Course Content

* Spring Boot
* Spring REST Webservices
* Spring Cloud and Microservices

Software’s required

* Java 8 / JDK 8
* Eclipse IDE / STS
* H2 or MySQL
* Open Internet

Spring Framework: It is a Java Framework used to develop different types of applications like web, mobile, enterprise, cloud and so on, it simplifies the development process by providing many design patterns required to build an application

* Dependency Injection
* Singleton & Factory
* MVC
* Proxy
* Prototype

Note: Spring Framework uses XML configuration file to configure the dependencies, in this XML file you will mention all the classes that Spring should instantiate in the container.

Spring Boot

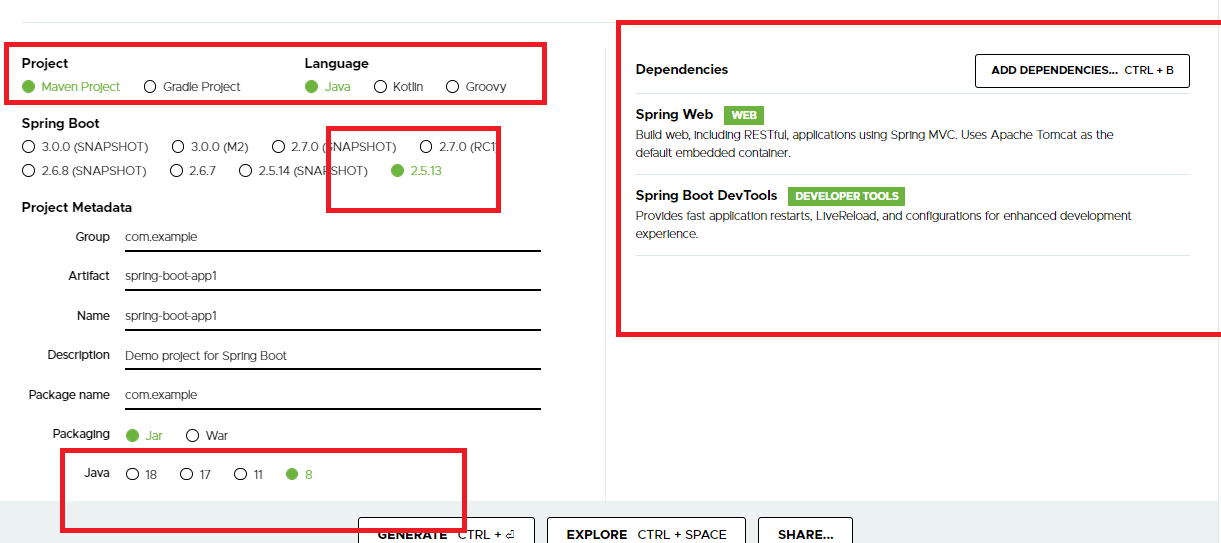
It simplifies developing spring applications in a simple way by taking care of all the generic configurations i.e., spring boot automatically does the configurations based on the spring boot starter libraries you add in pom.xml

* You don’t need any xml file because there’s a spring boot starter library that takes care of creating objects in the container
* You don’t need to set up server because there’s a spring boot starter web library that adds server to your application
* You don’t need to supply the dependencies for datasources because there’s a spring boot starter data library it takes care of autoconfiguring the datasources
* You can migrate to another version without making more changes because spring boot provides a parent library and all other spring boot libraries follow this parent library, so if you change parent library then all the other spring boot libraries follow that change automatically.

