Spring Boot & Microservices

Pre-requisites

1. Java
2. Spring Framework & Rest API’s
3. Git

Contents Overview

Spring Boot

* Autoconfigurations
* Webservices
* Adding Data JPA to interact with DB
* Configuring different servers like jetty, undertow
* Running the applications using jars

Spring Microservices

* Overview Microservices
* Monolithic vs Microservice architecture
* Design patterns used for microservices
* Spring projects for microservices - Spring Boot & Spring Cloud
* Service Discovery
* Discovery Client
* Load Balancer
* Circuit Breaker with Resilience4J
* Distributed Configuration
* API gateway
* Securing Microservices
* Deploying microservices using Docker & AWS

Spring Boot:

* It simplifies development by reducing all the generic configurations you do in the application like XML configurations, Server Configuration, Front Controller Configuration, Component Scanning, Bean Dependency configurations,
* It doesn’t need any XML
* It uses a plain text property file for any application related configurations which are much easier to maintain
* It provides starter projects to automatically setup the project/environment
* Spring Boot Starter Web: This starter helps spring boot to setup necessary features for developing web applications like Server, Front Controller, Component Scanning
* Spring Boot Starter Data JPA: This starter helps spring boot setup necessary features to interact with DB, like establishing connection, supplying the datasource dependencies to other beans which does CRUD operations
* Spring Boot Starter Actuator: This starter helps spring boot to provide endpoints to check the application status, metrics, health
* Spring Boot provides a Starter Parent project to avoid versioning conflicts between the spring libraries

Spring Provides Spring Initializr for developers to quickly create spring boot projects or they can use STS IDE or STS plugin in Eclipse to use spring initializr feature

<https://start.spring.io/>

The above website allows developers to quickly create a ready to run project

Some of the dependencies

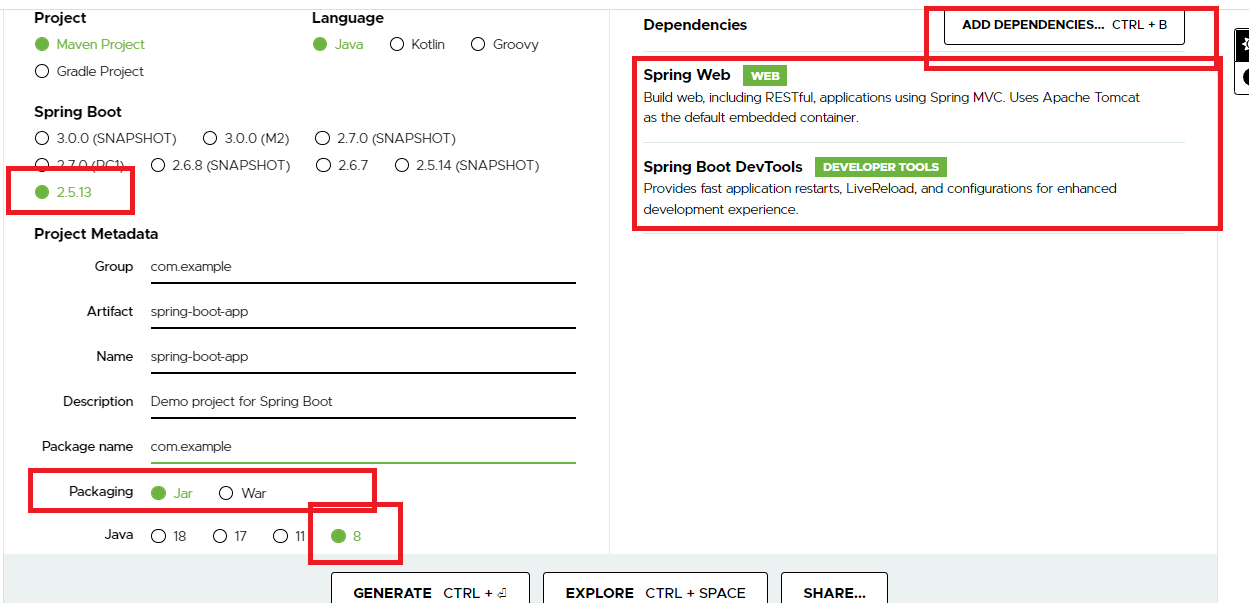
Web: This is used to develop web applications, it is a spring boot starter web library takes care of all the set up required for developing web applications

