Spring Boot & Spring Cloud Microservices

Pre-requisites

1. Java Technology: OOPS & Collection Framework
2. Spring Framework: Dependency Injection, Spring MVC & Annotations

Spring Boot:  
It simplifies developing the spring applications by taking care of all the generic configurations

i.e,

- You don’t have to write any configuration file like xml files

- No need to setup because the server is added into the application itself

- Set up required for applications are provided by spring boot starter projects

Starter projects in Spring boot

* Spring Boot Starter Web: It automatically configures setup for web development like Front Controller, Server Configuration (Embedded servers), Component Scanning
* Spring Boot Starter JPA: It automatically configures the dependencies required for Databases like Connections, Connection Factories, Templates (JdbcTemplate & HiberanteTemplate)

For Spring Boot we need following software’s to be installed

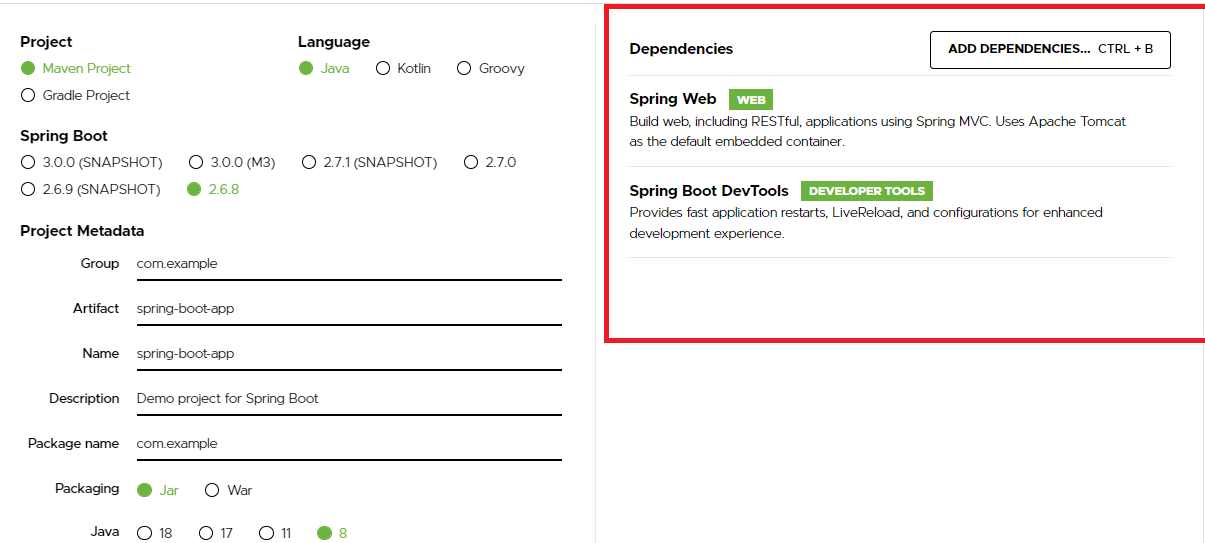
* Java
* Eclipse IDE / STS
* Postman App

Spring provides us a website to set up the spring boot project i.e., Spring Initializr

Creating our first web application in Spring Boot

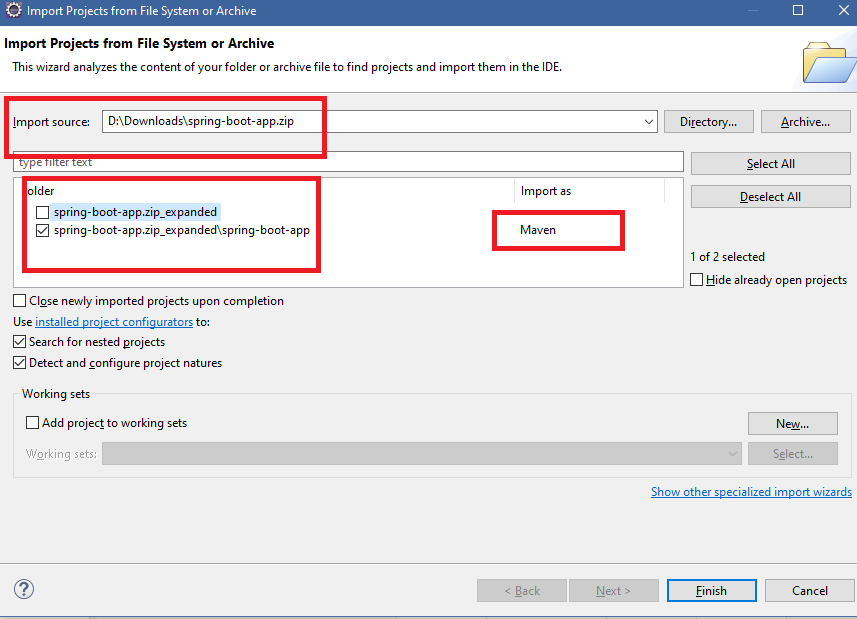
Dependencies required are:

* Web : Provides configurations for web applications like Front Controller, Server, Component Scanning – we get everything auto-configured for us
* Devtools (Optional) : auto-reload feature

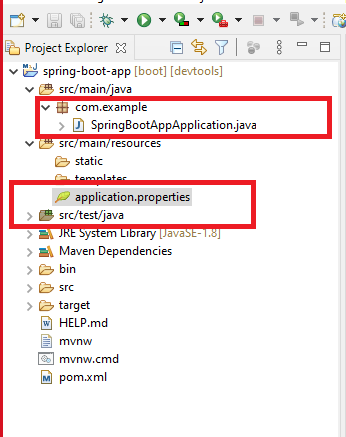


Click on Generate so that it downloads the zip file of the project, then you need to open this project from the Eclipse

