Spring Boot

It is a module in the spring framework which needs minimum configurations i.e, no XML file required at all.

It performs auto-configurations based on the libraries you have in your classpath

Spring boot released some of the starter libraries to perform these auto-configurations, some of them are:

1. spring-boot-starter-web
2. spring-boot-starter-aop
3. spring-boot-starter-jpa
4. spring-boot-starter-actuator
5. spring-boot-starter-devtools

These starters takes care of auto-configurations for your applications.

i.e., starter-web will configure a server, dispatcher servlet, view resolver, annotation based configurations, component scanning

starter-jpa will configure all the dependencies required for the databases and scans all the entities, however you need to provide datasource informations in the application.properties

Resolving Version mismatch

Spring boot has a *spring-boot-starter-parent* POM that has all the list of compatible starters according to the spring boot version

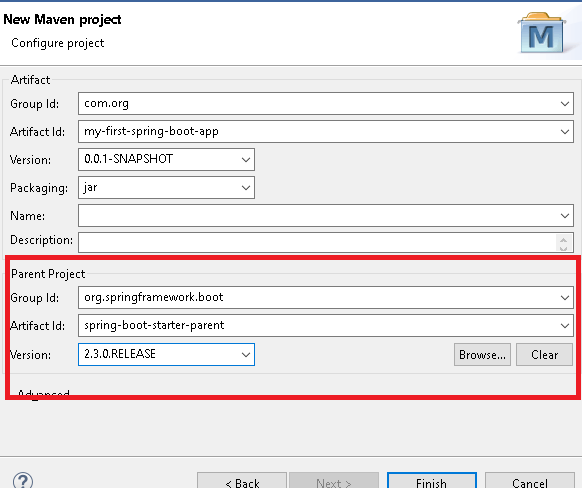
i.e., if you add spring-boot-starter-parent version 2.3 then any starters of spring boot like web, or jpa doesn’t need to mention the version number, because spring boot pulls the compatible version of web or jpa as per the starter-parent.

Note: Spring boot can be done only through either Maven or Gradle

There are two ways you can create project to use spring-boot

1. Maven project with spring boot parent starter
2. Maven project downloading from spring initializr (better)

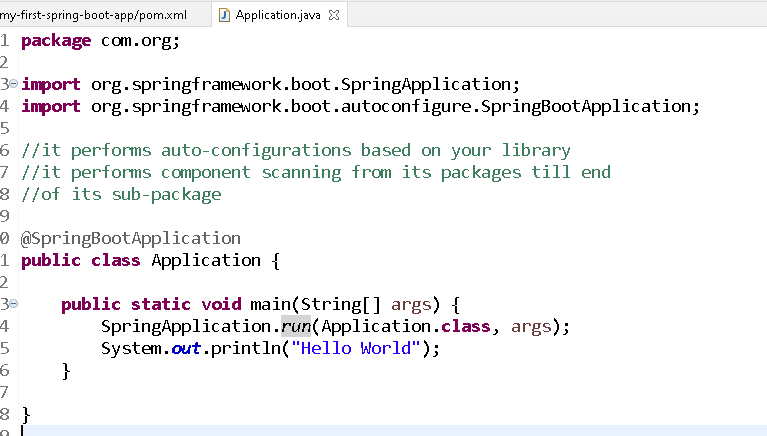
Maven project by manually selecting the spring boot starter parent and mentioning some dependencies



pom.xml



Application.java



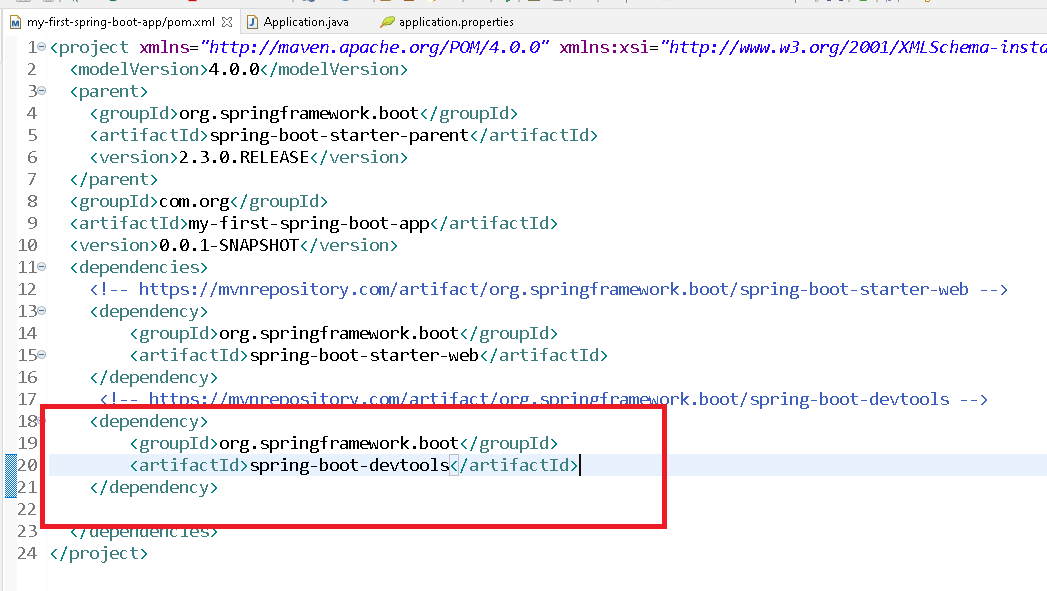
@SpringBootApplication will do auto-configurations like

* stating a tomcat server
* component scanning from its root package
* configuring dispatcher servlet
* reading application.properties & configuring the application

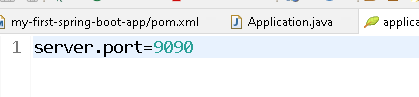
application.properties file should be created inside src/main/resources folder, however you can have yml file instead of properties i.e, application.yml

You can add spring boot devtools to automatically reload your application on server

pom.xml



create application.properties inside the src/main/resources



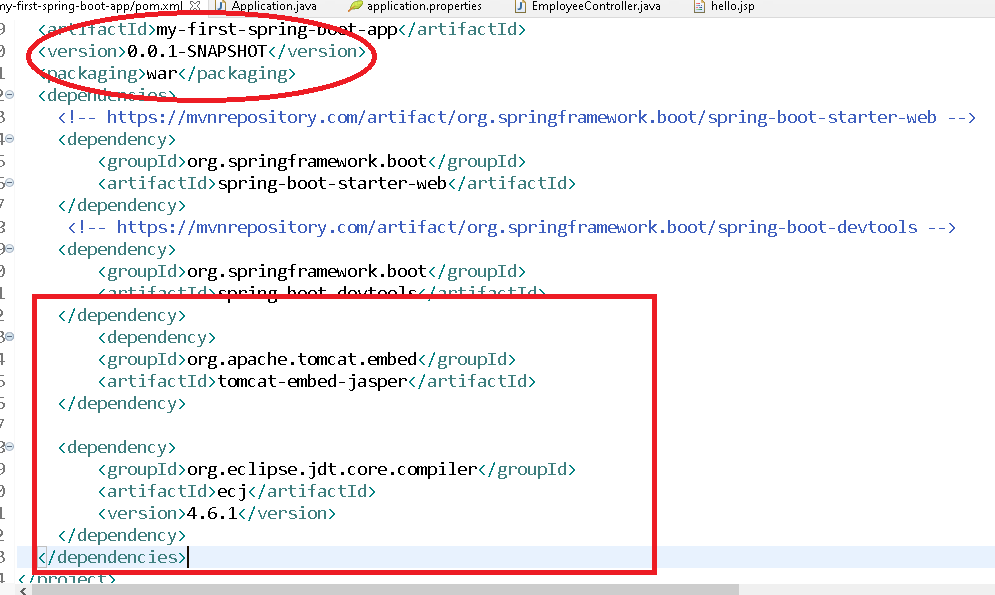
You can create a controller using @Controller because of auto-configuration feature DispatcherServlet will be configured and you can use controller directly.

Adding JSP files to your spring boot project needs lot of configurations.