Devtools: This reloads your application to detect the changes while you are coding

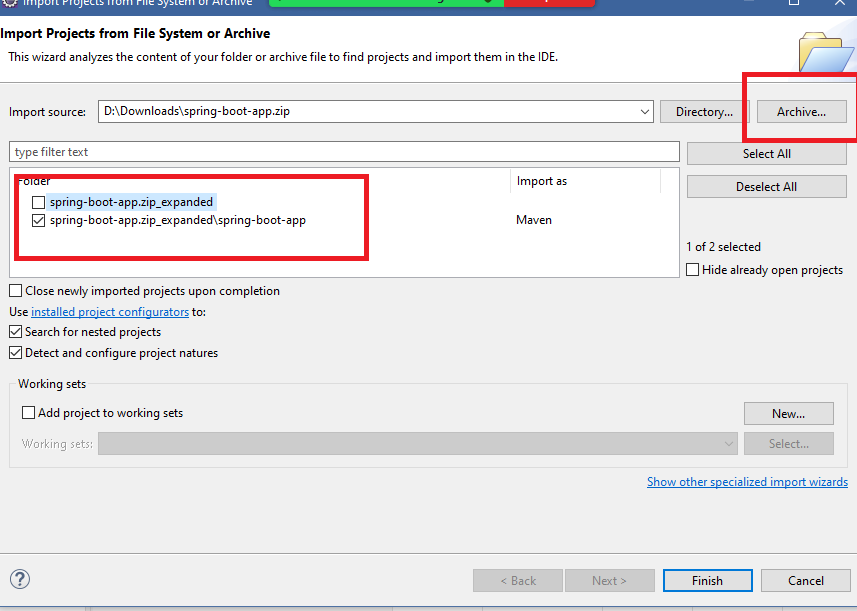
Embedded Servers: These are embedded in the project, by default spring boot provides Tomcat, but it supports two more servers as embedded server like Jetty, Undertow

Note: When you build your application for production based on the packaging like jar or war the build file is created, if its jar packaging then jar will be created, to run this you just need JDK, because you can run using java -jar command, in case its war then a war file will be built and it must deployed in the external server

Creating our first project

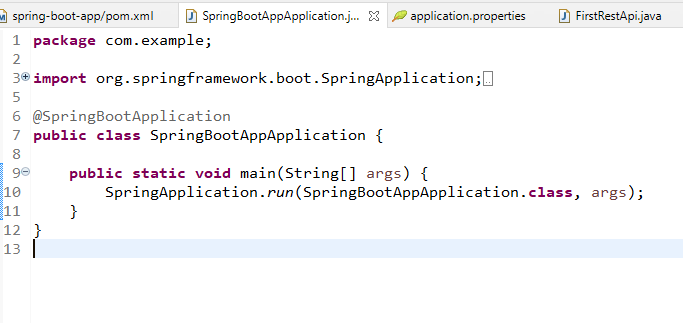


After downloading Open the zip file from eclipse



Spring Initializr provides an entrypoint to launch the spring boot application

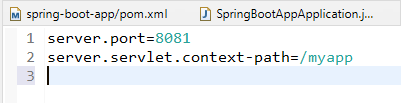
SpringBootAppApplication.java



@SpringBootApplication: It is a predefined annotations which does auto-configuration based on the libraries present in the classpath, it does component scan from the package it is present