Open project with Maven selected in the eclipse



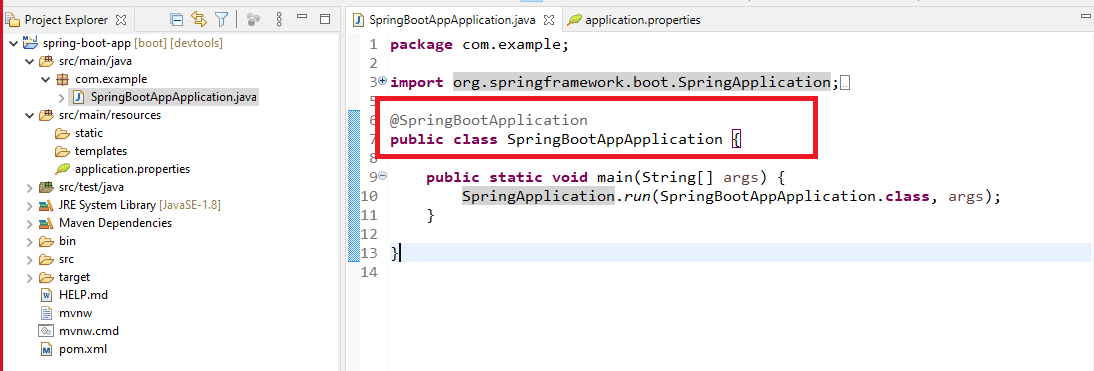
When we see the project structure we can observe two files



The Java file is the entry point file to launch your application, the application will be launched in the embedded server (Tomcat)

The application.properties is the configuration file for your application which keeps all the application related configurations like data-source informations, server port information’s, microservice related configurations

Note: Spring Boot uses application.properties as the default configuration file, but you can change this name also and load a different configuration file

The Java file looks like below  


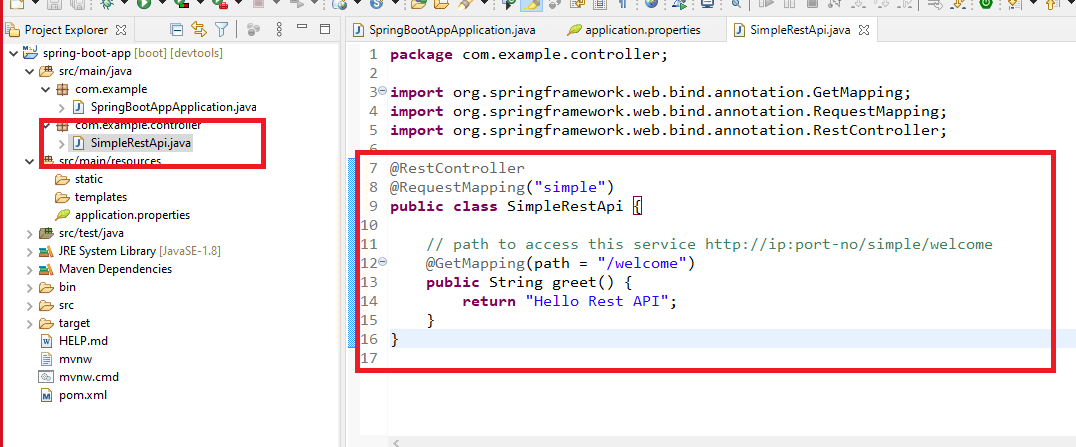
@SpringBootApplication:

* It takes care of doing all the auto-configurations for the application based on the library we add in our classpath
* The class having this needs to be loaded so that an application which is fully configured will be running for use.

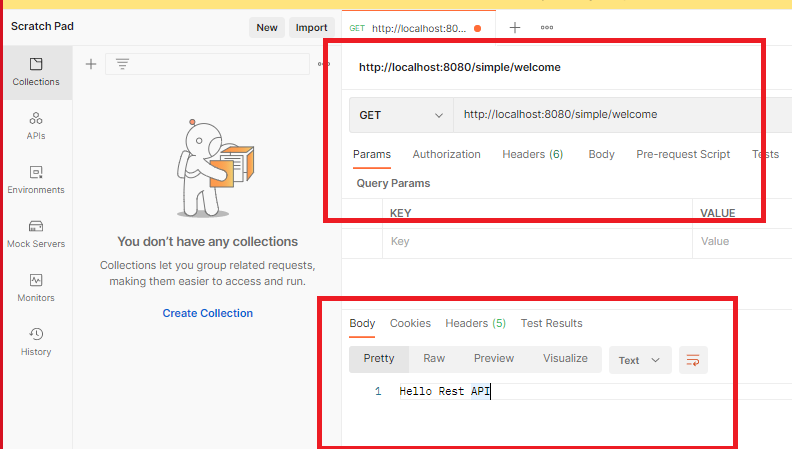
Since we have a web starter added we can create webservices in the application so that different applications can consume the data.

In Spring Framework we can create webservices using @RestController on top of the class

SimpleRest.java

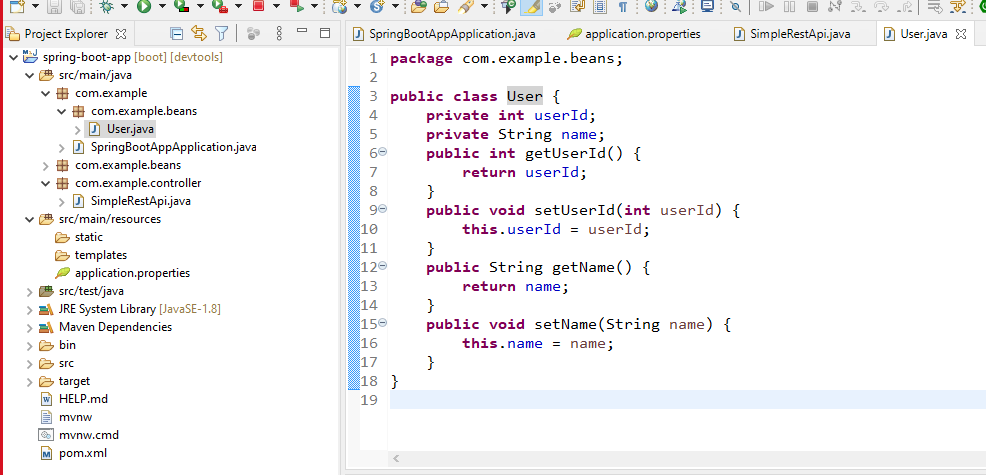


Now we are ready to run this application, since it is run in an embedded server the default port will be 8080 and the server is apache tomcat, but we can change all these configurations if required.