In pom.xml you add two dependencies and also change the <package> to war

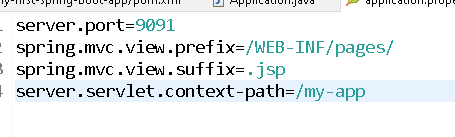
1. tomcat embedded jasper
2. ecj

pom.xml



Add view resolver properties in pom.xml

application.properties



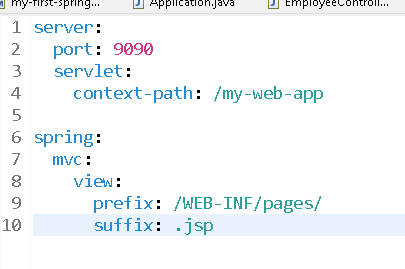
Create a controller



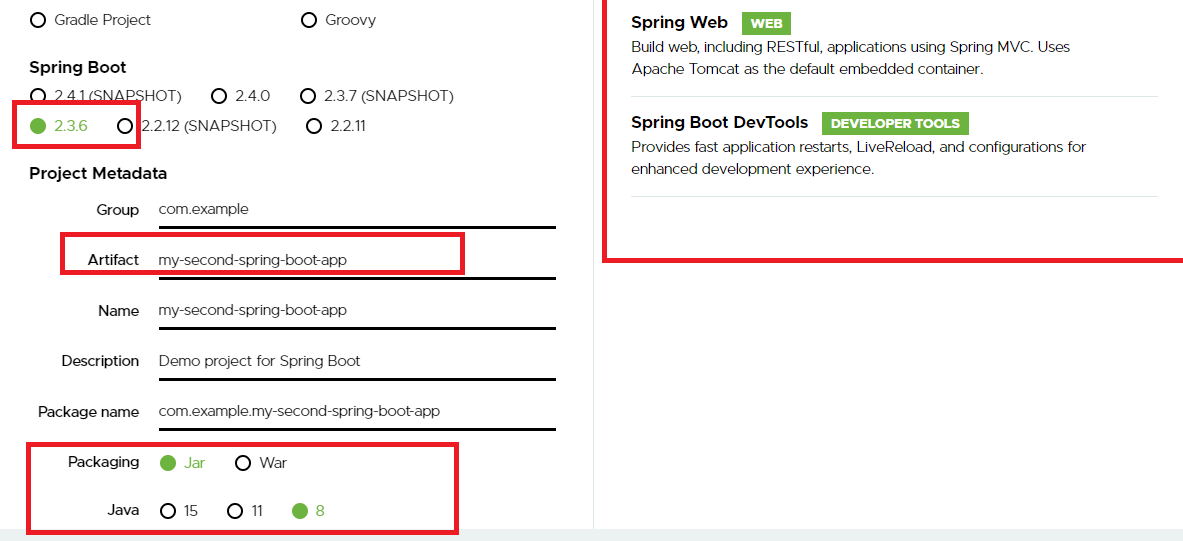
You can use yml files instead of properties because it avoids you using repeatable properties.

Note: yml files use indentation to recognize the nested properties

Delete the application.properties & create application.yml



Creating the project using spring initializr



When you use spring initializr you will get a Java file that launches the spring boot application and also an application.properties where you can configure the application configurations like port, server address, datasource, context-paths, actuator informations, you can create yml file by deleting the properties file.

How to run/launch the application in the production

Inorder to launch the application in the production you need a deployable war/jar, since you have created project through spring initlializr it gives you deployable jar/war.

You need to use mvn commands to build a jar and run the jar

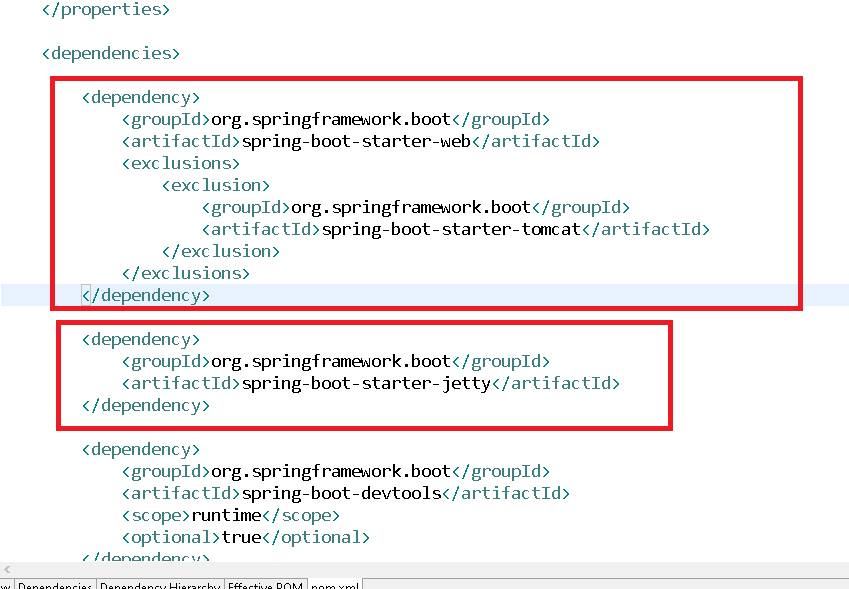
mvn clean: deletes the target directory

mvn package: compiles and creates the executable file

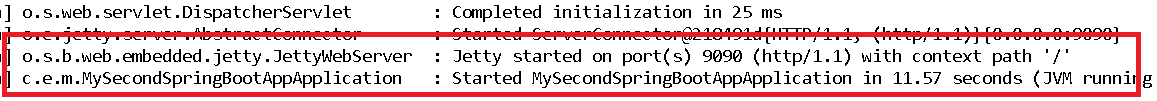
java -jar target/file.jar --server.port=9093

Note: All these commands you need to enter from the project locations

You can add different servers other than tomcat like jetty server, but you must exclude the default tomcat server in the web dependency

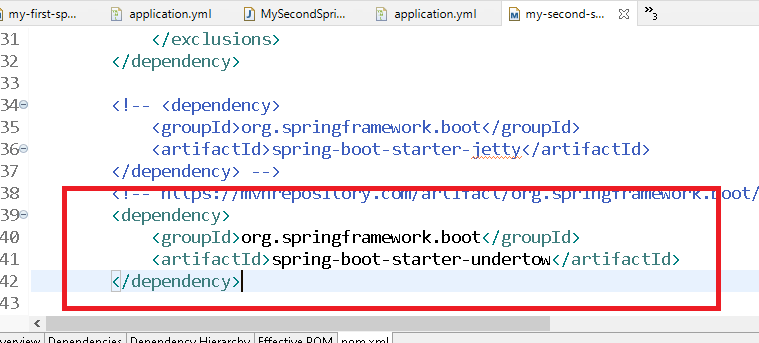


Output:

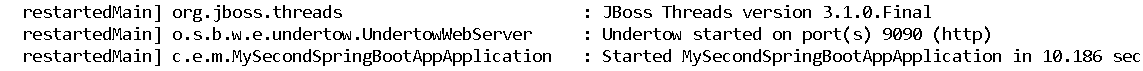


You can also use undertow servlet container

It is servlet container form jboss



Output:



Building webservices in Spring Boot

Webservices: These are online services that are exposed for other applications to call

ex:

1. Banking applications expose checkBalance, debit, credit functionalities to the applications like google pay, phone pay, paytm
2. Google map has exposed its location services to the applications like Uber, Ola, Zomato and also to the public customers

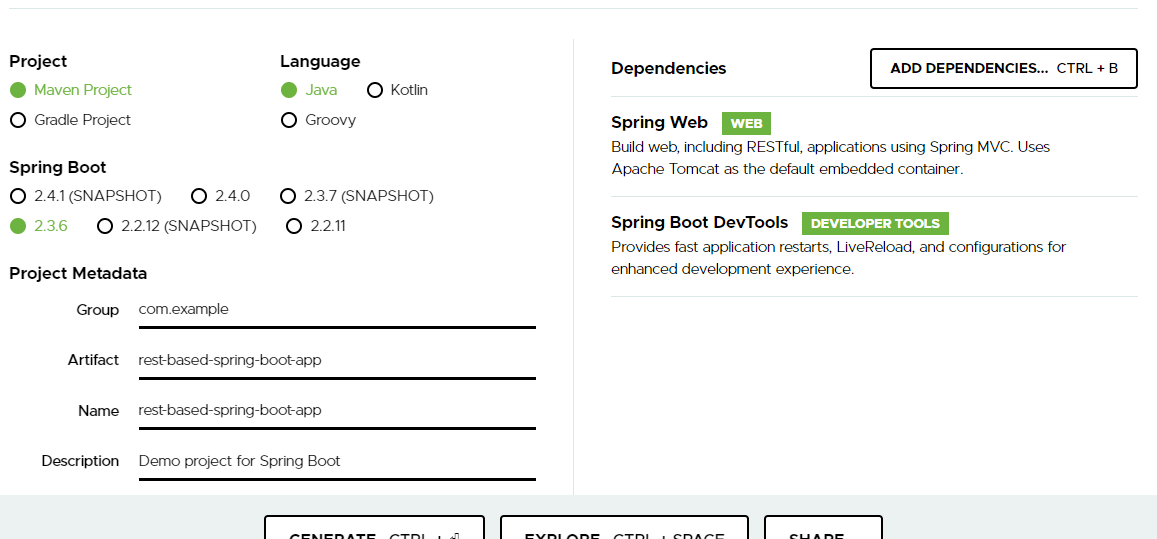
How to build webservices

1. REST stands for REpresentational State Transfer, where you exchange the state(data) in JSON/XML/Text Representations and transfer between the each other applications
2. You must have a class with @RestController
3. Inside the RestController you must have methods that can handle different type of http request and URI
   1. GET
   2. POST
   3. PUT
   4. DELETE

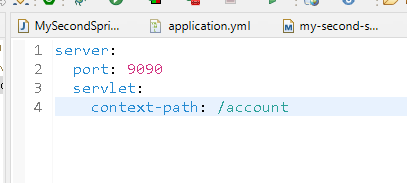
Client applications

These are the applications which will have some programs to call the webservices and convert the response of webservices to the format they can understand.

