterward, press generate. A zip file that contains the generated project will then be downloaded.

### **Using Spring CLI**

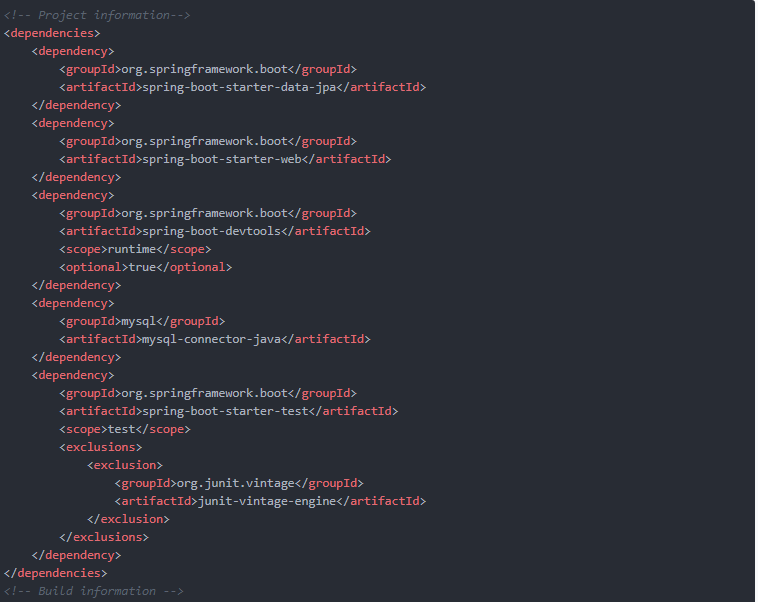
If you have [Spring CLI](https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html" \l "getting-started-installing-the-cli" \t "https://stackabuse.com/build-a-spring-boot-rest-api-with-java-full-guide/_blank) installed, then you can opt for using the console to build your base project using this command:

spring init --build=maven -p=jar UserDemo

****Note:**** Spring CLI directly calls Spring Initializr to perform this operation. Both options will produce the same project.

After building your base project, download and import it to your IDE or text editor of choice. If you want to build, compile, and run your application through your IDE, make sure you import it as a Maven  project.

Upon importing, the generated base pom.xml in your project will look like this:



\*\*\*Note: The MySQL driver above will be replaced with the PostgreSql driver you configured.

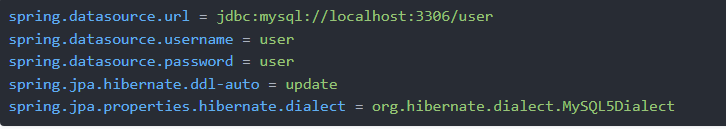
All the configurations that you did will be reflected in this file. On top of that, default dependencies, your base snapshot 0.0.1-SNAPSHOT, and the Maven build plugin are also automatically configured.

# Connecting Spring Boot to the Database

Next, before we start working on the application, we'll want to set up the database. This can easily be done through Spring Data JPA, which allows us to set this connection up with just a couple of parameters.

It abstracts away everything needed to be done by the developer, and allows us to switch underlying databases if we'd like, just by changing a couple of properties.

To tell Spring how to connect to your preferred database, in your application.properties file, you'll need to add some rudimentary information:



\*\*\*Note: You need to update the above settings for PostgreSQL database as done in the training sessions.

Here, we've set the datasource.url to our JDBC connection URL. This depends on your database. We've provided the username and password required to authenticate into that database, as well as set the ddl-auto property to update. Change username and password to suit your environment.

The jpa.hibernate.ddl-auto property directly influences the hibernate.hbm2ddl.auto property, and essentially defines how Hibernate should handle schema tool management.

For production applications, this value is typically set to none, as dedicated personnel conduct management. In development, it's most common to use update, to allow the schema to be updated each time you restart the application, allowing you flexibility while working on development.

Finally, we've set the hibernate.dialect property. Hibernate has different dialects for different databases. It can automatically set the dialect based on your configuration, though, for additional safety, it's always a good idea to specify it explicitly.

# Create the PostgreSQL database

Connect to PostgreSQL pgadmin4 and create a database called “user”:

Tables will initially be empty.