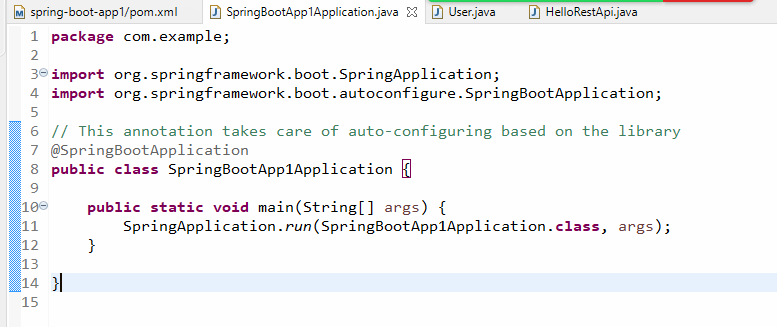
Spring vs Spring Boot: Suppose you want to develop any web application’s or webservices you will configure following things in both

|  |  |  |
| --- | --- | --- |
| Features | Spring Old approach | Spring Boot |
| pom.xml | spring mvc, faster xml | spring boot starter web |
| DispatcherServlet | need to configure in one xml file | Automatically done by the web library |
| Bean object creations | need to configure in one xml | Automatically one by the spring boot |
| Server | need to configure | Automatically provided in the web library |
| Migration | need to change the version of each libraries | need to change only the parent version and automatically other libraries follow |

First Spring Boot Project

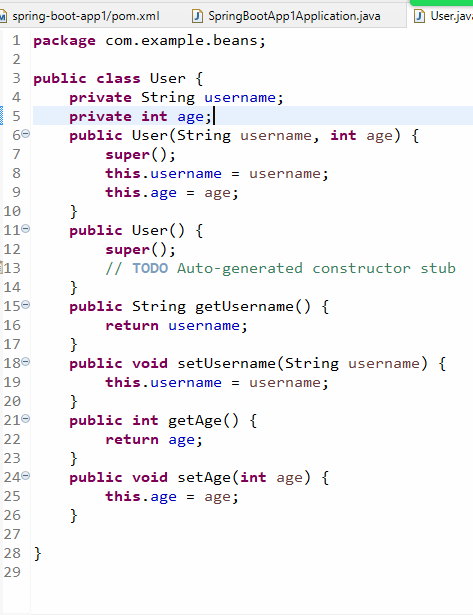


SpringBootApp1Application.java

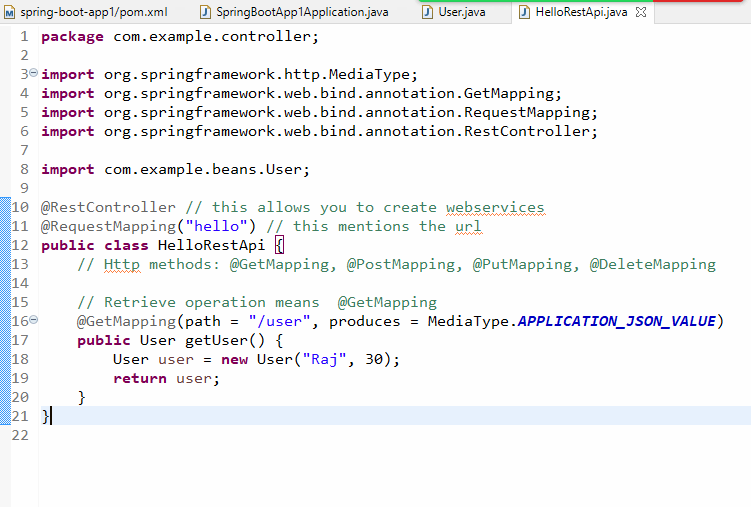


@SpringBootAppication: It auto-configures the server, scans all the classes having @Repository, @Service, @RestController, @Controller, @Configuration, @Aspect, and so on and registers in a spring container without having an xml file, it configures the application based on the libraries in the class path