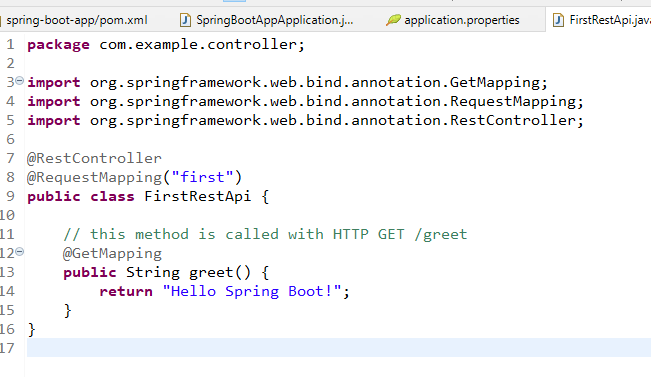
Note: Create all your Spring component classes or other stereo type classes like @RestController, @Service, @Configuration, @Component, @Aspect and so on inside the package or subpackage where @SpringBootApplication class exists.

application.properties

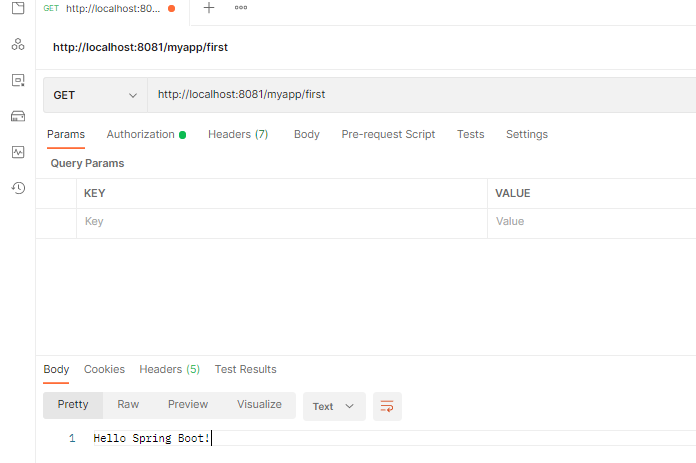


This property file is a default file the spring boot loads, it reads the properties and sets up the configuration accordingly

FirstRest.java



Output:

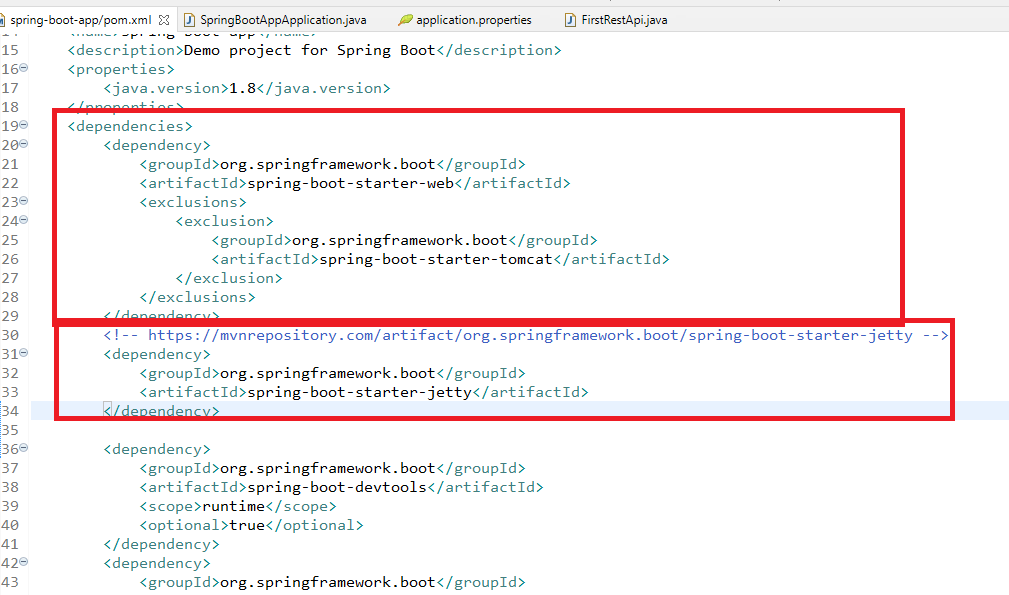


Here the default embedded server is Tomcat, but spring boot supports other embedded servers like Jetty, Undertow