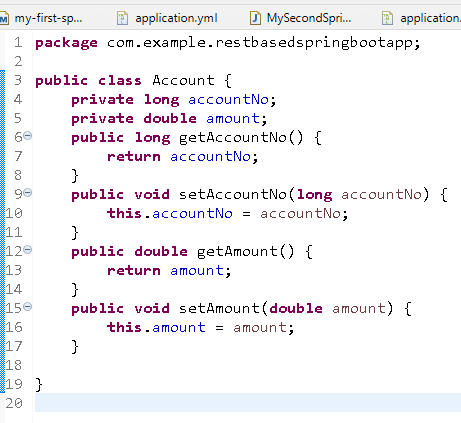
ex: Google Pay is a client to many banks, Google pay takes care of converting JSON to the language Google pay is written in similar way what google pay sends to banks is also converted to the language the banks is written in



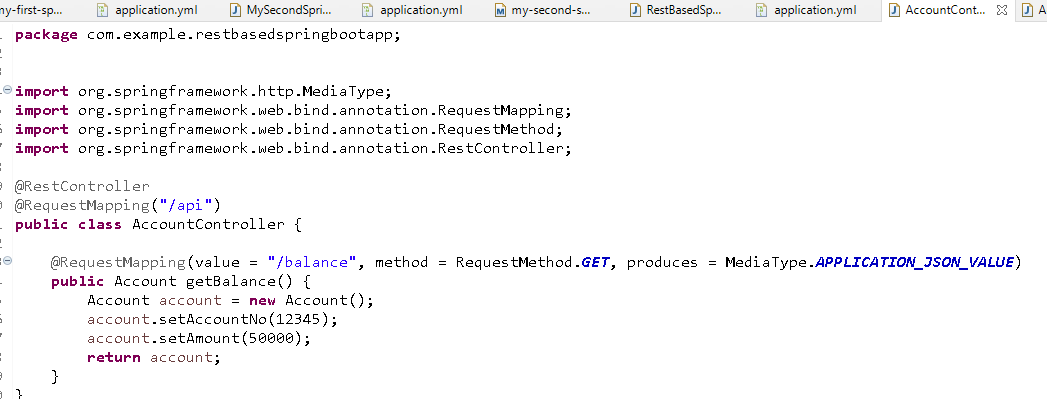
application.yml



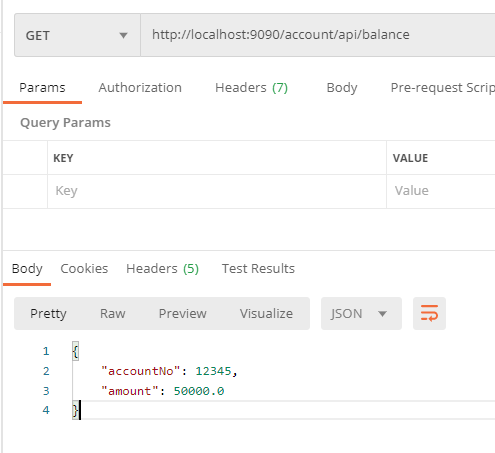
Account.java



AccountController.java

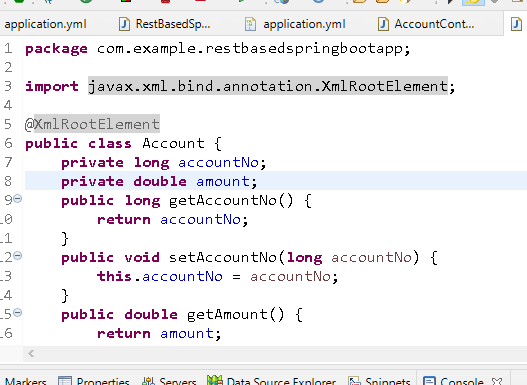


Output:

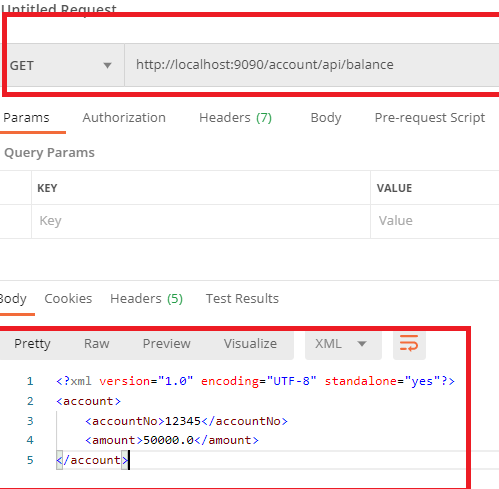


XML response: It is not a default response hence you need to use @XmlRootElement on top the java class that needs xml representation

Account.java



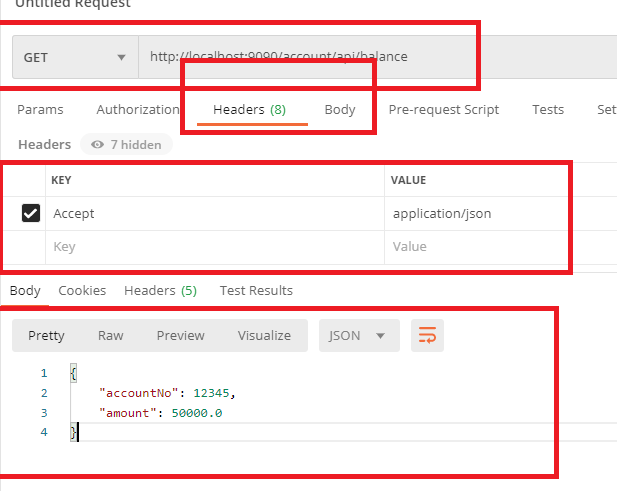
Output:



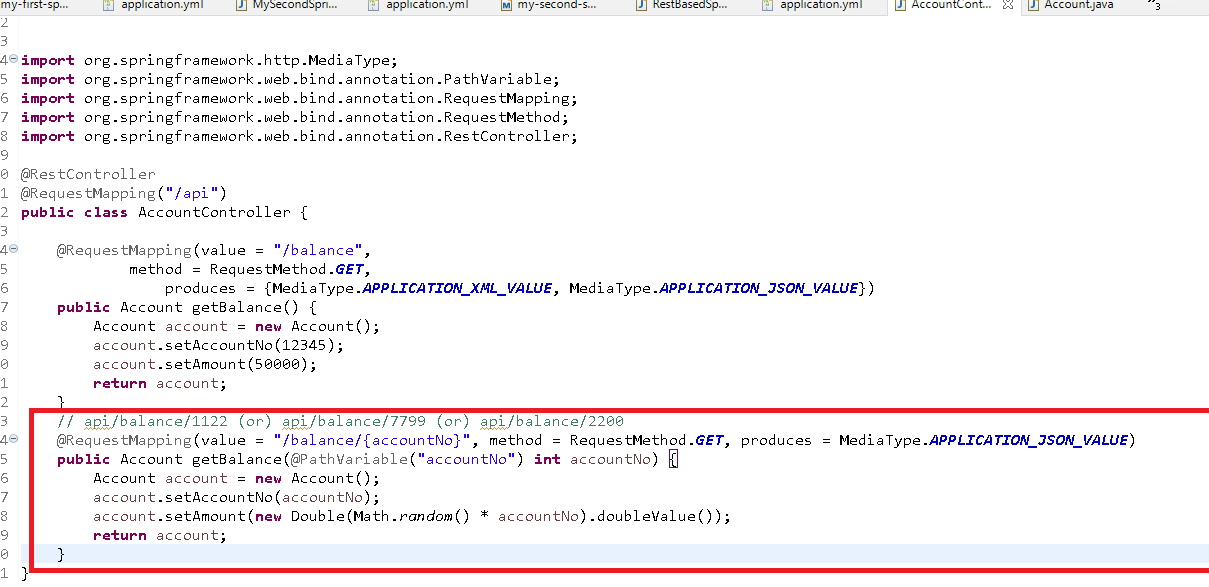
You can send the response in multiple formats

However the client needs to mention in its request headers what is the response it needs using accept-type