User.java



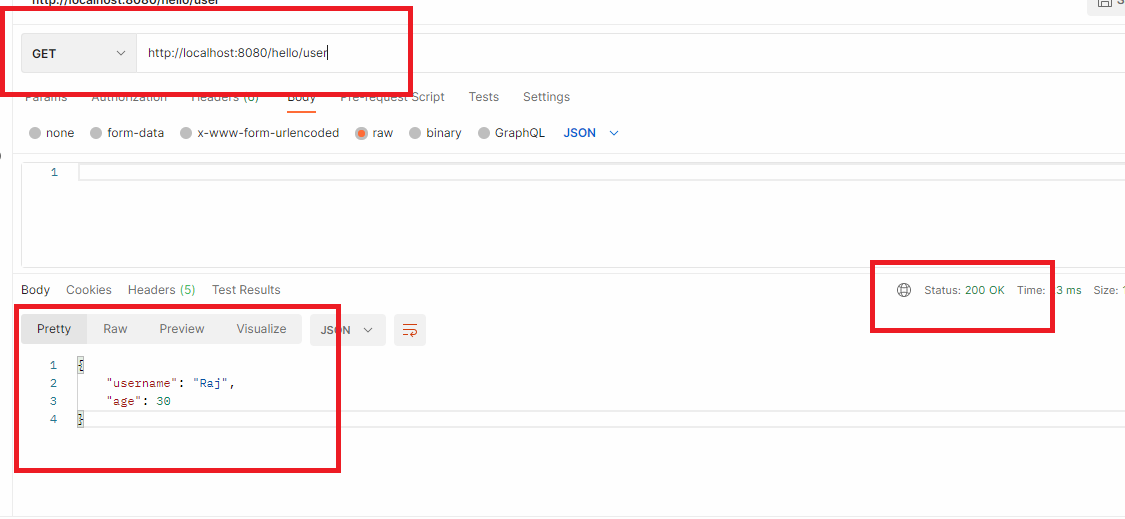
HelloRestApi.java



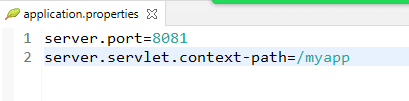
@RestController: It is an annotation used to create webservices

@GetMapping: Represents the operation is fetching

Postman output:

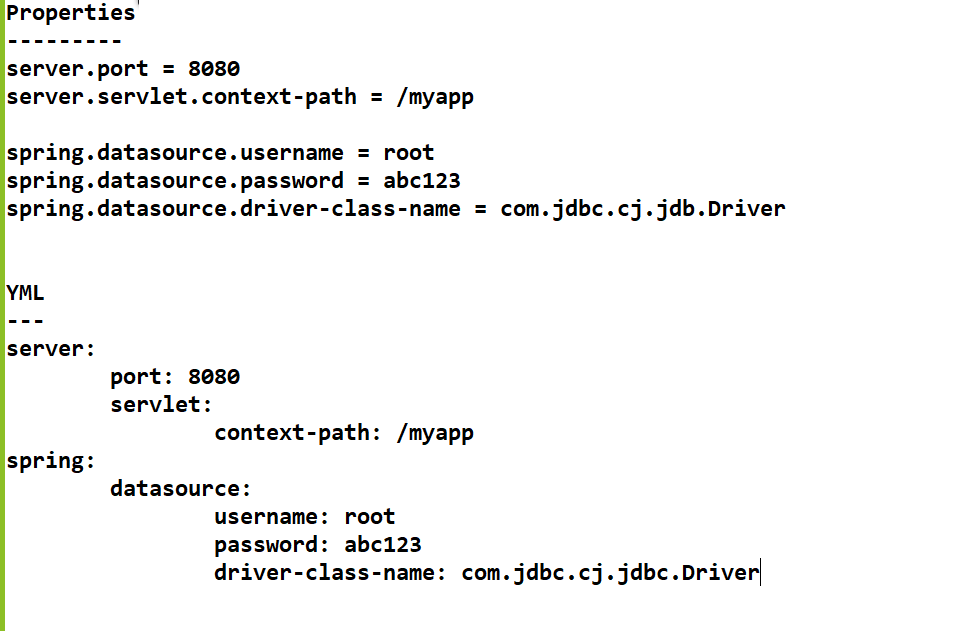


application.properties



Here the context path will be /myapp, so that your webservice needs to be accessed by <http://localhost:8081/myapp/hello/user>

