Spring Microservices

Pre-requisites

* Core Java – OOPS & Design pattern (Factory pattern)
* Comparator & Lambda expressions
* Java 8 Streams – stream(), forEach(), filter(), collect(), sorted()
* Spring Framework – Dependency Injection & Annotations
* Spring Boot – Webservices & Configurations

Factory Design pattern

It is to abstract object creation at the client side

interface DBOperations {   
 void store();  
 void delete();  
}

class One implements DBOperations { }   
class Two implements DBOperations { }  
class Three implements DBOperations { }

Scenario1: Client creates object – leads to tightly coupled code

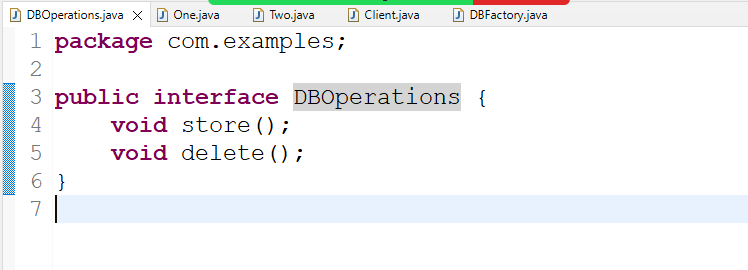
DBOperations db = new One(); // if new implementation must be used, then code must be modified here  
db.store();  
db.delete();

Scenario2: Client doesn’t create object – they use factory pattern to get the object – makes code loosely coupled

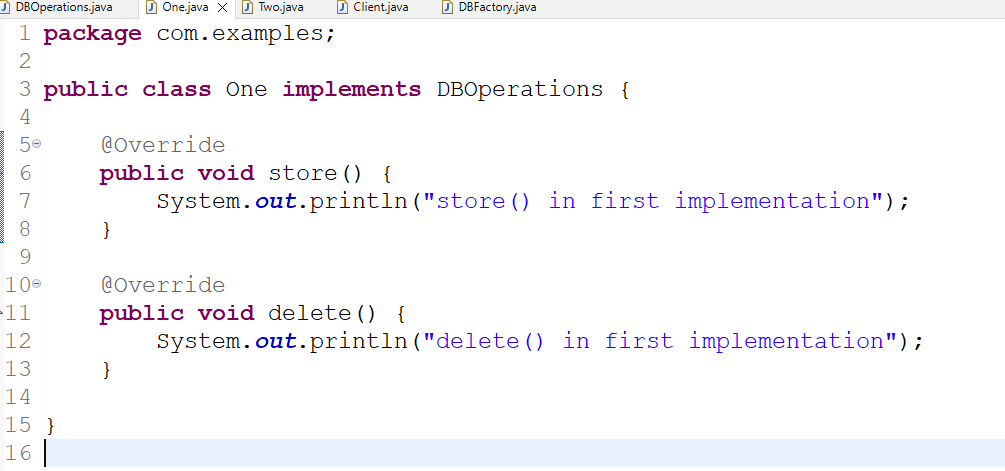
class DBFactory {   
 public static DBOperations getInstance() {   
 return new Two();  
 }  
}

DBOperations db = DBFactory.getInstance(); // client doesn’t know which implementation object is returned.  
db.store();  
db.delete();