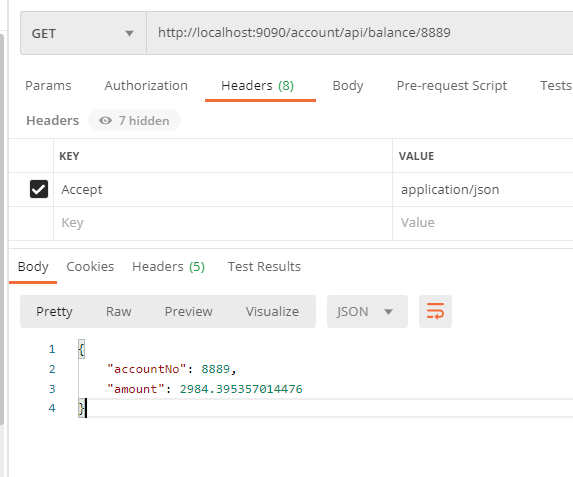
Content Negotiation: Client mentions in the request header what format it is expecting



@PathVariable: it can extract the inputs from the URI



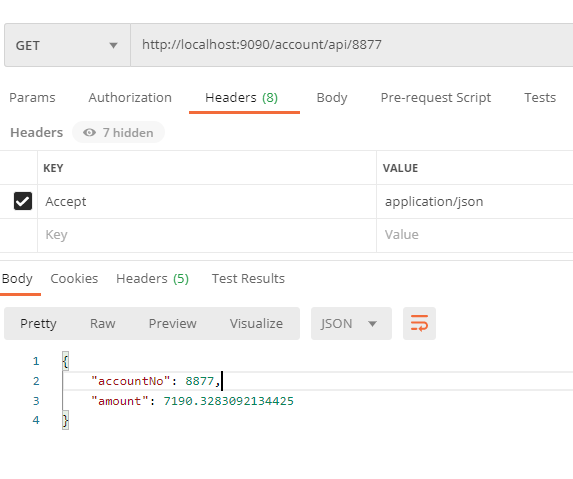
Output:

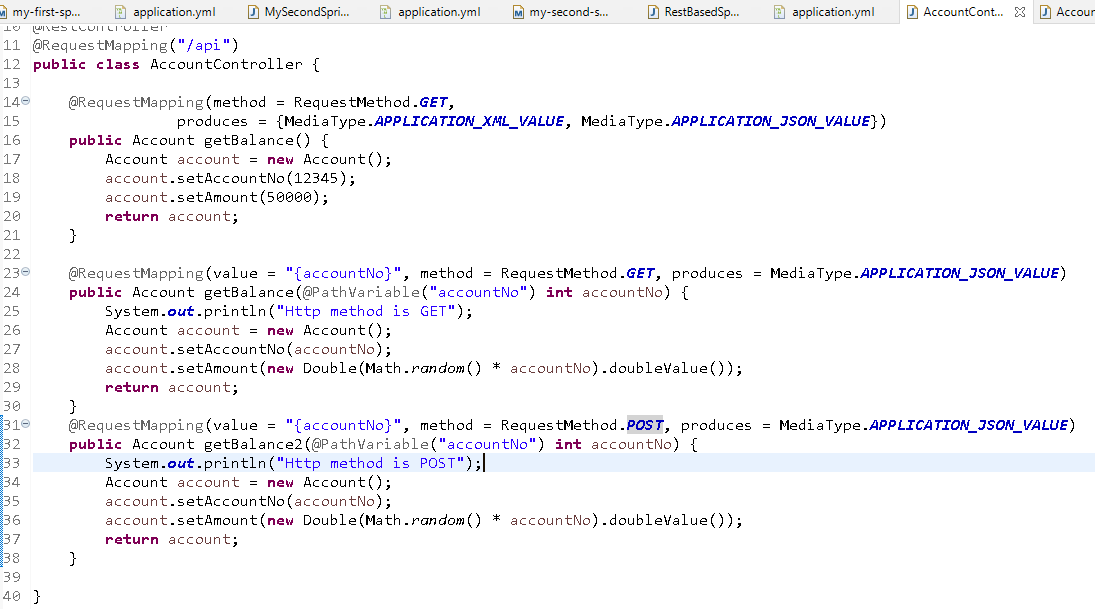


You can use common uir’s for multiple webservices but it must have different HTTP methods

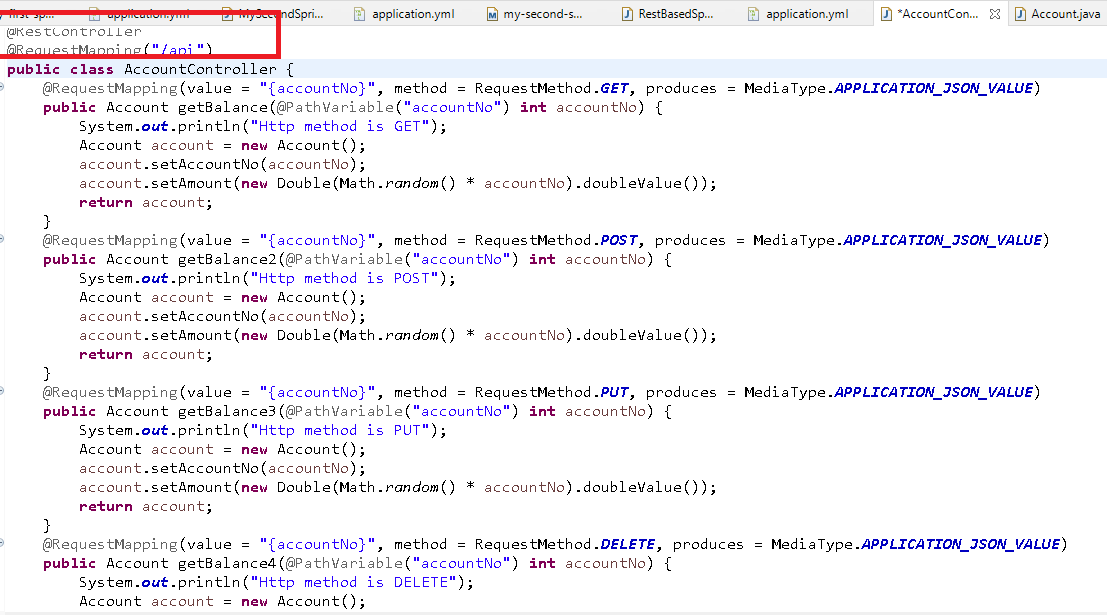


Output:





Above webservices can have same URL’s but different HTTP methods so that client can use different HTTP methods for same URL



Above webservices take same URI’s but different HTTP methods

HTTP methods are useful like contract between the client & the server

1. GET: fetching the resource
2. POST: creating the new resource
3. PUT: updating the existing resource
4. DELETE: deleting the existing resource

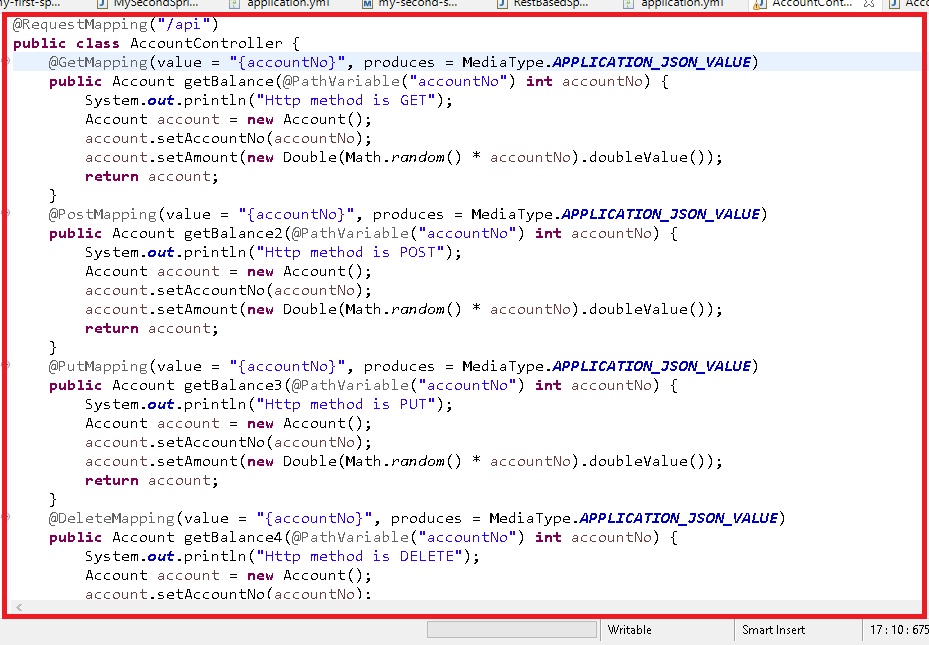
Spring Boot also gives annotations for different HTTP methods

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping



Sending data in the request body, you can send data in the request body through POST, PUT, DELETE but not through GET.