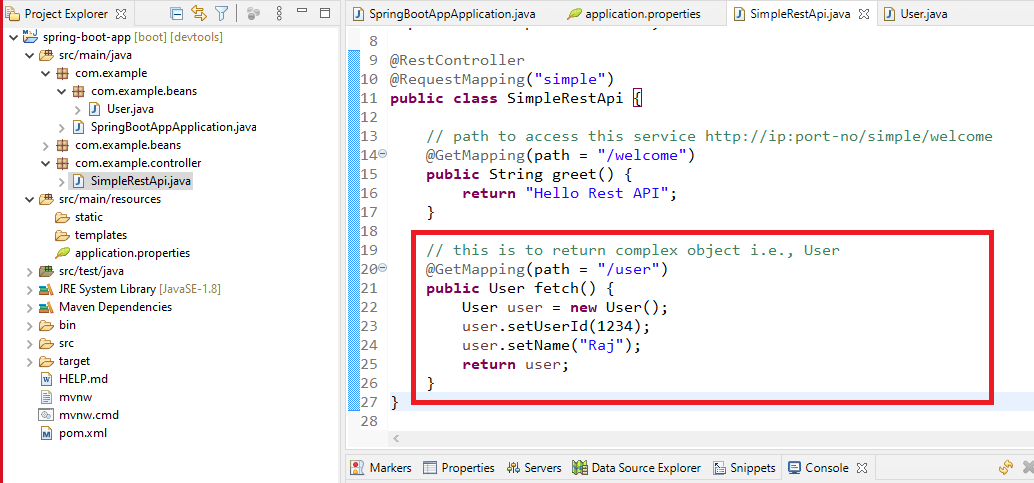


Here the response data is simple type hence it is coming in text format, however if the data is an object then it comes in JSON format.

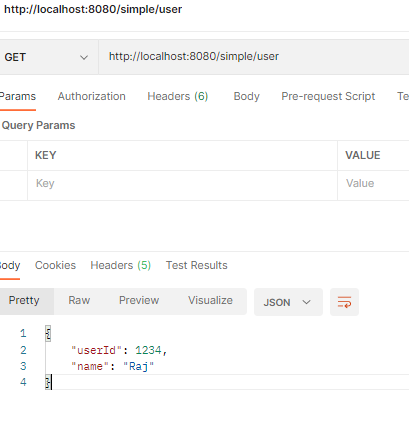
User.java



Now the Rest service can respond in json format if it returns user object



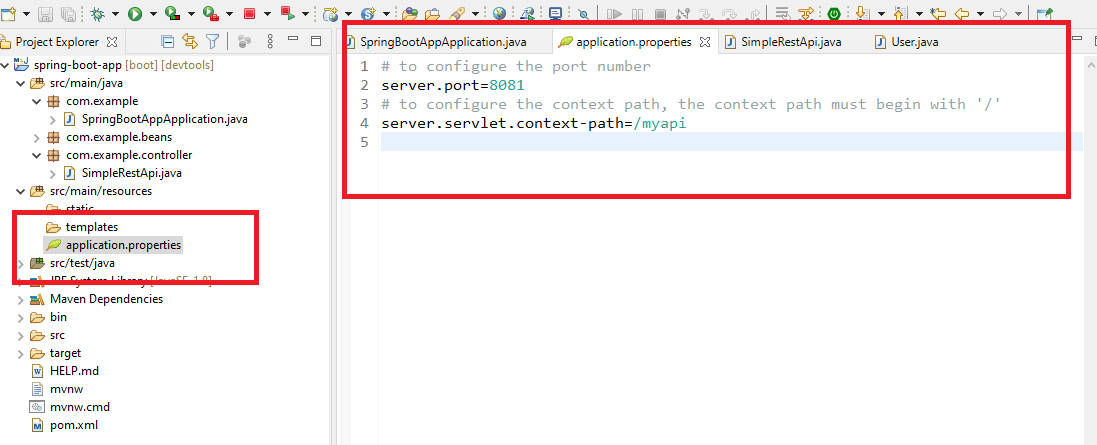
Output:



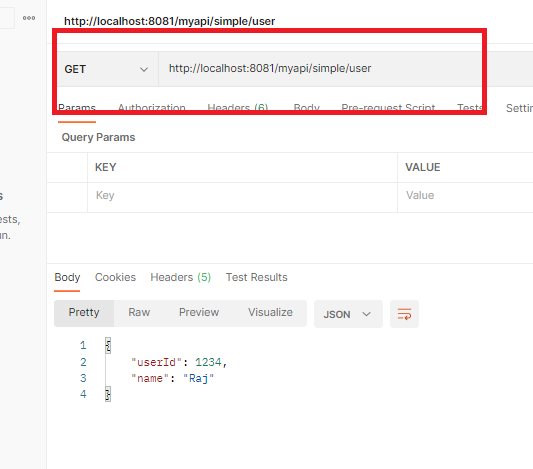
Configurations that needs to be done for the application

* How to add the context path to the application i.e., the root path to access the application
* How to configure the port number
* How to configure data-source information