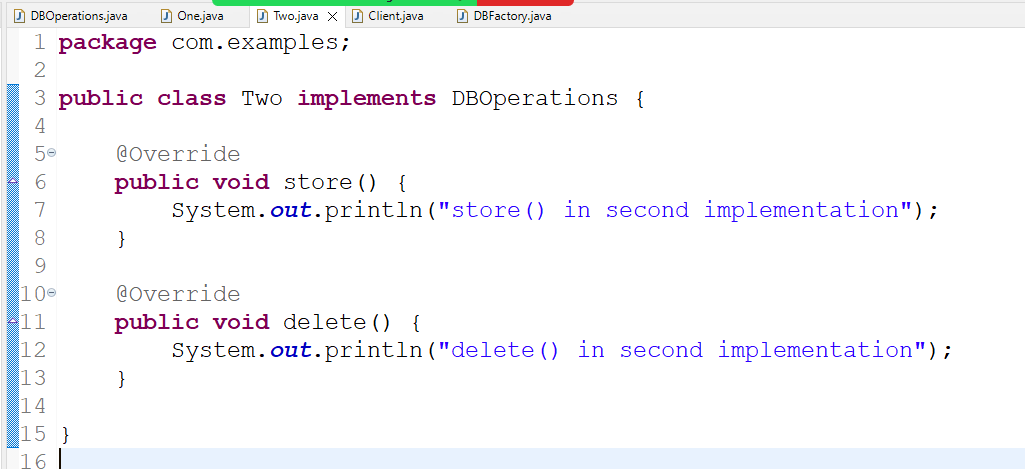
DBOperations.java



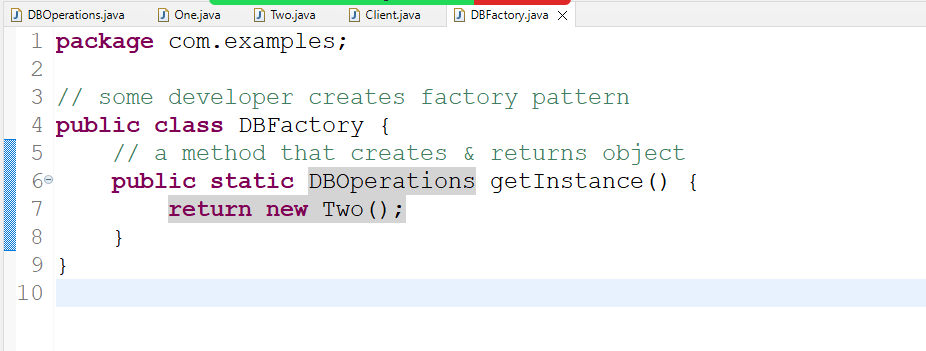
One.java



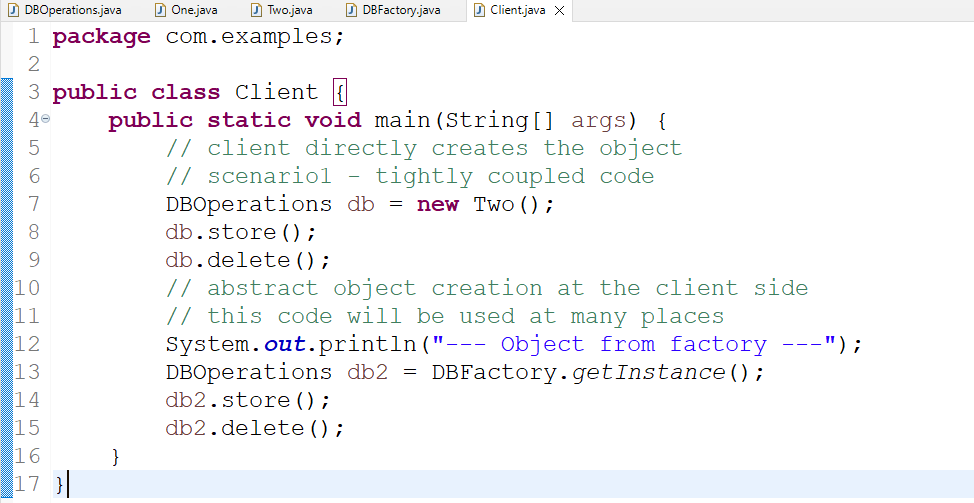
Two.java



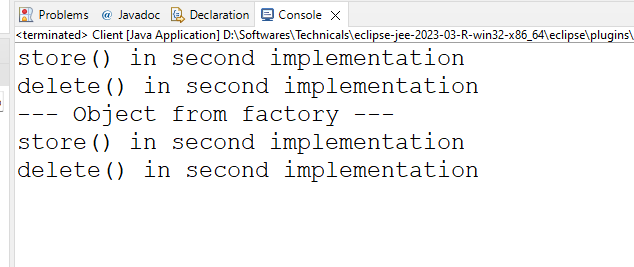
DBFactory.java



Client.java



Output:



Summary:

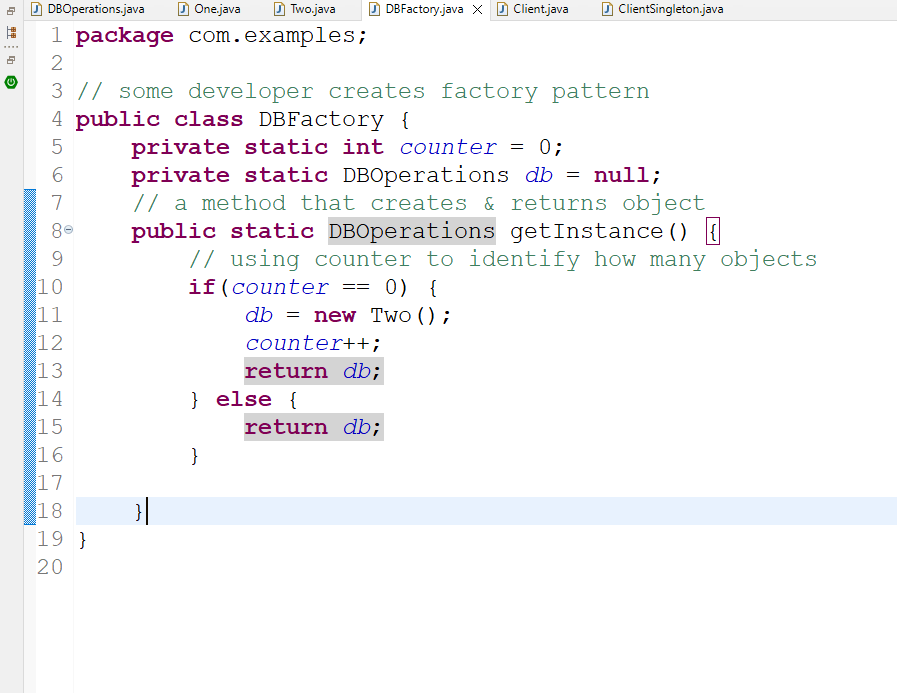
Developers who create object directly need to change their code when a new implementation needs to be used, developers who gets the object from the factory pattern need not to change the code because factory pattern takes care of giving the object.

Factory pattern vs Singleton pattern

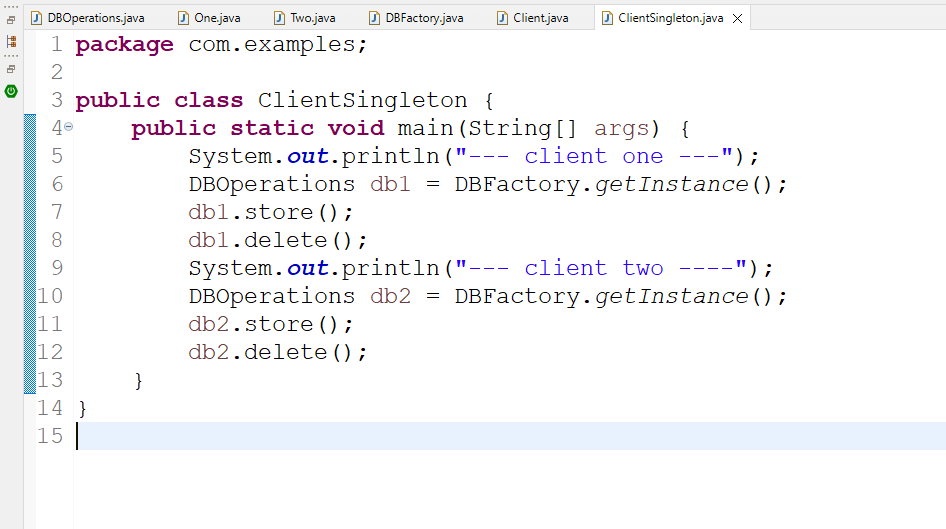
Factory pattern just creates the object, however singleton pattern is a factory pattern where on a class only one object will be created even if you call the factory method more than once