Configuring Jetty Server

* You need to add spring boot starter jetty library in the pom.xml
* You need to exclude spring boot tomcat server in the pom.xml

pom.xml

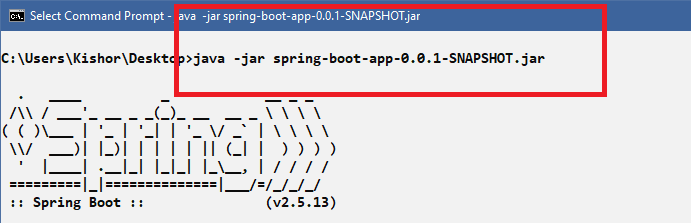


Note: Stop & Restart the application, you can see in the console which server started

Note: The spring boot applications are ready to run application even in production environment, for that you can build the application

Since we have Maven, we can use its goal: package to build the application, it is jar package, hence we get jar file.

* To run the application you can use jar -jar file-name.jar

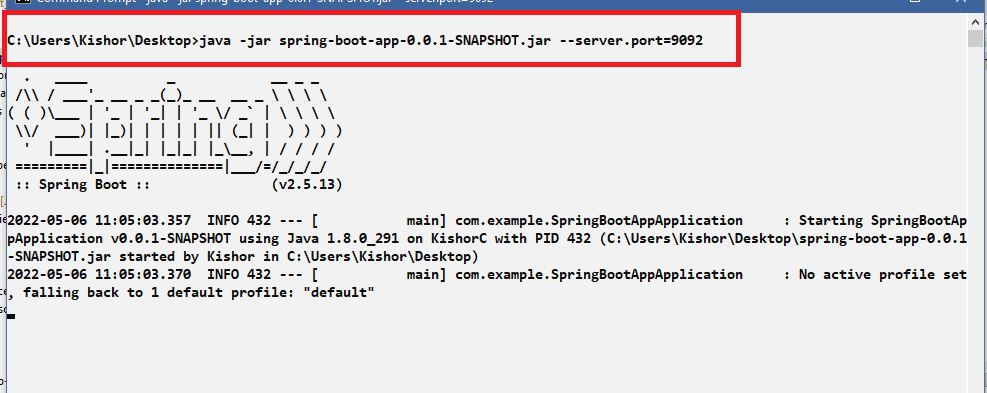


Note: You can assume this as a cloud machine and some unix based machine terminal, they must have java installed and the jar which is built, if they have the Java & Jar file then they can run it

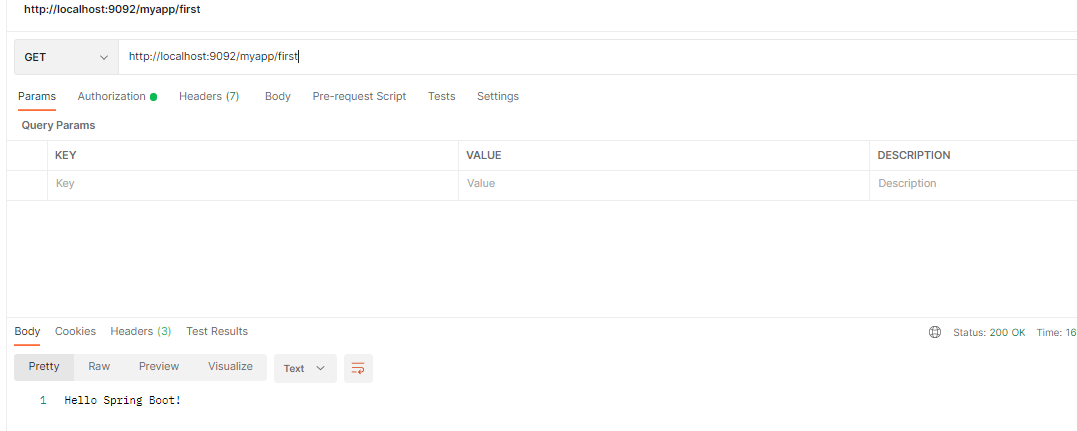
Overriding the properties

Since the application.properties is using server.port = 8081, you can override and give different port

i.e.java -jar file-name.jar --server.port=9092



Output:

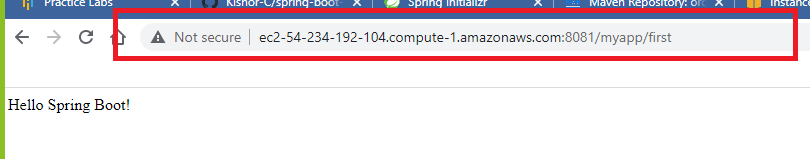


You can deploy this in AWS/Azure clouds, there you need a machine with Java installed

Things to do for launching our jar in the AWS machines

* You need to create an EC2 instance
* You need to connect to the EC2 instance either from GIT bash or putty
* You need to install git & java
* You need to get the build file i.e., jar from GIT in EC2 instance
* You need to run the jar in the EC2 instance
* Open TCP port that can accept incoming request for you app

Once you run your application in EC2 machine you must able to access the application through the port



Above application is accessed via AWS-EC2 machine URL, but to run that application we need follow all the necessary steps like

* Using Git
* Opening TCP port in EC2 machine
* Installing java in ec2 machine

We can interact with any databases through spring boot

* H2 (in memory database)
* MySQL / Derby / Oracle and so

You have a starter libraries to interact with the database which is spring boot starter jpa

Benefits you get from the spring boot starter data jpa

* It establishes connection with the database looking at properties in the application.properties
* It injects all the dependencies required to perform Database operations
* It provides proxy implementation for the database logics with the help of Repository interfaces
* It gives two repository interfaces like CrudRepository<T, ID> & JpaRepository<T, ID>
* You need to create an entity class that can be used with the above repositories and you need to create an interface that can extend the above repositories
* Spring Boot data jpa implements the interface based on the entities you have used in the repository
* You don’t have to write database logics at all
* You can inject the proxy implementation in the service layer using the interface you have created

Assume you have an entity mapped to employee table

@Entity  
class Employee { … } // mapped to employee table

You can create an interface to work with employee table as below:-

interface EmployeeDao extends JpaRepository<Employee, Integer> {   
}

(or)  
interface EmployeeDao extends CrudRepository<Employee, Integer> {   
  
}

(or)

interface EmployeeDao extends CrudRepository<Employee, Integer> {   
 @Query(“select e from Employee e where e.salary = ?1”)  
 public List<Employee> getEmployeesBasedOnSalary(double salary);  
}

Here the spring boot data jpa implements the method getEmployeesBasedOnSalary() with the query mentioned.

You can inject the proxy implementation using the EmployeeDao interface via @Autowired

@Service  
public ServiceImpl {   
 @Autowired  
 private EmployeeDao dao;  
}

From EmployeeDao you can access all the methods of JpaRepository/CrudRepository all the methods interact with the table the entity is mapped with.

CrudRepository<T, ID> has following methods

* save(T)
* deleteById(ID)
* findAll()
* findById(ID)

JpaRepository<T, ID> has following methods

* All the methods of CRUD
* sort()

All the above methods perform the operations depending on the entity class

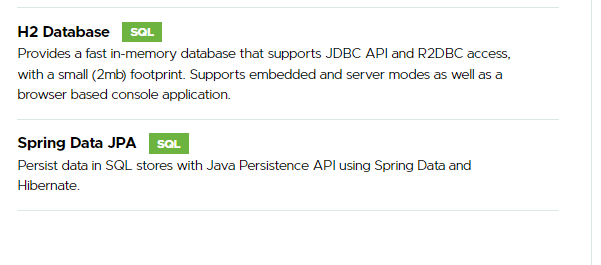
Now we can interact with any databases using the Repository interface