All these configurations we need to do it in the application.properties

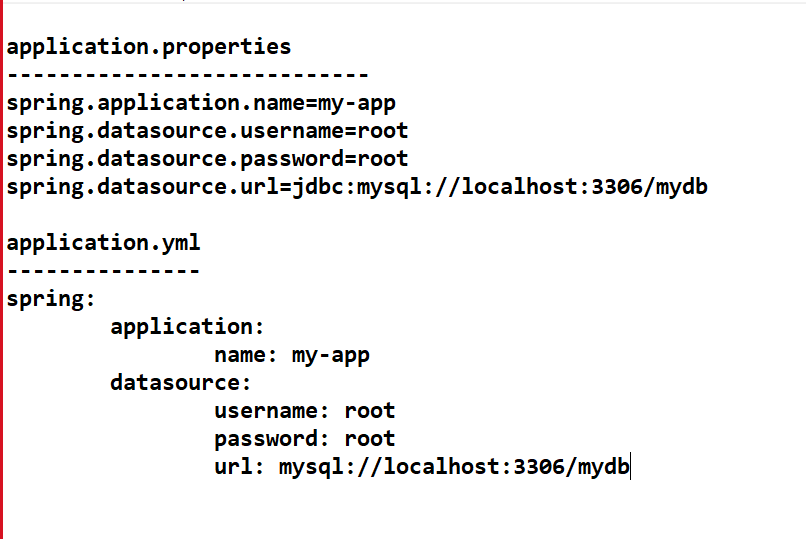


Now we must able to access the application using /myapi

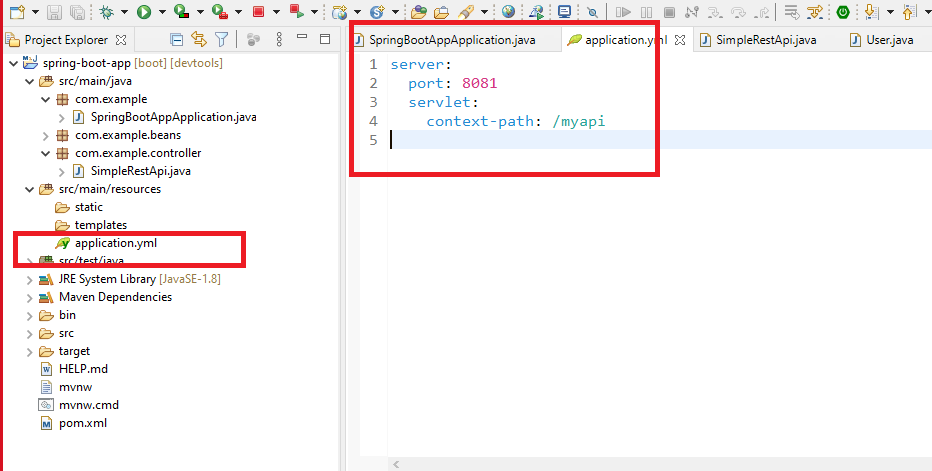


YAML Files:

These are alternate files for property files, they use indentation to mention the properties & sub-properties so that you can avoid writing property names repeatedly.



In Eclipse you will get plugin to convert properties to yaml files



Note: In Eclipse the plugin is taking care of providing the indentations to the existing properties, however if have yml already & want to add properties then we must take care of providing the indentation.

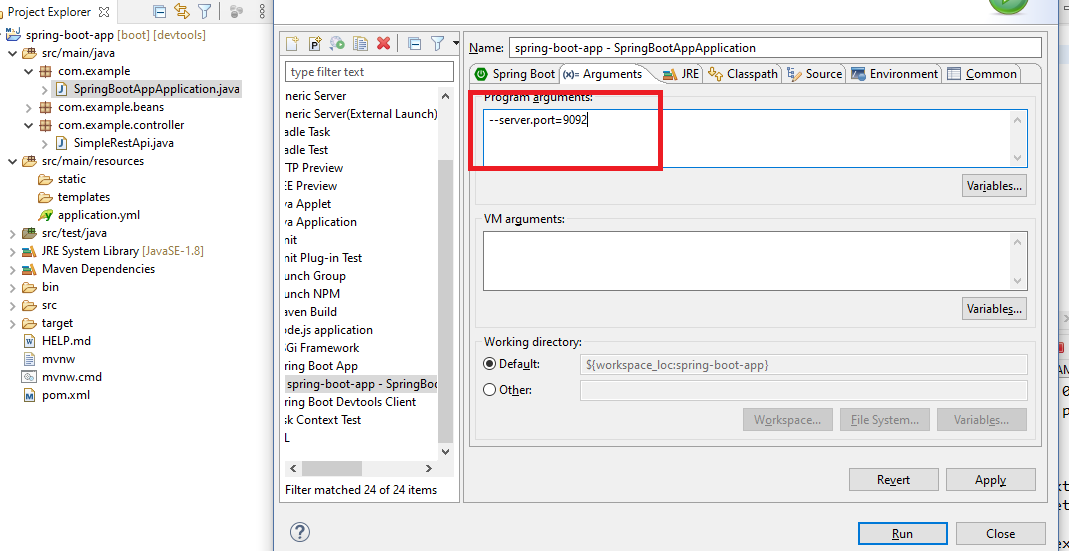
Overriding the properties

We can override the properties mention in the application.properties/yml while running the application.

* You can launch the application in eclipse and change the command line arguments
* If you have a deployable jar file then you can use command line arguments in the terminal

How to launch the application in eclipse by changing command line arguments

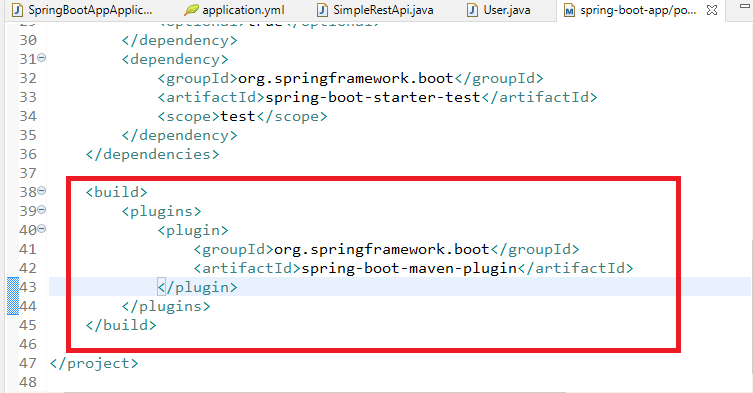
We need to use the properties with two hyphens followed by properties separated by dot



If in case you have a deployable jar then you can use the same arguments at the of time running

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

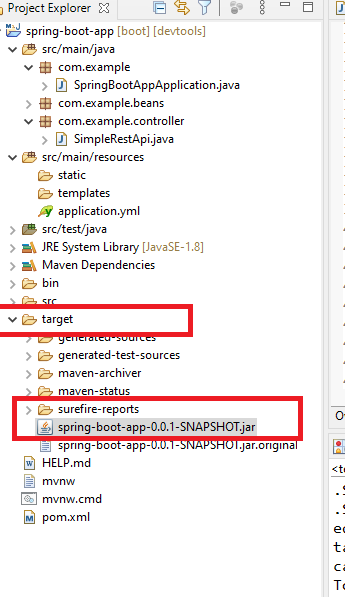
We can use maven built to build the deployable jar file, In Spring Boot its already configured to build a jar which will be an executable jar because of the entry in pom.xml