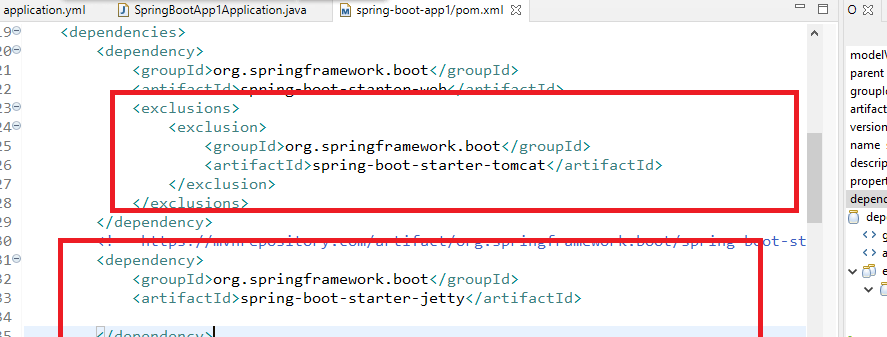
application.yml



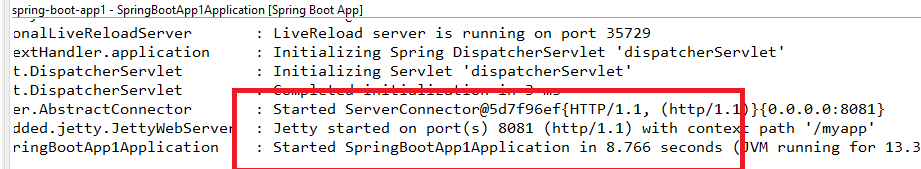
The spring boot application can be run on any JRE machine even in cloud machines also, you need to create an executable jar file for it

You can add other open source servers like jetty or undertow in spring boot apart from tomcat.

Since by default tomcat is used, you can exclude tomcat and add either jetty or undertow.

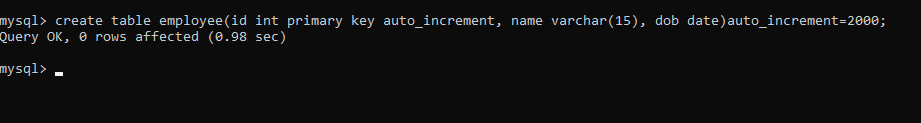


You can see in the console log the jetty server status



How to interact with the database

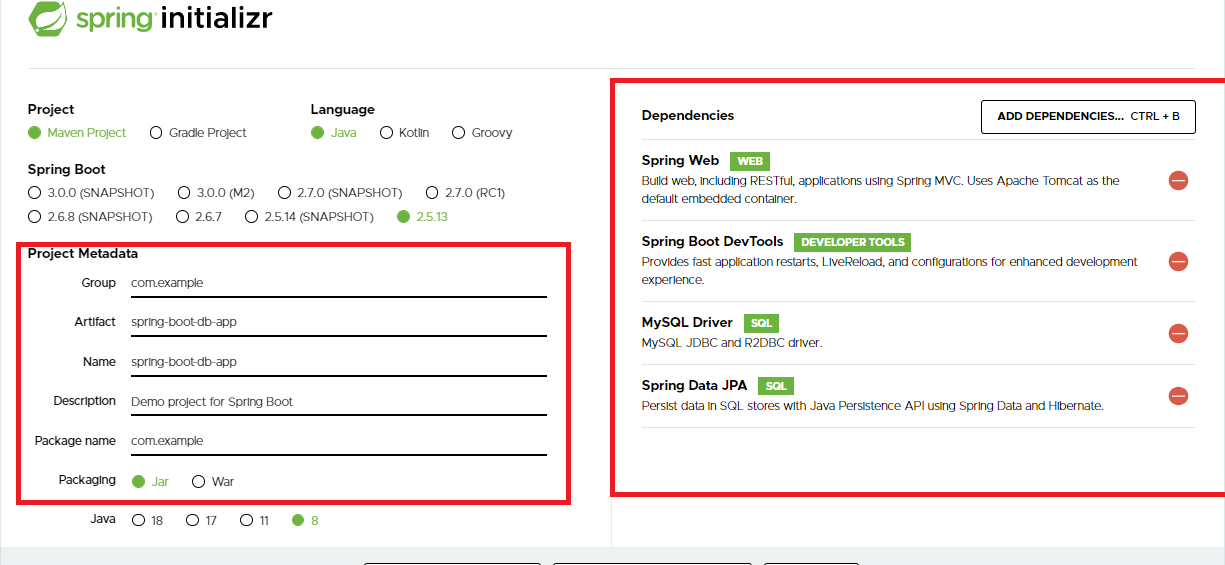
* You must add the database dependency like H2, MySQL, Derby so on
* You must create the class that maps to the database i.e., entity class
* You must configure the datasource in the application.yml
* You must use spring data jpa library to easily perform CRUD operation without any implementation