application.yml

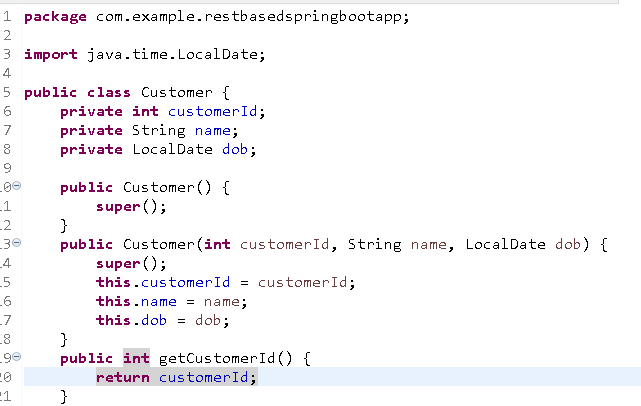
server:

port: 9090

servlet:

context-path: /banking-app

Customer.java

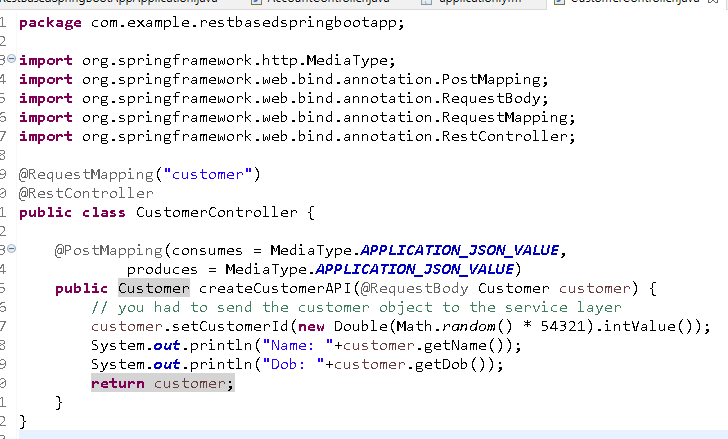


Client sends the data in the request body so that the Webservice should consume it using @RequestBody which extracts the data present in the request body, @RequestBody takes care of mapping each properties of XML/JSON to appropriate Java object properties

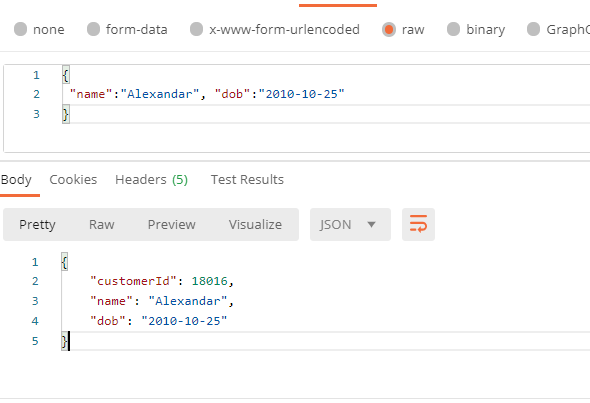
i.e., {“name”:”Alex”, “dob”:”2010-11-22”}

The corresponding class must have variables name & dob then only @RequestBody can map JSON values Alex to Java variables name and 2010-11-22 to dob

CustomerController.java



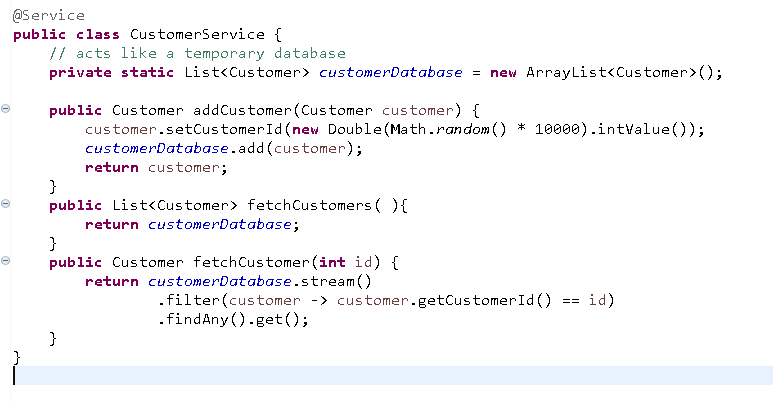
Output:



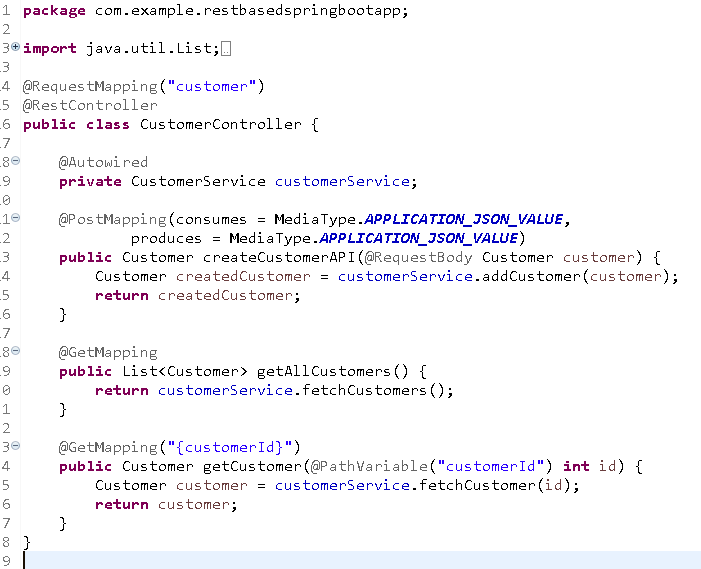
CURD operations without database

We will have a service layer with List<Customer> and maintain the customer items, later we will change this with Dao layer

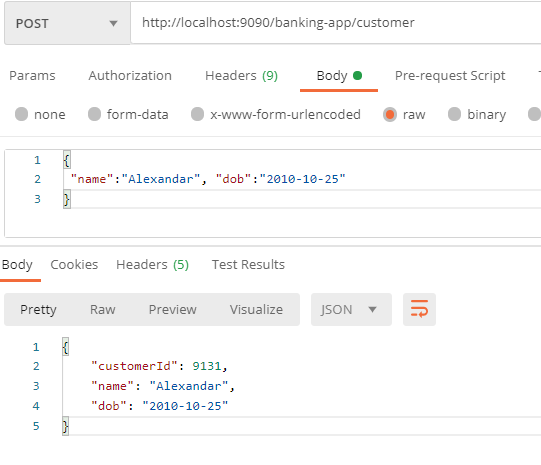
CustomerService.java



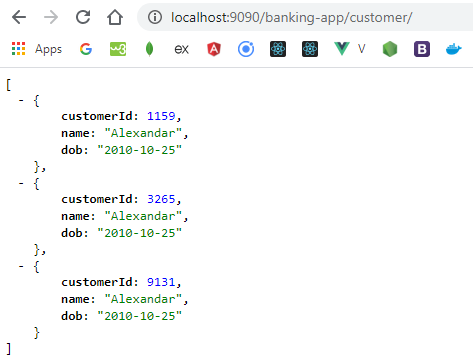
Now you can create URI’s in the controller for all these 3 methods



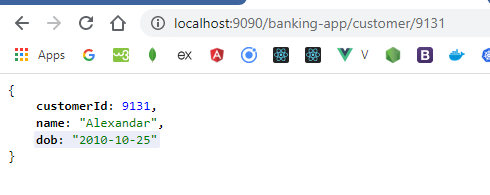
Output:



GET request: /banking-app/customer



GET request: banking-app/customer/9131



Spring Data Jpa:

It simplifies performing operation on the database by providing some inbuilt repositories which are interfaces that work on the provided entities.

You need to just extend these interfaces and call the methods present in it.

Note: These interfaces are automatically implemented by your spring boot based on the methods you call on the interface as it would know you are performing operation on which entity.

In Spring Data JPA, the datasource information’s are provided in property files & table informations are provided in Entity class, hence you just need to extend the Repositories provided by Spring Data Jpa, without implementing DAO layer you can perform operations because of auto-implementation done by spring-boot.

Some of the inbuilt repository interfaces

CurdRepository<T, ID>:

* save(T): saves the entity and returns the saved entity, it will also update the entity
* findAll(): returns all the entities in a List<T>
* findById(ID): returns the entity matching to the ID
* delete(T): deletes the entity

JpaRepository<T,ID> : it extends CurdRepository

* sort(..):
* pagination(..)

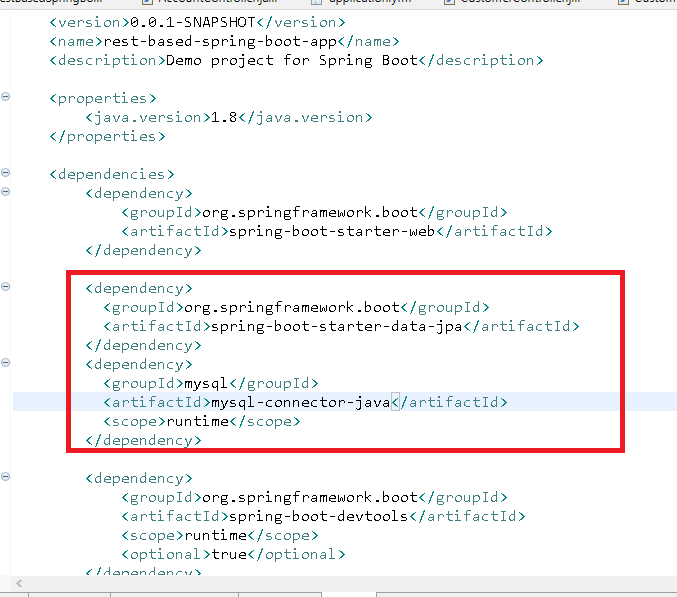
You just need to extend any one of these interface and use this interface reference @Autowired in the Service layer and spring-boot supplies the implementation.