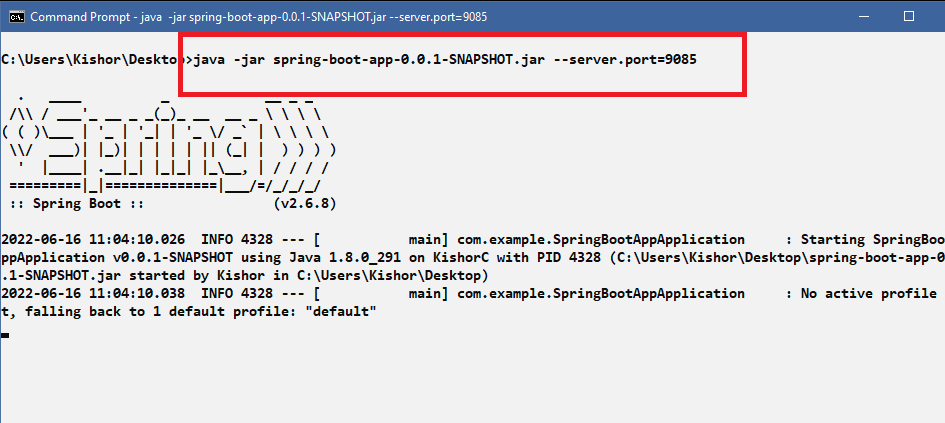
Note: This entry comes only we create project from spring initializr

How to build the jar from eclipse

Project -> Run As -> Maven Built -> Goals: package



Note: refresh the target folder to see the build file, we can run this jar using java command now



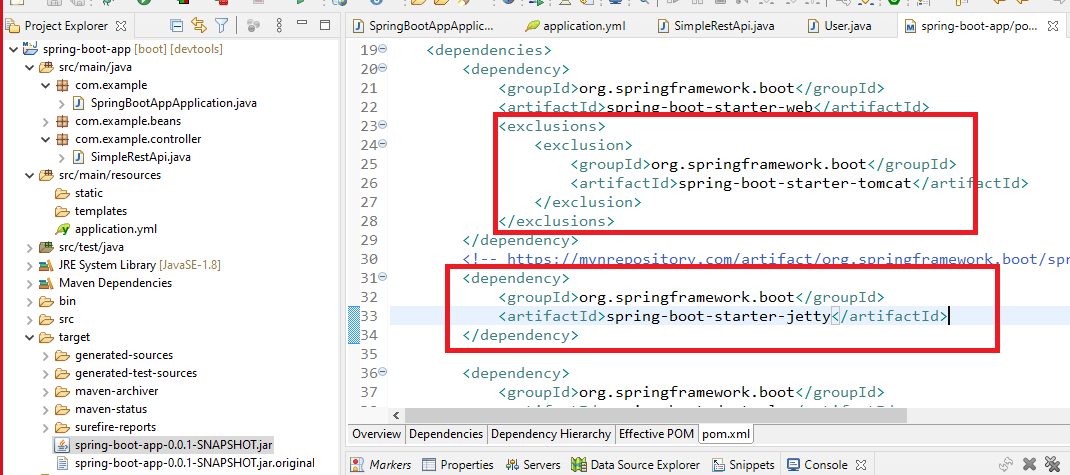
Spring boot provides following Embedded servers

1. Tomcat
2. Jboss – Undertow
3. Eclipse – Jetty

To configure different servers you need to exclude the default container tomcat and add different embedded server.

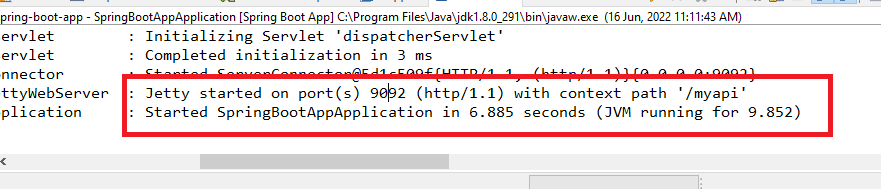
Steps

1. Exclude the embedded tomcat dependency
2. Add the other embed server you want to use



Output:

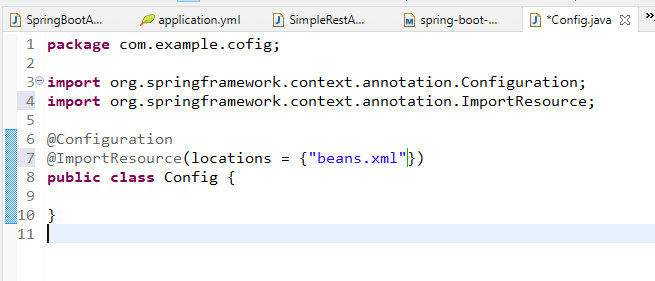
We must see in the console the jetty server has been used instead of tomcat



By default spring boot applications are packaged as jar, however if we want to deploy in our own external server then we need to package as war.

Can spring boot load XML files and use the configured dependencies in the application

Spring boot provides @ImportResource annotation which accepts the location of xml files so that the dependencies of the application can be used



Here if the beans.xml has <bean> configurations which registers in the spring container we can just load the xml files and use the registered beans.

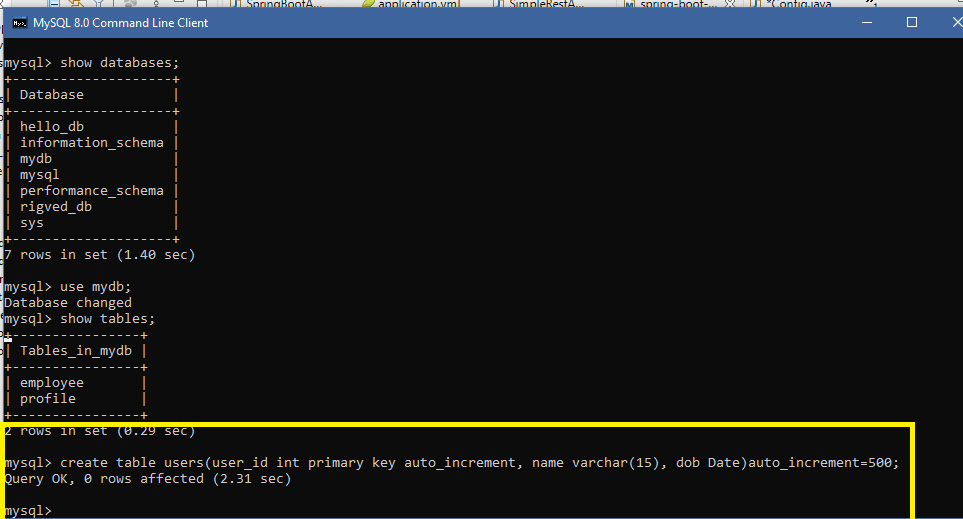
Spring Boot Best Practices:

1. Build System – Maven or Gradle
   1. Dependency Management: No need to provide version for any list of dependencies
   2. Upgrading from one spring boot version to another version would be easier
2. Spring Boot Starters: Avoids lot of copy paste codes you do for some common tasks like database connections, dispatcher servlet, dependency injection
3. Packaging the application as an executable jar in the production & it works in cloud environment also
4. Referring the official spring boot migration guide to know about how to migrate old spring boot versions to new spring boot which gives you step by step solution whenever you want to migrate
5. Using developer tools to get a development experience because it re-loads the application when you modify the code & this is automatically disabled in the production.