Now the application must have 3 layers

1. Controller: To have webservices takes care of converting JSON to Java and vice versa
2. Business layer/ service layer: To have business logics
3. DAO layer: This will have database logics

Note: In Spring Boot you don’t have to write any implementation to the database logics, because you get a proxy object to do the job for you, this proxy object needs which entity class you are mapping to which table, then it provides all the necessary methods to perform CRUD operations.



Spring Data Jpa: It takes care of implementing the database logics according to the configuration done in the repository interfaces, it takes care of establishing connections with the database as per the application.yml configuration

JpaRepository: It is an interface with all the CRUD operations like save(), delete(), findById() and so on, all these methods are implemented by proxy object at runtime based on the entity class mentioned

Entity Class: it is a class that maps to the table, suppose Employee is a class then it can be mapped to employee table in Mysql or anyother database

interface EmployeeDao extends JpaRepository<Employee> { } : This is an interface that will get all the methods of JpaRepository and all the methods will interact with Employee table because Employee class is an entity class maps to Employee table

Custom methods: These methods you can create in your dao interface which can perform some other operations on the table with the JPQL query

interface EmployeeDao extends JpaRepository<Employee> {   
 @Query(“select e from Employee e where e.name = ?1”)  
 public List<Employee> findEmployeesBasedOnName(String name);  
}

If you call findEmployeesBasedOnName() by passing some name then it gets all the employees having that name in List<Employee>

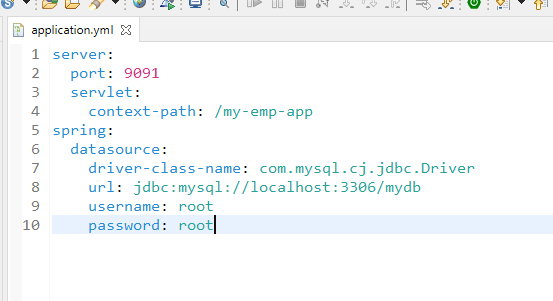
Whenever there are multiple layers in the code like DAO, Service, Controller then we need to know that service code calls dao code and controller code calls service code, means service depends on dao and controller depends on service.

Note: Spring Boot automatically takes care of injecting the dependencies like DAO will be injected to Service & Service will be injected to Controller.