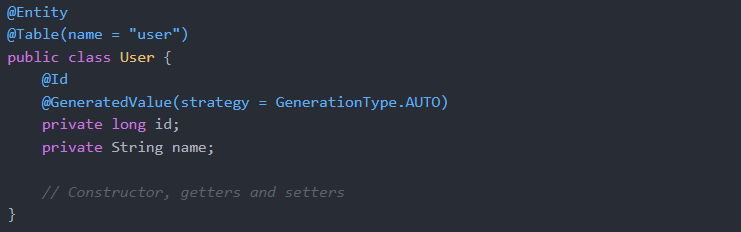
# Domain Model - Creating a User Model

Now that the database connection is up and running, we can go ahead and jump into the Domain Model. This is a set of classes, or rather models, we'll use in our application. With Hibernate, they are also called Entities, as well as annotated by the @Entity annotation.

Each @Entity is picked up by Hibernate, a table is created for it, fields are mapped, and it becomes a managed entity for the database you've set up.

First, let's create a simple User entity. We'll annotate the class with @Entity and the optional @Table annotation to specify the name for our table.

If not set, it'll just use the same name as the class:

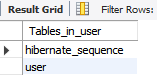


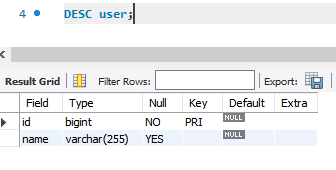
To annotate a field as the id of an entity, you use the @Id annotation, and it'll be set as the auto-incrementing, primary key of the table. Additionally, you can set that it's a @GeneratedValue and set the GenerationType to AUTO.

This is the default setting, if you omit the @GeneratedValue annotation. Other values you can set are IDENTITY, SEQUENCE and TABLE.

Additionally, you can set @Column annotations for each of the fields, providing a name for each of them if you'd like custom names - @Column(name = "user\_id"), would save the id field as user\_id instead of just id.

This class (entity) is now registered with Hibernate. If we run the application, considering our ddl-auto setting, the table will show up in your respective database, with the correct table and mappings for the data types.





# Persistence Layer - Creating Repository Classes

Next, let's work on the Persistence Layer. We'll want to have a UserRepository to perform CRUD operations on our User entities. To do this, we'll specify an interface that extends CrudRepository, and annotate it with @Repository.

@Repository is a variant of the @Component annotation, which lets Spring know that it's a component that should be managed by the IoC container. Specifically, repositories are meant to define logic for the persistence layer.

The CrudRepository extenssion accepts the entity class, as well as the id data type it should use to query:



CrudRepository declares methods like findAll(), findOne(), and save() which constitute the basic CRUD functionality of a repository. You can use this UserRepository as-is, to perform CRUD operations on User entities now, with no further setup required.

You can override some of this behavior, if you'd like, though, it's set up automatically to help you bootstrap some basic functionality.

# Business Layer - Creating a Controller

Finally, we've gotten to the Business Layer, where we implement the actual business logic of processing information, and use the components from the Persistence Layer, alongside the Domain Model to store data.

Let's create a controller, mark it as a @RestController, as we're creating a REST API, and add a @RequestMapping to it. @RestController is just a combination of @Controller and @ResponseBody, which means that instead of rendering pages, it'll just respond with the data we've given it. This is natural for REST APIs - returning information once an API endpoint has been hit.

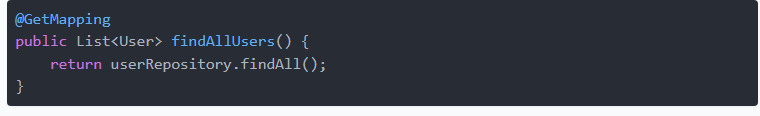
Let's go ahead and make a UserController:



We've @Autowired our UserRepository. It's used for dependency injection, as the repository class is a dependency here.