Performing CRUD operations using Database, we can use either in memory database(H2) or the physical database (MySQL, Derby, …).

We have a spring boot starter to configure the data-source which is Spring Boot Starter JPA.

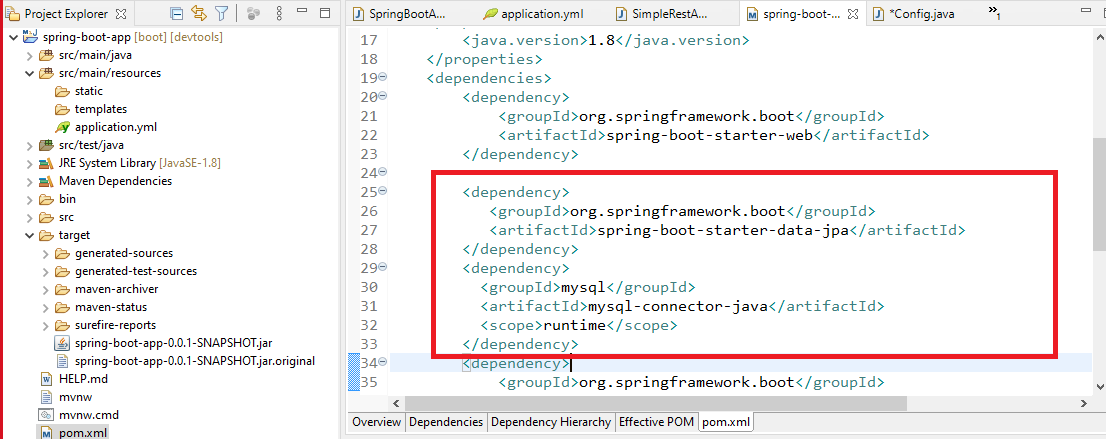
We can create a User table which will have user\_id, name & dob columns



Now we need an entity class that maps to this table and its column

We need to add a library to get all the relevant annotations for entity classes & also for database operations

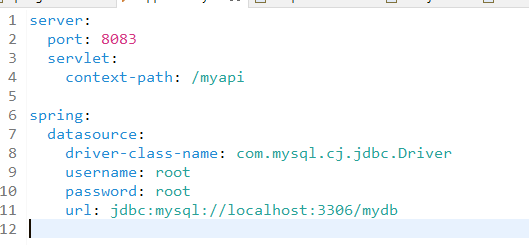
* Spring Boot Starter Data JPA



Note: If you don’t have database you can use H2 library that creates in memory database

Configuring the datasource information to interact with MySQL

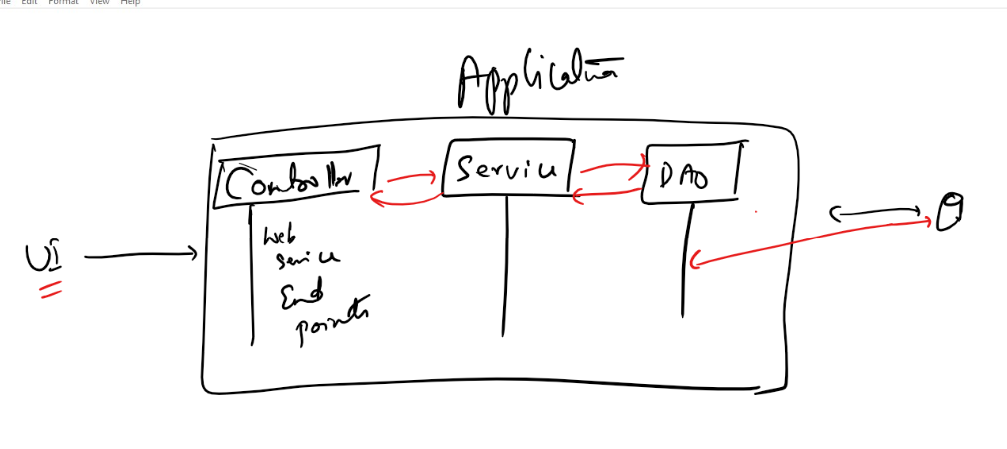
application.yml



If you use H2 then you can provide any values to the username, password & url

Note: When we have spring boot data jpa library already, then it always looks for data-source configurations in the property file, if we don’t have it then spring boot raises error.

Since we are interacting with the database, we need to separate the logics into multiple layers like Controller layer, Service layer & DAO layer

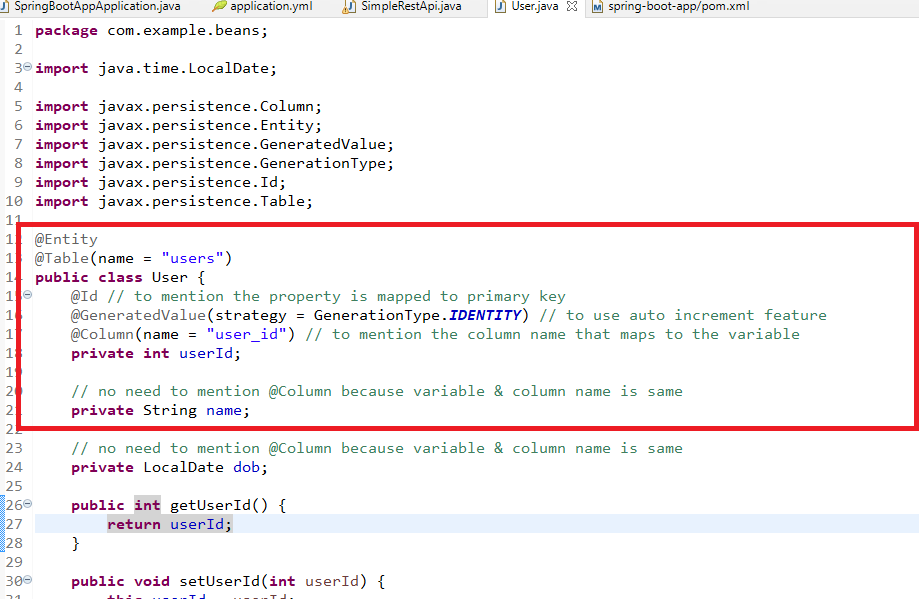
Controller will have web service end point url, It uses Service instance that will have business logics & Service uses DAO instance that will have methods to perform Database operations like Insert, Update, Delete, Retrieve.