Previous factory pattern creates object more than once based on the how many time you call the getInstance() factory method, this leads to more number of object creation, to avoid this we can change the factory method to give only one object regardless of how many times you call the factory method

Modifying the factory to return a singleton object



ClientSingleton.java



Spring Framework:

Framework is like a semi implemented application which provides all the common features every application needs like

1. Type conversion: String to Number to String & Java types to SQL & vice versa
2. Design patterns:
3. Object Creation & Initialization
4. Exception Handling
5. Transaction Management
6. Connection Pooling

Spring Framework is an application framework which helps you to create various types of applications

1. Web / Webservices
2. Desktop
3. Cloud based application
4. Console based application

Spring Framework provides many modules to develop the above applications

1. Spring Core
2. Spring Web
3. Spring Boot
4. Spring Cloud
5. Spring Batch

Spring Core: It is the base modules that takes care of all the features every application needs like

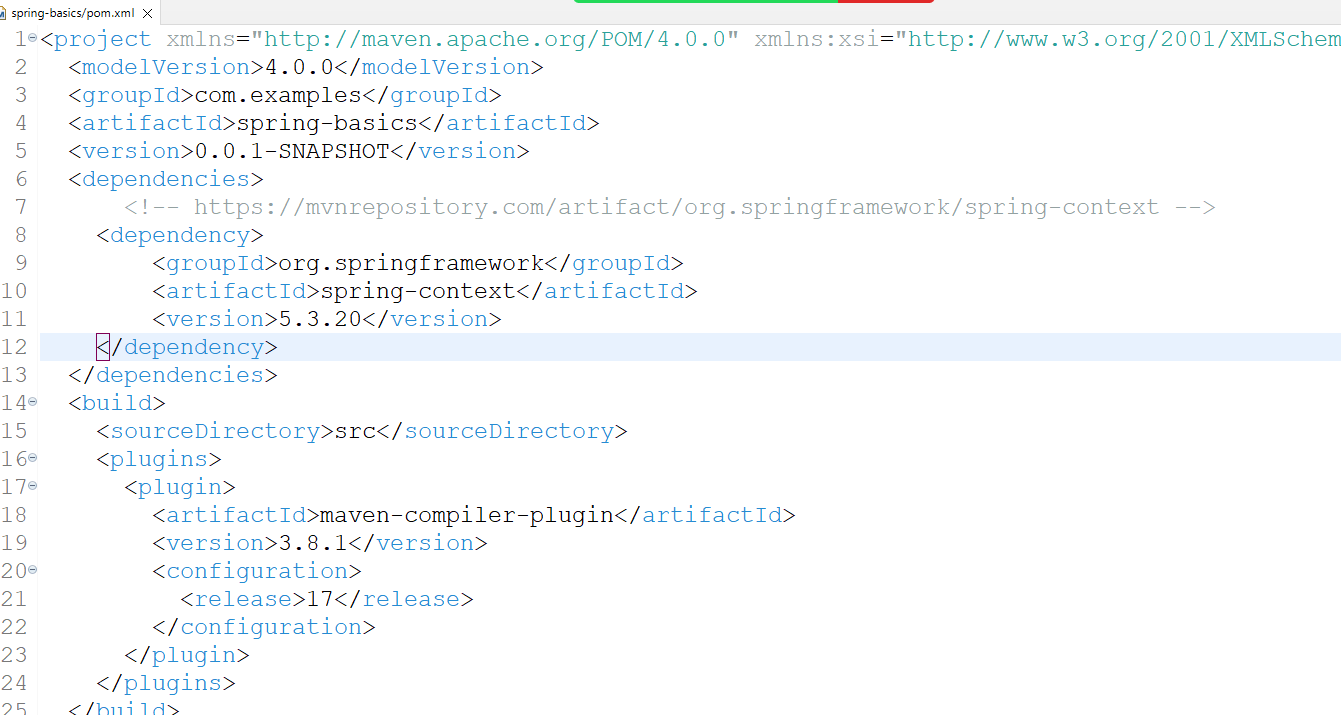
1. Design pattern
2. Object creation

Dependency Injection: It is a process of supplying an object to another object

Spring framework uses factory pattern internally to create objects so that developers don’t have to implement factory pattern