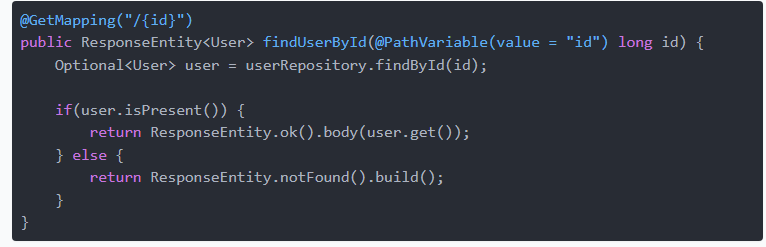
We've also used the @GetMapping and @PostMapping annotations to specify which types of HTTP requests our methods are accepting and handling. These are derived variants of the @RequestMapping annotation, with a method = RequestMethod.METHOD set for the respective types.

Let's start off with the implementation for the findAll() endpoint:



This method just calls the userRepository to findAll() users, and returns the list as the response.

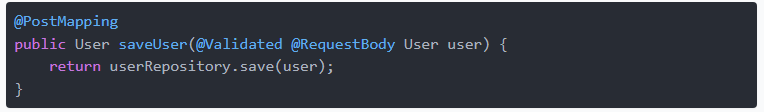
Next, let's implement the endpoint to get each user by their id:



A user with the given id might not be present in the database, so we wrap the returned User in an Optional.

Then, if the user.isPresent(), we return a 200 OK HTTP response and set the user instance as the body of the response. Else, we return a ResponseEntity.notFound().

Finally, let's make an endpoint to save users:



The save() method from the user repository saves a new user if it doesn't already exist. If the user with the given id already exists, it throws an exception. If successful, it returns the persisted user.

The @Validated annotation is a validator for the data we provide about the user, and enforces basic validity. If the user info is not valid, the data isn't saved. Also, the @RequestBody annotation maps the body of the POST request sent to the endpoint to the User instance we'd like to save.

Now, it's time to run the app and test if it works.

# Compile, Build, and Run

The default port that Spring Boot runs in is 8080. If you want to change the port for whatever reason, you can set it up in your application.properties file:



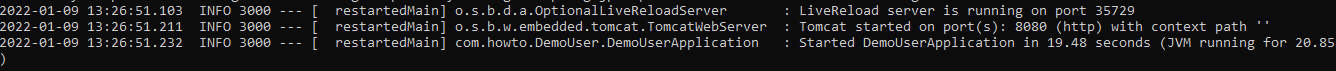
We'll be using the command line to run our project. We can run the application directly by executing mvnw spring-boot:run on the command line from your base project folder where pom.xml is located.

Another option is to package your application into a jar file and running it that way.

To do this, we just have to execute mvnw clean package and run the jar file by executing this command:



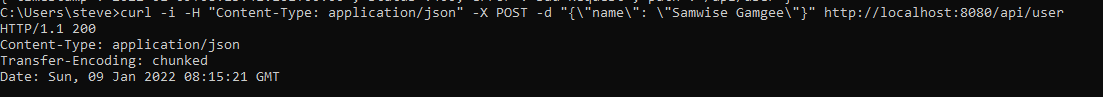
You will know when your application has successfully run if you see these audit logs at the end of your command line:



# Testing the APIs

Now that your application is up and running on http://localhost:8080/, we can now test the endpoints to see if they work.

First,let's send a HTTP POST request and add a user to our database, by providing the data required in our model. The fields in the JSON payload have to match the field names in our DB/model:



This strange syntax is necessary in order to escape the double quotes.

Note the id field will be auto populated by the database.

The API will return 200 as a response with this as the response body of the persisted user:



Add a few more users by simply re running the command after changing the user value in the JSON request.

For the GET requests, we can use browsers, curl or Postman - whatever's most convenient for you.

Let's hit the http://localhost:8080/api/user endpoint with a GET request:



Or, in your browser address bar, visit http://localhost:8080/api/user, and your browser will display a JSON response:



We can modify this URL to include a path parameter, the id to get a specific user. Let's send an HTTP GET request to <http://localhost:8080/api/user/:>



# Conclusion

There you have it. You've successfully built your very own Spring Boot REST API!

In this lab, we've built a fully functioning Spring Boot project that exposes an API to the end user. Using this API, a user can perform CRUD operations on a User entity.

We've covered the Domain Model, Persistence Layer, as well as the Business Layer, after setting up the connection to the database and configuring the project.

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