Transactions are handled in the Service layer itself.

Since Controller talks to Service & Service talks to DAO we need to abstract the implementation of the Service & DAO and use only the interfaces in the client code.

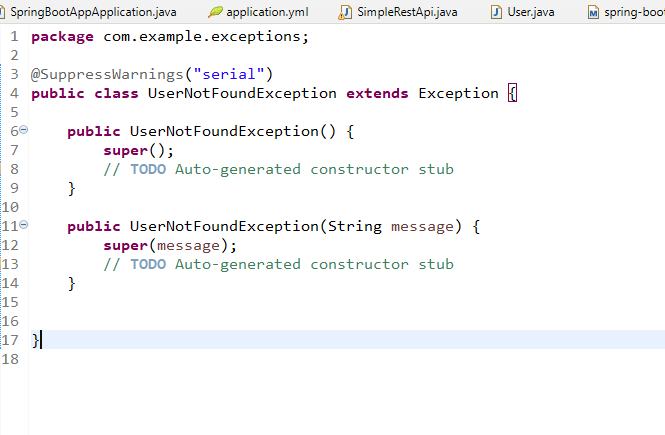
We can also create an Exception class to throw if any problem occurs so that user will know what happened.

User.java



Since it is interacting with the database and always we can’t expect to get a successful response so we can also create an Exception class that will be raised if any data we are searching is not found

UserNotFoundException.java



To interact with the database Spring Data Jpa gives you interfaces which are called as Repository interfaces which will have CRUD operations.

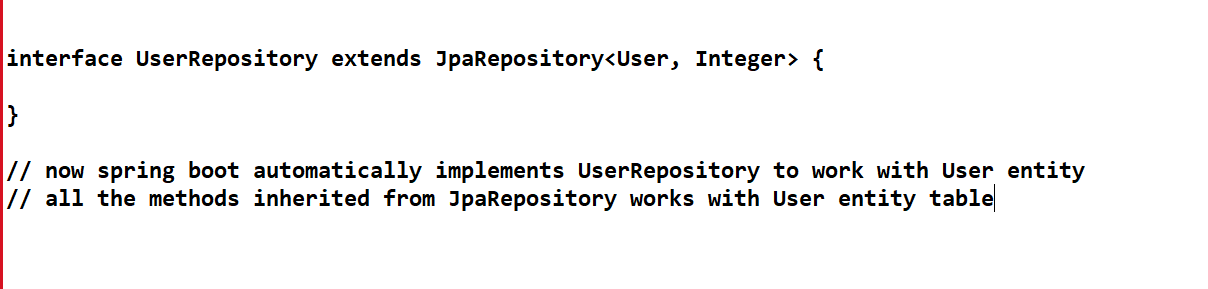
1. CrudRepository<T, ID>: Has methods to perform CRUD operations
2. JpaRepository<T, ID>: Extends CrudRepository and also provides methods for pagination & sorting

These two interfaces can dynamically perform the operation on the entity without any implementation, Spring Boot can implement these interfaces once it knows these repository is mapped to which entity class.

These repository interfaces like CrudRepository or JpaRepository has below methods.

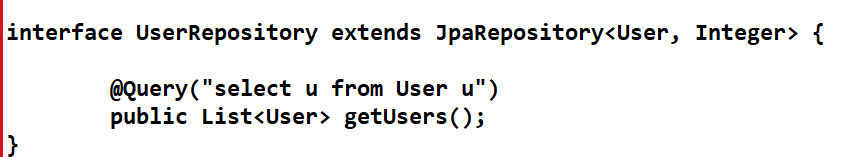
1. save(entity)
2. delete(entity)
3. findAll()
4. findOne()

All these methods are automatically implemented by Spring Boot

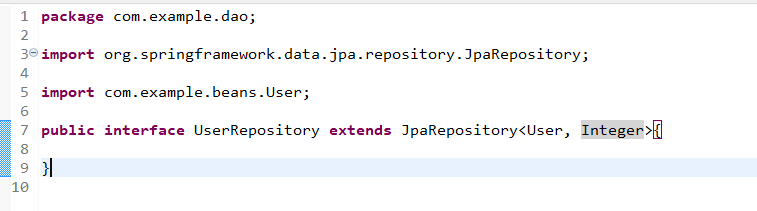


Now we need to just supply the implementation of UserRepository to the Service layer using @Autowired

Note: You can also create custom methods inside this interface which will also be automatically implemented.



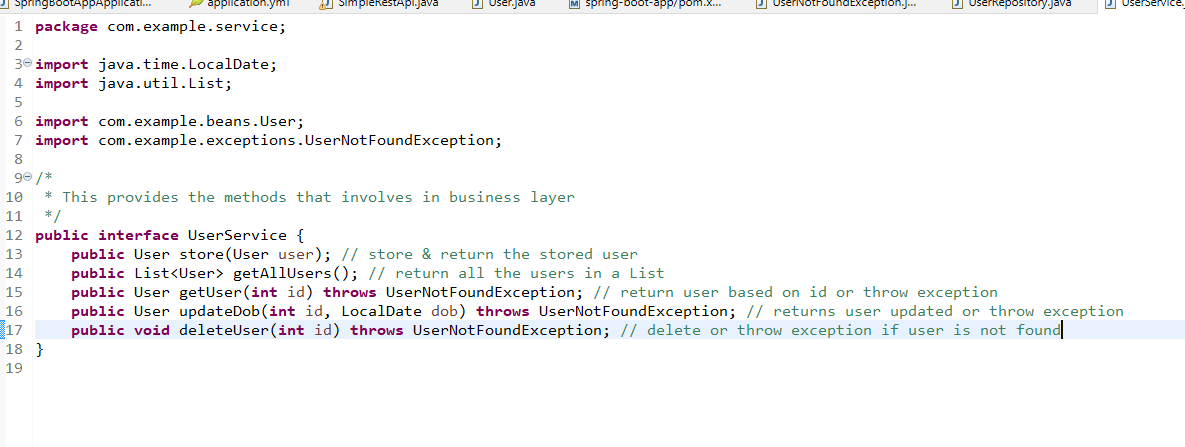
UserRepository.java



This UserRepository is automatically implemented and registered in the spring container, so we need to only supply the UserRepository in the Service layer.