Things we need to create to interact with Employee table from Spring Boot

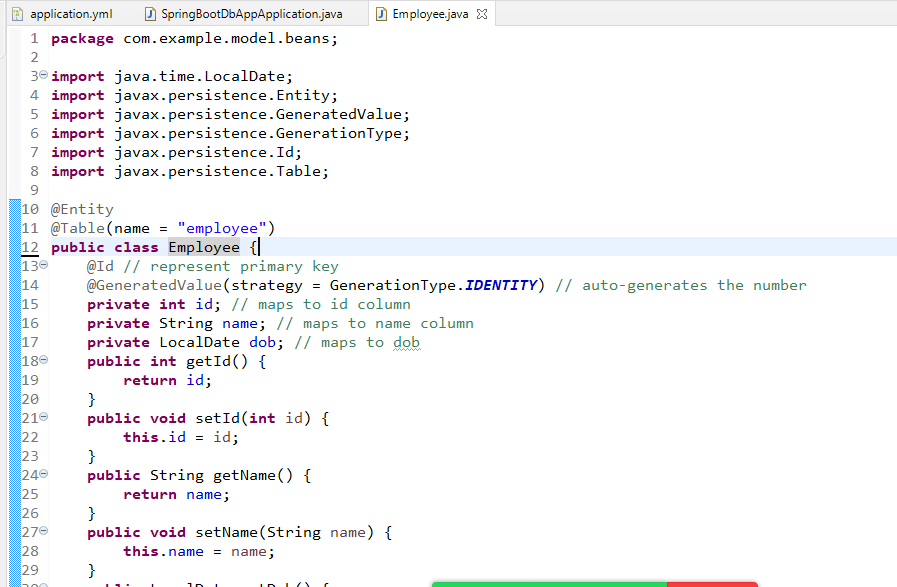
1. Employee class: Entity class maps to Employee table
2. EmployeeDao: Interface that extends JpaRepository<Employee>
3. EmployeeService interface: All the methods to perform business logics
4. EmployeeServiceImpl: All the methods of service must be implemented like transactions, other business tasks
5. EmployeeRest: Webservices
6. EmployeeNotFoundException: Exception that needs to be generated if employee id is wrong

application.yml

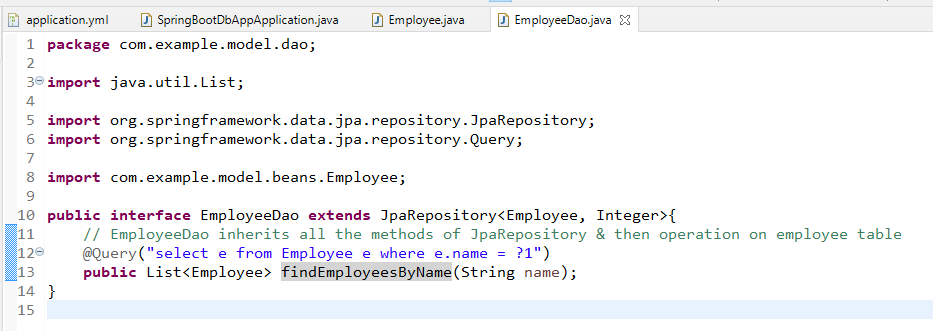


The above configuration helps spring boot to automatically connect to the database & create all the beans required to interact with the database.