Note: You don’t need to implement DAO layer, it means DAO layer is just an interface.

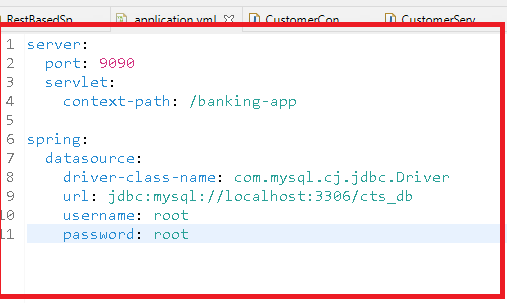
Dependencies required:

* Spring Data Jpa
* Mysql Connector
* Web
* Devtools

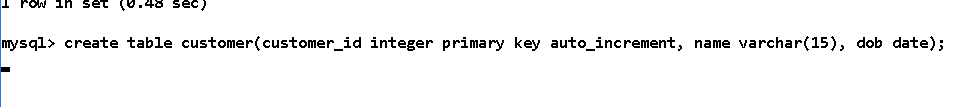
pom.xml



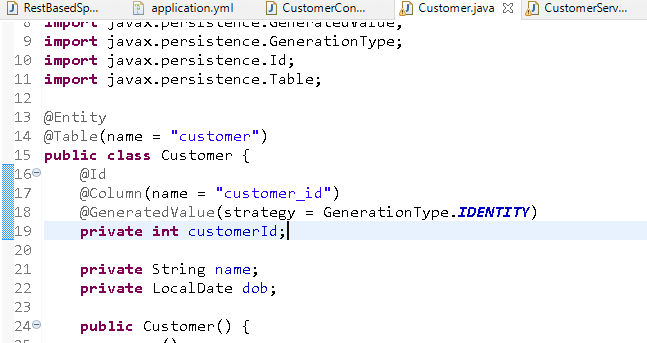
application.yml



Entity class mapping the Table



Entity class



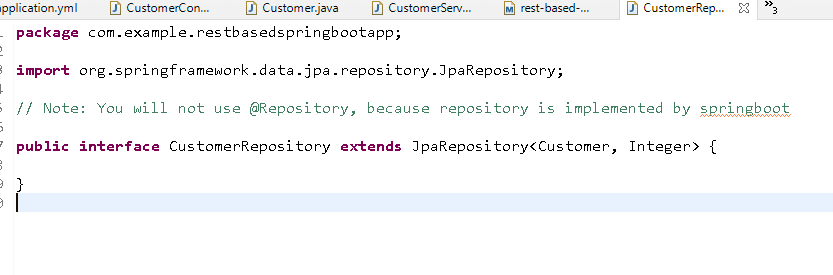
Creating a repository that extends JpaRepository<T, ID>

T: entity class type

ID: primary key type

Since Customer is the entity and Integer is the primary key we will use

CustomerRepostory.java



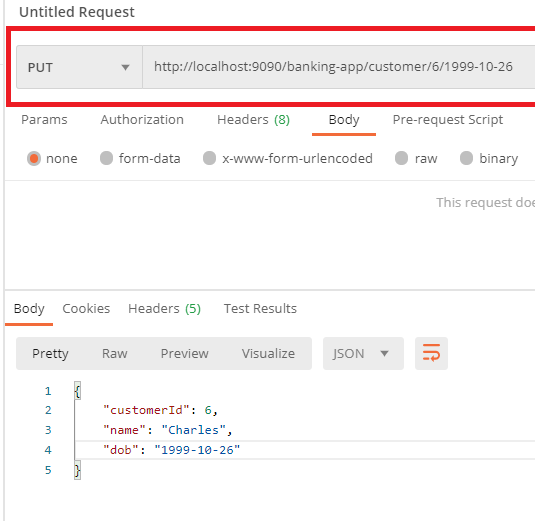
CustomerService.java



CustomerController.java



Output:



Exercise:

1. Perform the delete operation based on the id
2. Structure the packages according to the different mvc layers
   1. Controllers in controller package
   2. Service in service package
   3. Repositories in dao package

Note: All these package must be under root package of @SpringBootApplication class

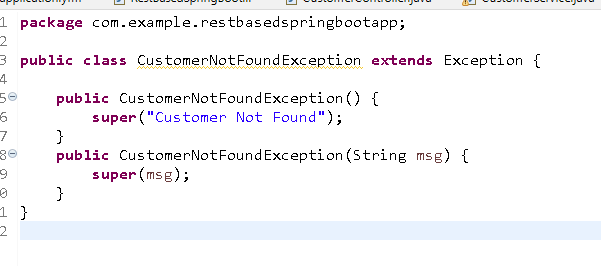
Note: Push the code to your forked repository (cts-hands-on) once done, have this structure in your repository

/employee-id-name/Morning-Batch/Spring Boot Exercise/your-exercise

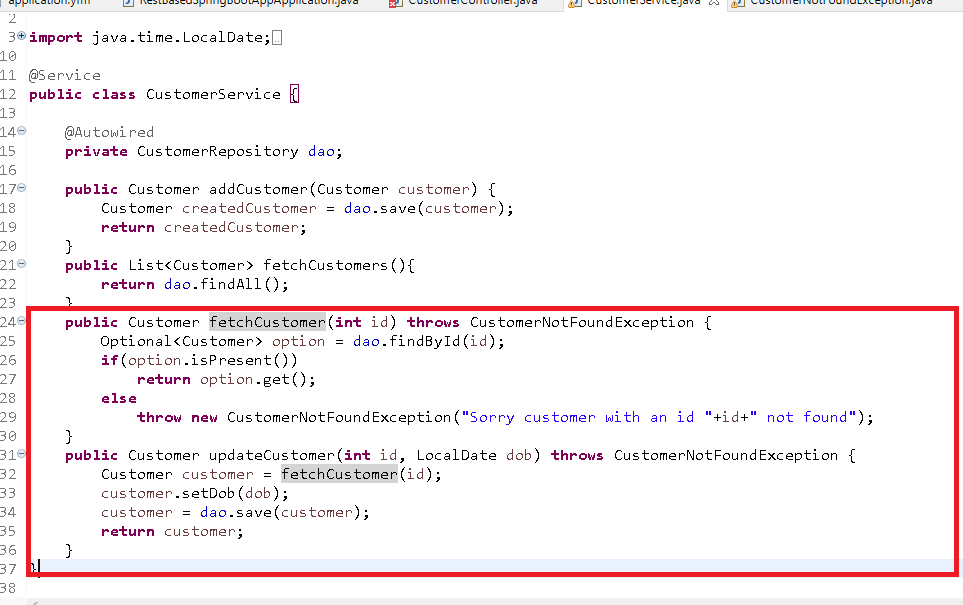
ResponseEntity: This is an instance that stores the response of the Webservice, you can change the status code and add different content to the response.

Let us try to return error message if there’s an exception while perform some operations like CustomerNotFoundException and with the help of ResponseEntity you can change the response content.