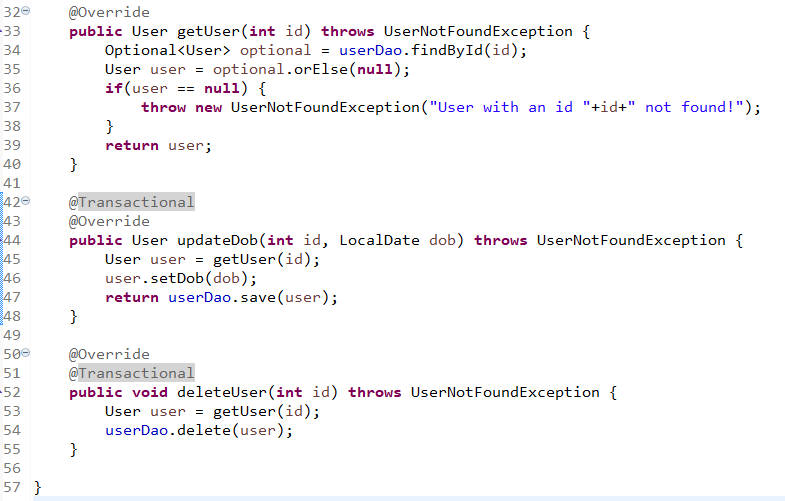
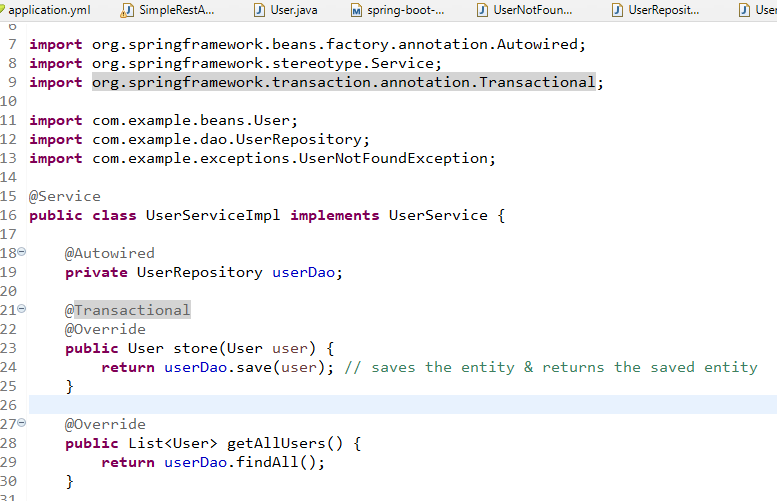
Firstly we need an interface in the Service layer that is used in the controller

UserService.java



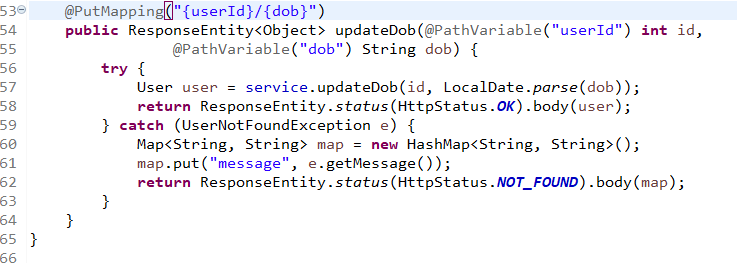
These methods we need to implement by providing an implement class that uses UserRepository

UserServiceImpl.java

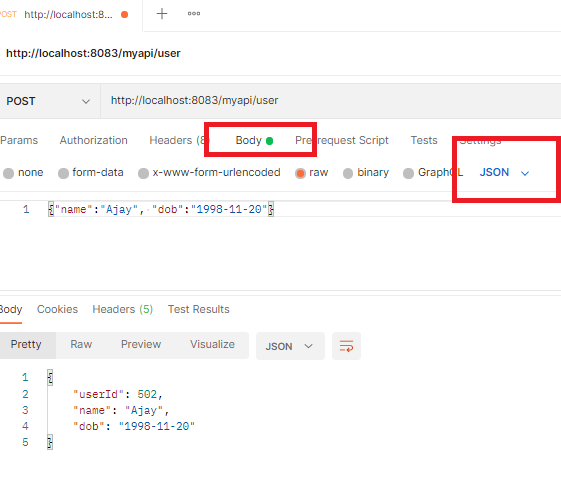


These methods are called from the controller

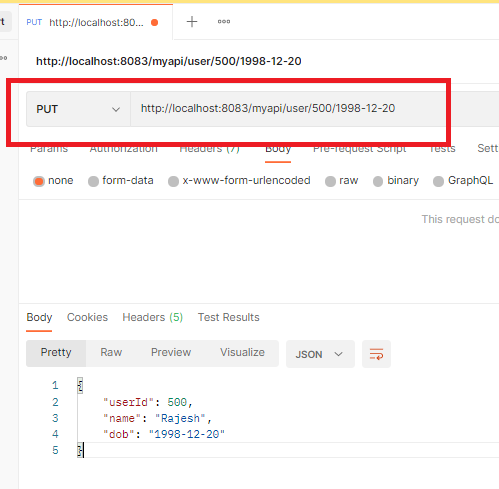
SimpleRestApi.java



Post method testing



PUT Method testing



In the Entity you can have an association mapping like @OneToOne, @OneToMany, where a single user may have one or more mapping of a particular object

ex:

User has an address

(or)

Order has one or more LineItem

(or)

Customer has one or more Order

Activity:

Create a layered architecture (Controller, Service & Dao i.e., Repository interfaces) which can perform customer to order items, the application must have a Customer who can place multiple orders and each order can have multiple line items

Customer will have: id, name, List<Order> (one to many)

Order will have: id, List<LineItems> (one to many)

LineItems will have: id, itemName, quantity, price

You must able to place one or more orders from the postman & also must able to view orders from the customers.

You will have 2 webservices

1. Place orders
2. View orders