* You can use H2 database which is a in-memory database

JpaRepository & CrudRepository: They work with SQL Databases

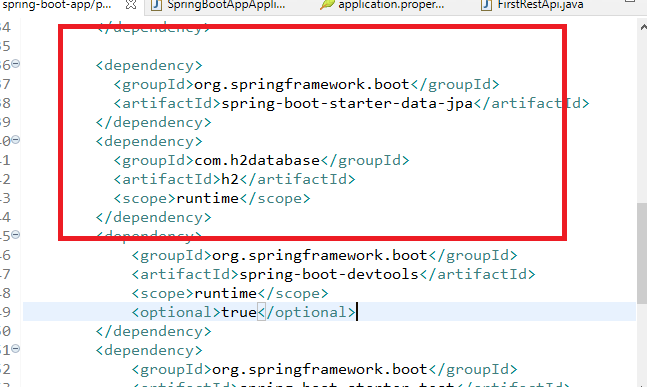
MongoRepository: This interface helps to work with MongoDB which is a NoSQL Database

Dependencies required to work with H2 database



You can copy these dependencies entry to your project

pom.xml

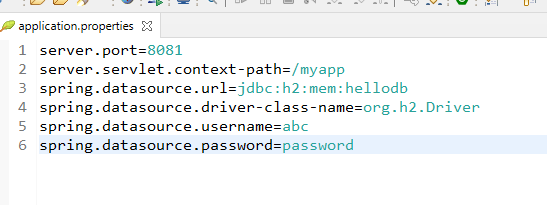


Note: You don’t have to create any table, you can use entity class

Things to create in the application

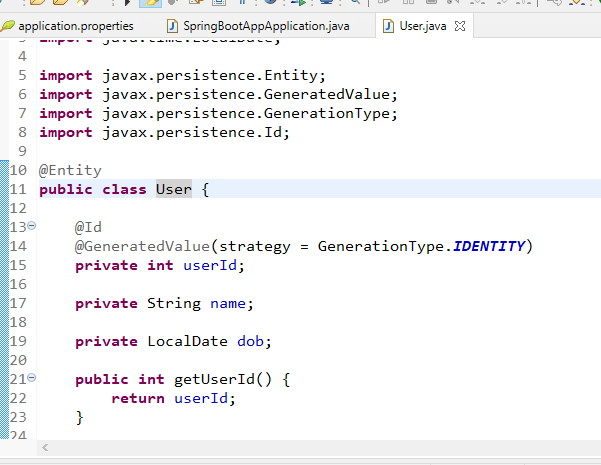
1. Entity class
2. Database configurations in application.properties
3. Interface that extends Repository
4. Injecting the Repositories in the Service
5. Injecting the Service in the controller
6. Create web services that does CRUD operations with appropriate HTTP methods

application.properities



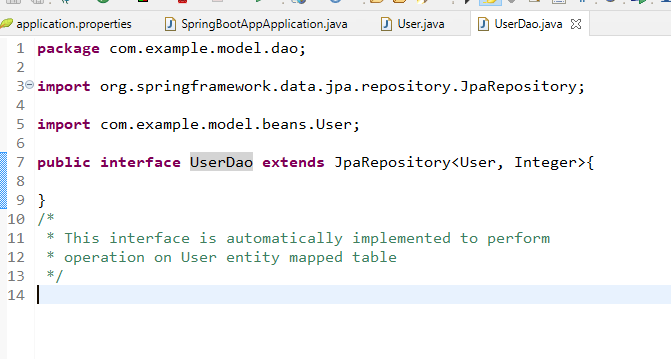
Entity class

User.java



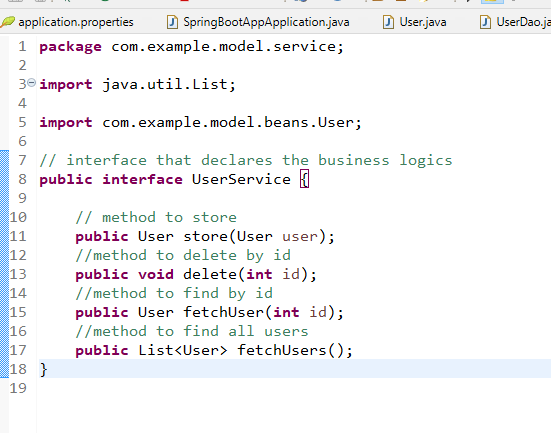
Since we are using H2 we don’t need any physical database