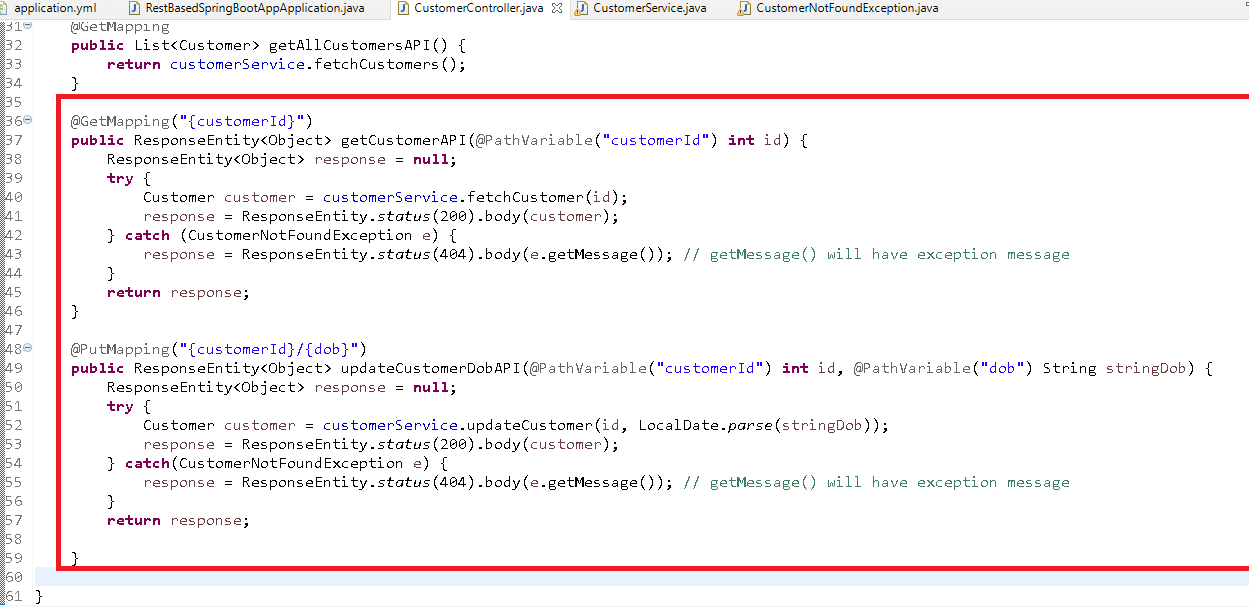
CustomerNotFoundException.java



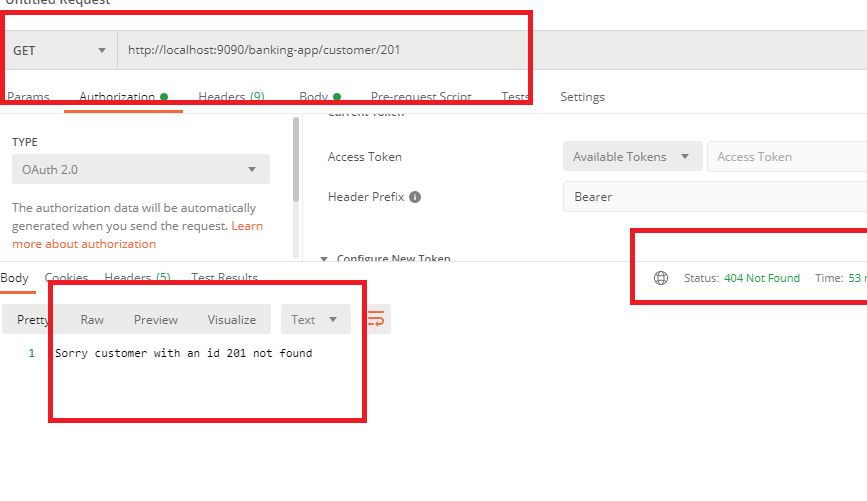
Service layer can throw CustomerNotFoundException



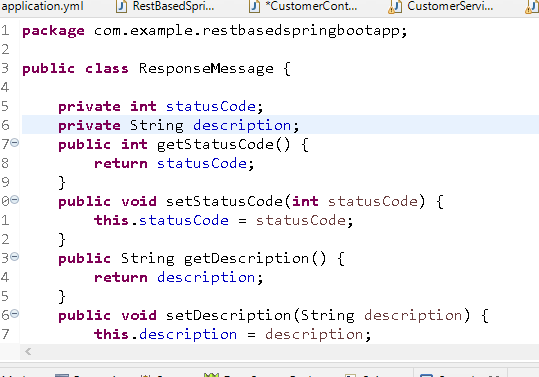
Since controller is calling fetchCustomer & updateCustomer it has to handle the exception and provide a different response to the client



Note: Now you don’t get an empty response if the customer id is not present, you will get the exception message with 404 status code.



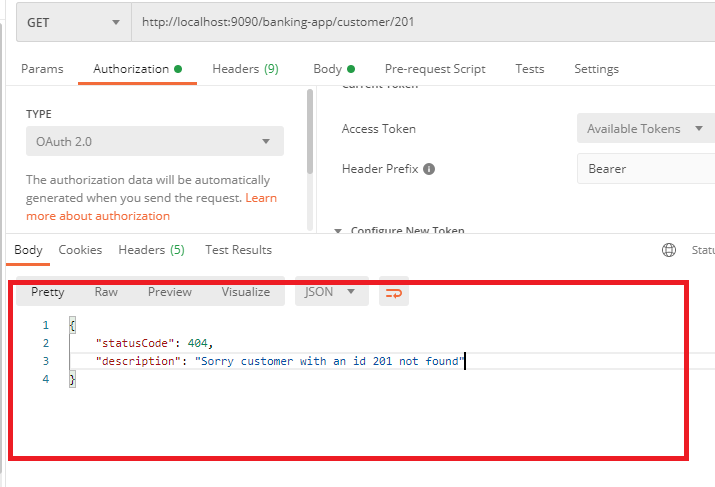
Now you can see the error response is having 404 status code & content is text, however you can have error content also in JSON structure.



Now you can pass the exception message to the description and response status to the statusCode so that when you pass the object the client gets json structure



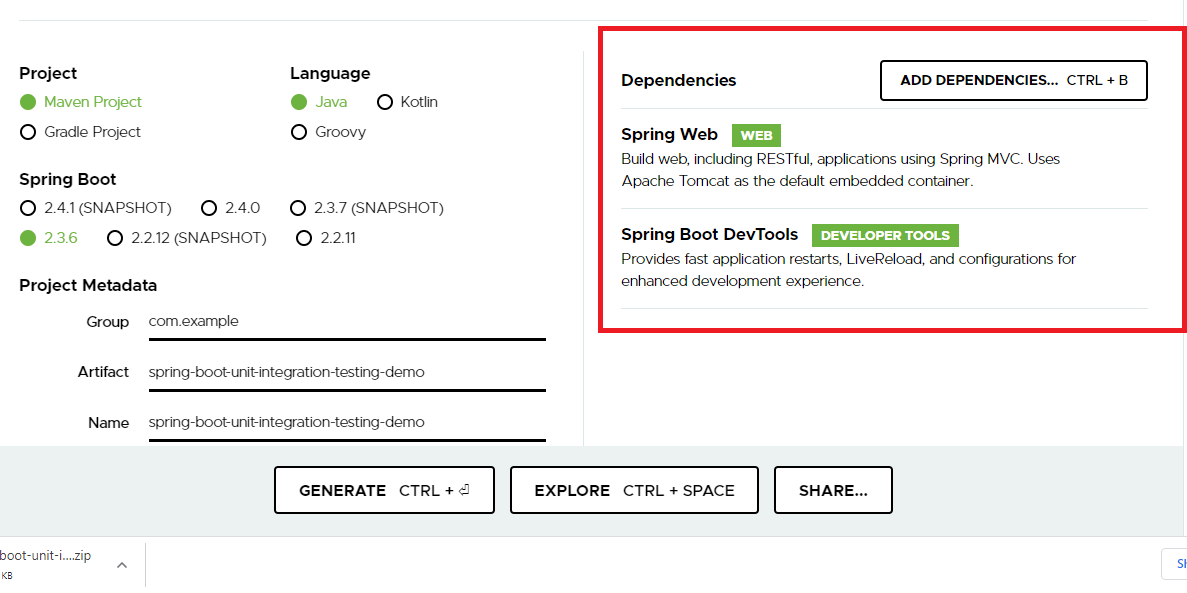
Output:



Exercise:

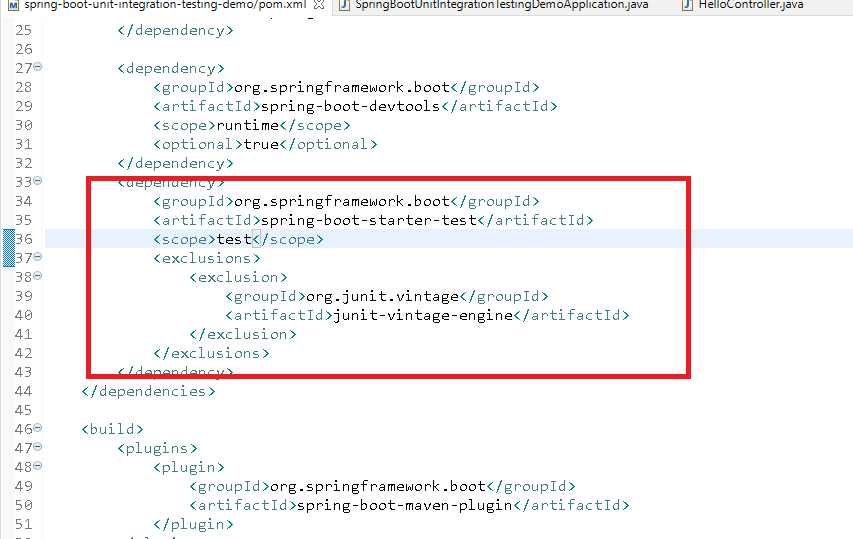
* Change the return type of restful controller to ResponseEntity
* Create exception class in a separate package exception
* Create ResponseMessage in util package

Spring Boot Unit Testing & Integration Testing



Note: By default test dependencies are added you don’t have to add them explicitly