Employee.java



EmployeeDao.java



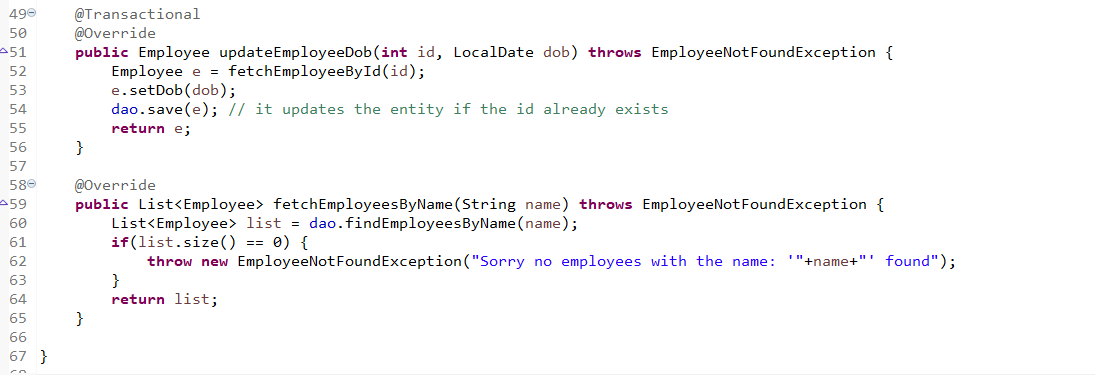
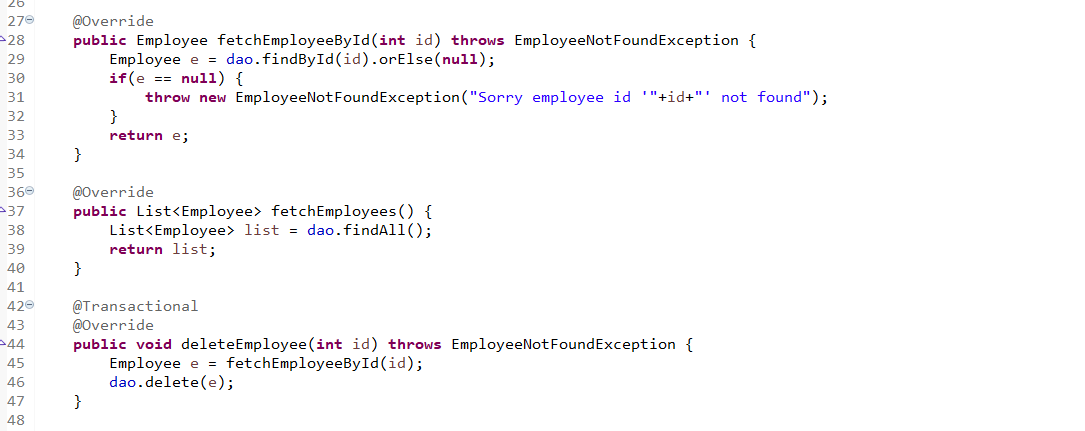
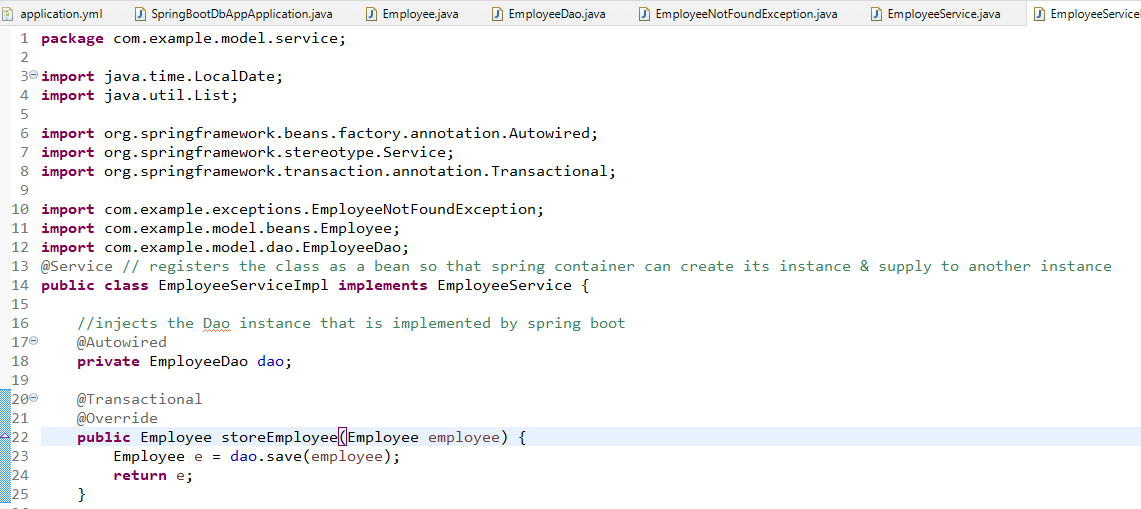
EmployeeNotFoundException.java