UserDao.java



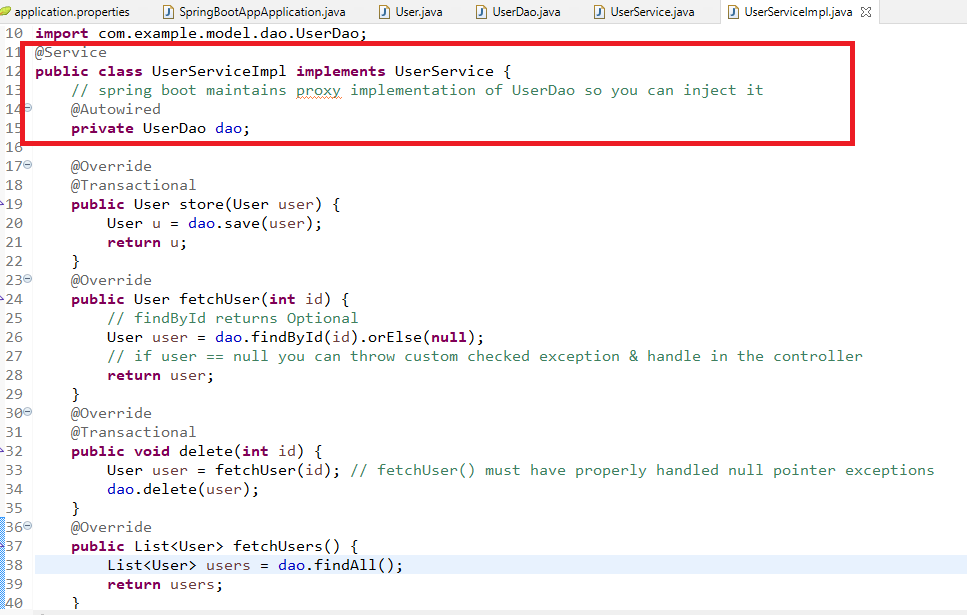
You can create a Service interface & Implement it

UserService.java

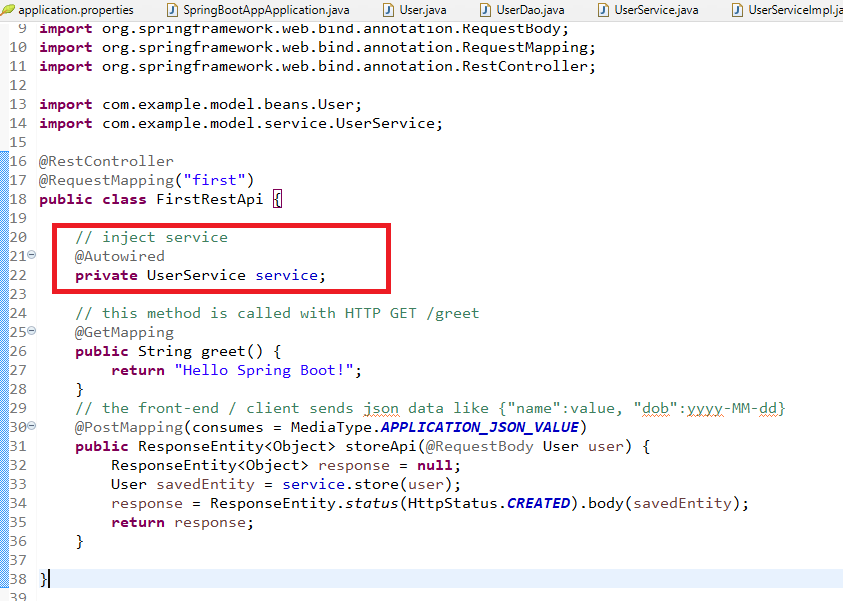


You need to implement this interface to call the repository methods & save() , delete() must use @Transactional as they modify the entity