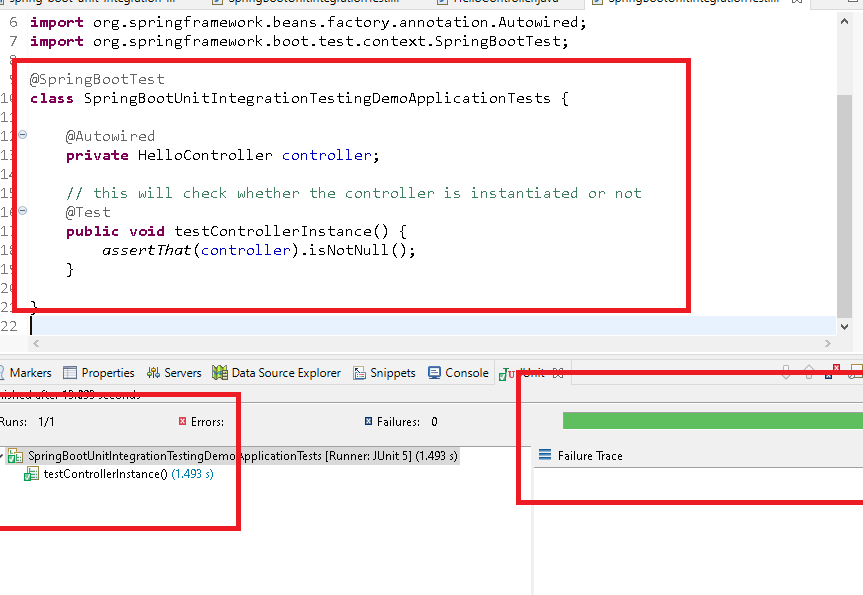
pom.xml



You can create one simple method inside the controller and test it without running the actual server

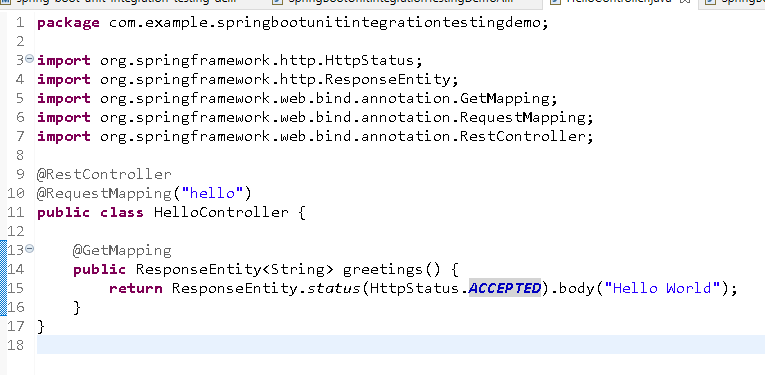


Write your test cases inside src/test folder

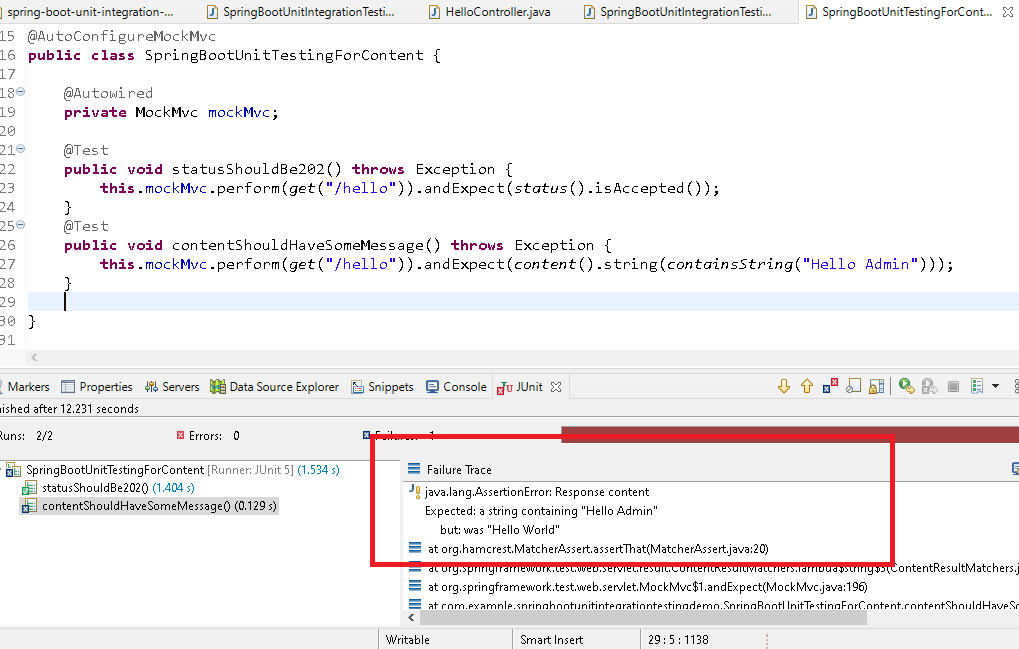


The above is just checking whether the instance of controller is created or not.

Let us try to use ResponseEntity in the @RestController and test status codes and content using MockMvc

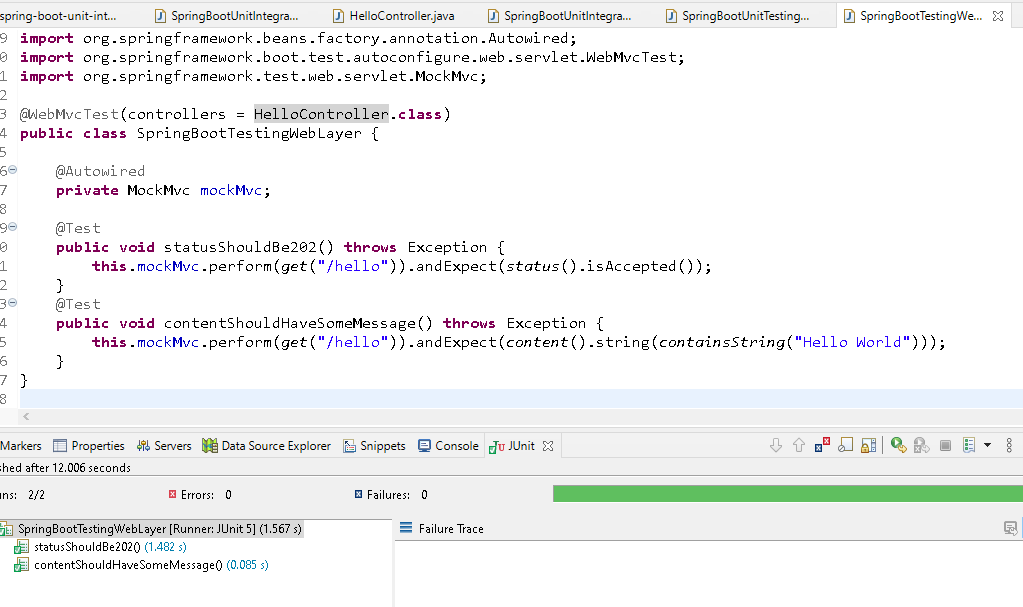


Writing test cases to test the status, content and so on

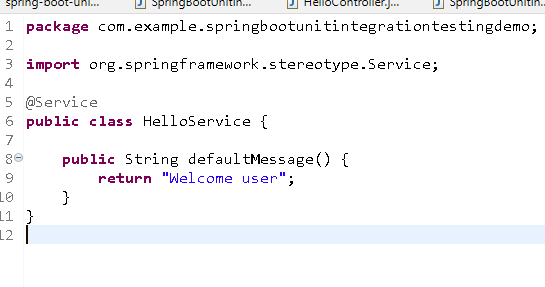


@AutoConfigureMockMvc: this configures everything of the application but you don’t need to use every object, you can specify whether you want to test web layer or service layer or dao layer using @WebMvcTest.

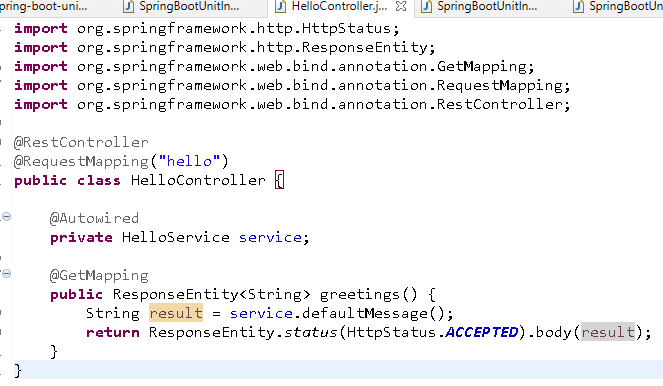
Ex: @WebMvcTest(HelloController.class)



Adding the service layer instance to the controller



Let us @Autowire Service layer in the controller & test the method



At the time testing you have to mock the service instance so that instead of getting the real value you will get a mocked value while testing the controller & controller should use that mocked value to test.