Spring Context: This is the library which you need to use to get the benefits of spring framework for Dependency Injection

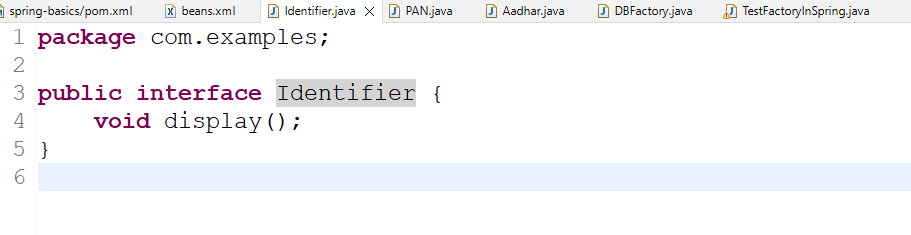
pom.xml



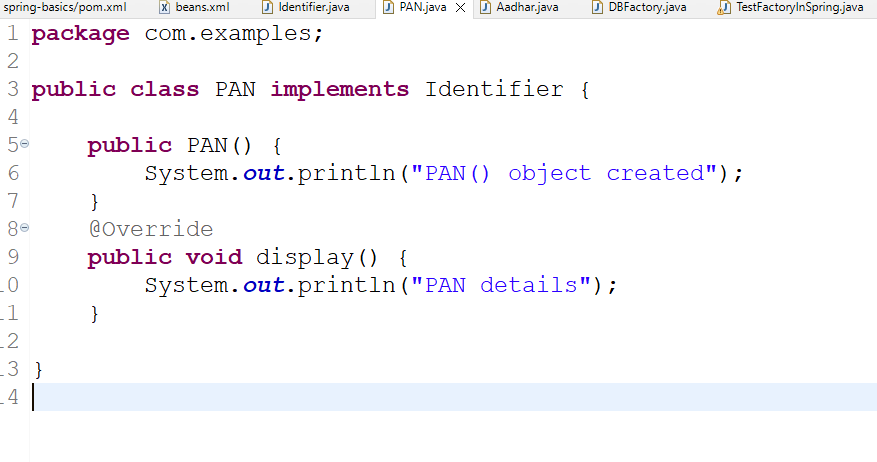
Project Structure

1. classes
2. XML – declare all the beans (java classes) whose object must be created by spring container