EmployeeService.java



EmployeeServiceImpl.java



@Transactional: It is used when you do any modification

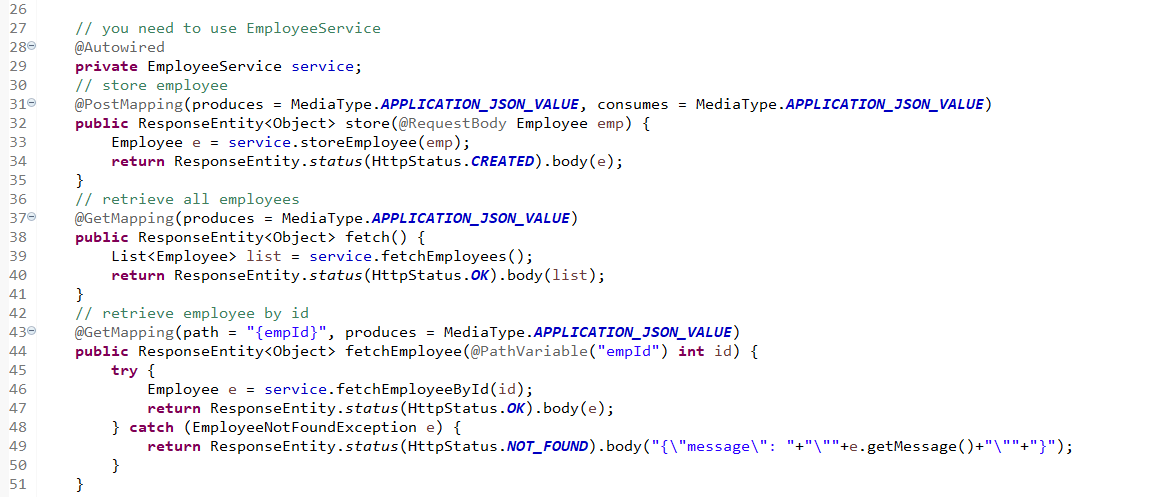
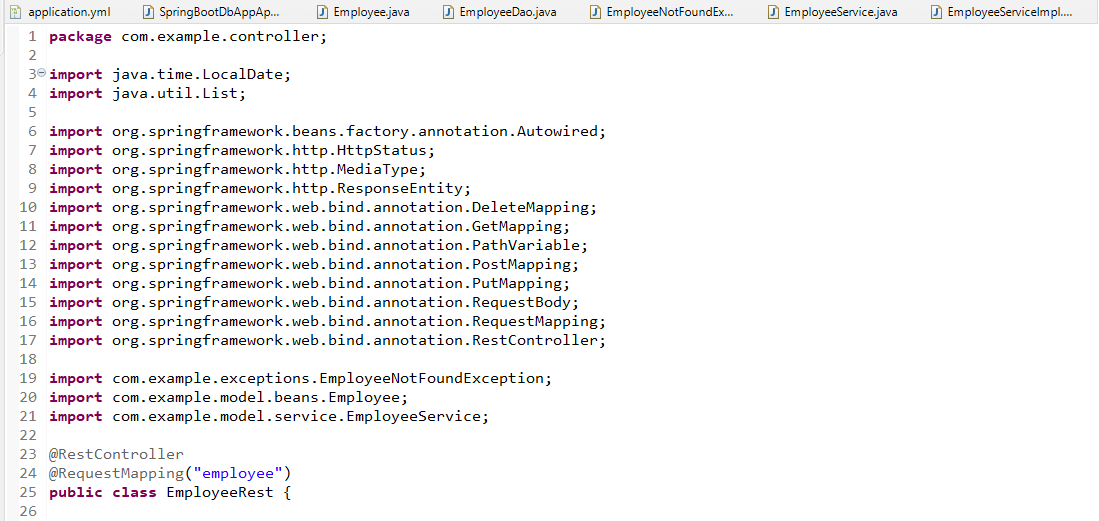
You can now implement Rest Apis to perform CRUD operations, here you need to use POST for new data , PUT for updating existing data, GET for fetching data, DELETE to delete the existing data

You can use ResponseEntity<T> as a return type in webservice, so that you can customize the response to have different status code & the contents

ResponseEntity.status(404).body(object);   
ResponseEntity.status(401).body(object);  
ResponseEntity.status(201).body(object);

Here the object in the body is sent to the consumer and consumer can use status code to decide his operations.

EmployeeRest.java



You can use MongoRepository to interact with MongoDB database, mongodb stores data in NoSQL format i.e., javascript object it stores

Activity:

1. Try the above activity on employee table
2. Create an address table & use a foreign key in the address table that will have employee id, in the address table you can have state, city & pincode
   1. When you store employee object you also need to update address table
   2. When you retrieve object you must able to retrieve address of that employee as well.

Hint: Use @OneToOne annotation, create an address reference in the Employee class