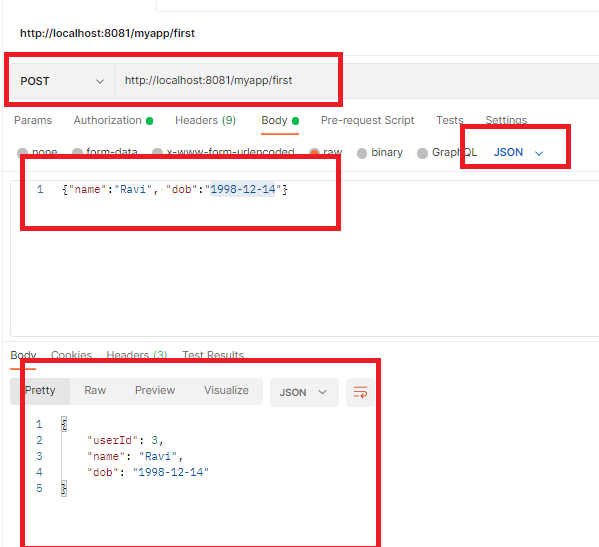
UserServiceImpl.java



Now you can create webservices to perform CRUD operations



This webservice has currently storing the user, you can pass user data from postman



Representing the data in XML format

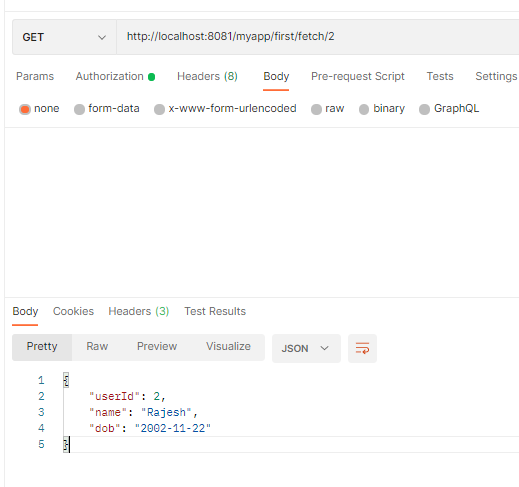
You need to use @XmlRootEelement on top of the bean, this specifies the spring boot that java bean can be converted to XML format and vice versa



Webservices for fetch by id and fetch all users



Output:



Http methods follows the standard for the particular operations

1. GET: It is for fetch operations
2. POST: It is for creating new resources
3. DELETE: It is for deleting the resources
4. PUT: It is for updating the existing resources

According to the Http methods spring boot has annotations

@GetMapping, @PostMapping, @DeleteMapping, @PutMapping

Use these annotations based on the operation you do

Activity:

1. Try all the above examples
2. Perform other operations on the User entity & use the appropriate mapping
   1. Deleting the user by id: Hint: *DELETE /100*
   2. Update the DOB using the id: Hint *PUT /100/2000-10-25*
3. For the same example create a Checked Exception which helps to avoid null pointer exception when the user id is not found in the database

Suppose user id 100 is not present then you must throw a checked exception ‘UserNotFoundException with an id 100’, this error message must appear in JSON format

ex: GET /fetch/100

Output: {“message”:”UserNotFoundException with an id 100”}

If GET /fetch/100 has user in the database then you can show the user details in JSON format

1. Create an entity that is part of the User, which will have One to Many association, the entity must be Friend, so you should able to add friends to the user using /POST, a user can have multiple friends,

Friend entity can have 2 properties id & name

ex: If user 100 is retrieved you can show all the friends of user 100 as below:

{  
 userId: 100, name : “Raj”, dob : 2000-10-22,   
 friendsList : [{id: 1, name:”Alex”}, {id: 2, name:”Bruce”}]  
}

ex: If user 100 needs to add a friend you can use

HTTP method & URL: /POST /addFriends  
JSON Data: {name: “Raj”}

Hint: Use @OneToMany annotation in the User entity to map list of Friend entities