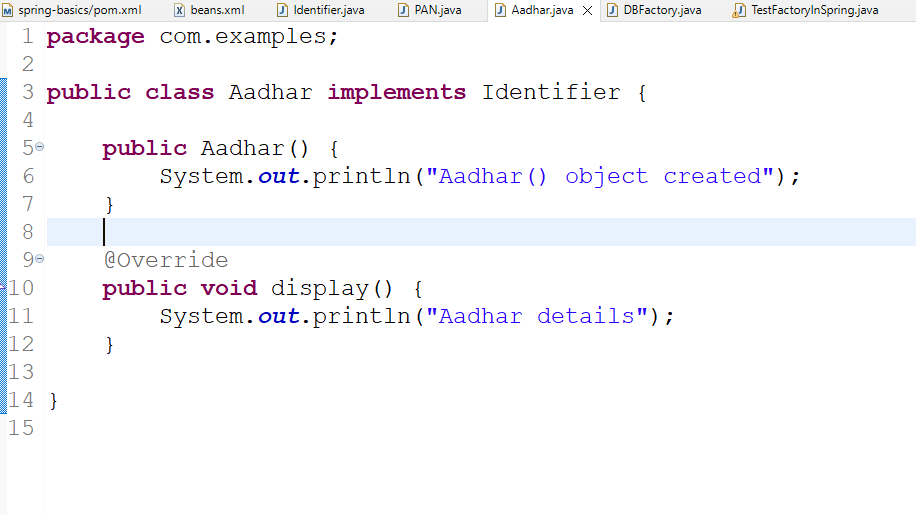
Identifier.java



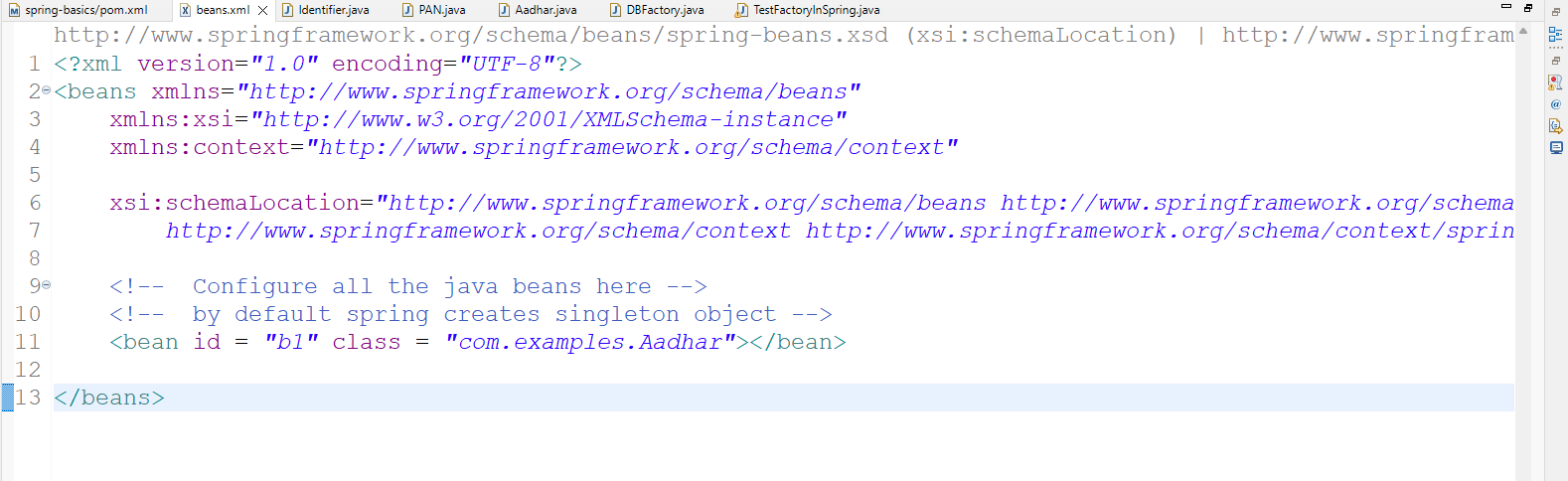
PAN.java



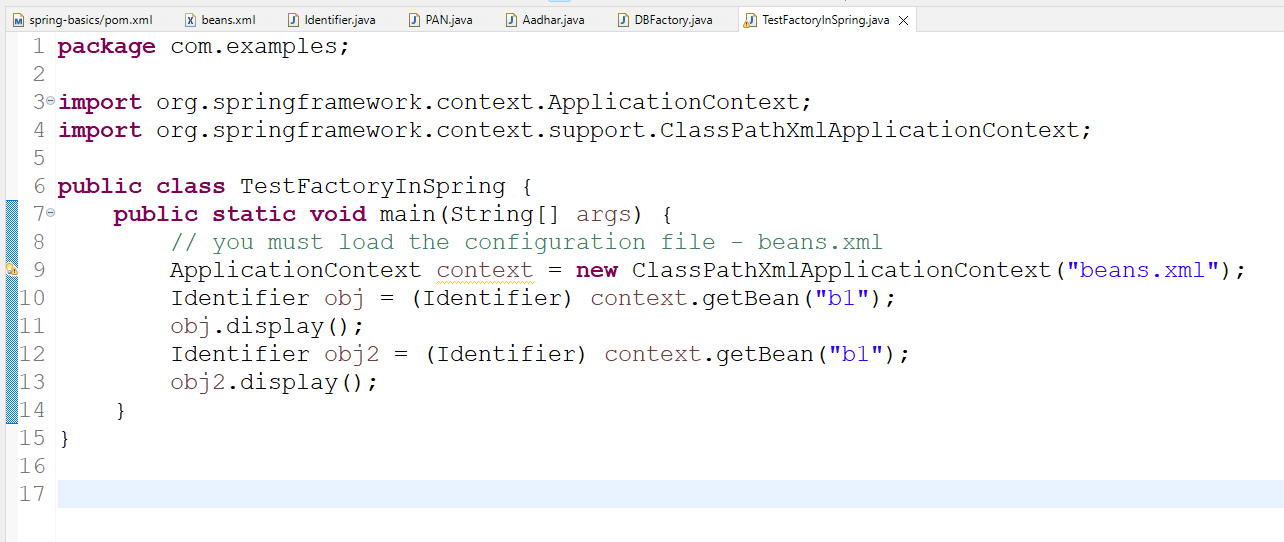
Aadhar.java



beans.xml

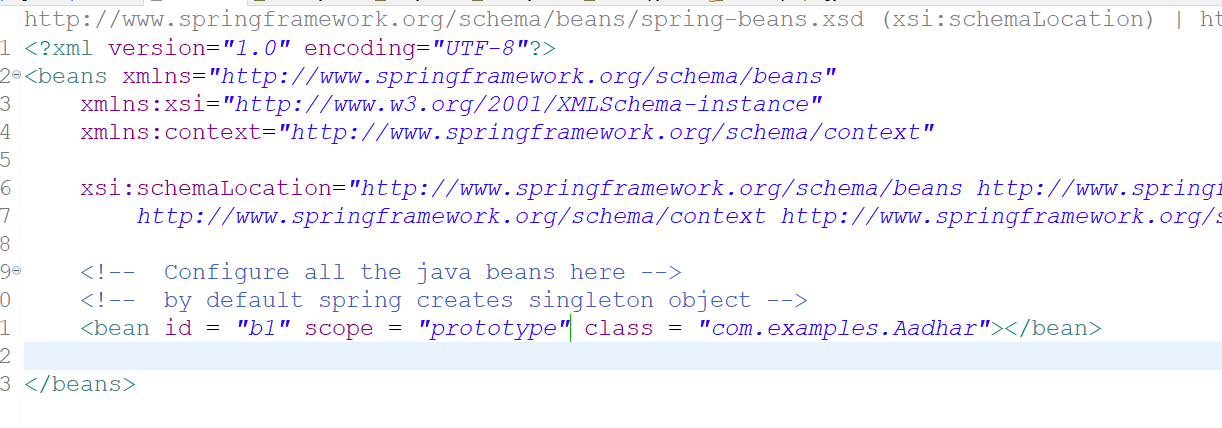


TestFactoryInSpring.java



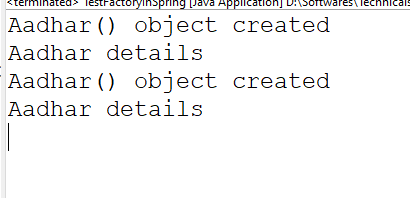
Note: At present spring container is creating singleton object, but you can make it to create multiple objects, for that you must use scope in the xml

beans.xml



scope is singleton by default, when it is prototype spring creates multiple objects for each getBean() invocation.

Output:



Annotation based configuration

It simplifies configuring the spring beans with simple annotations so that you can avoid declaring beans in the XML file

List of annotations spring provides to create the object

@Component  
@Service  
@Repository  
@RestController

@Service  
public class EmployeeService { } // id will be class name but first letter will be lowercase

This is equal to <bean id = “employeeService” class = “com.examples.EmployeeService”>

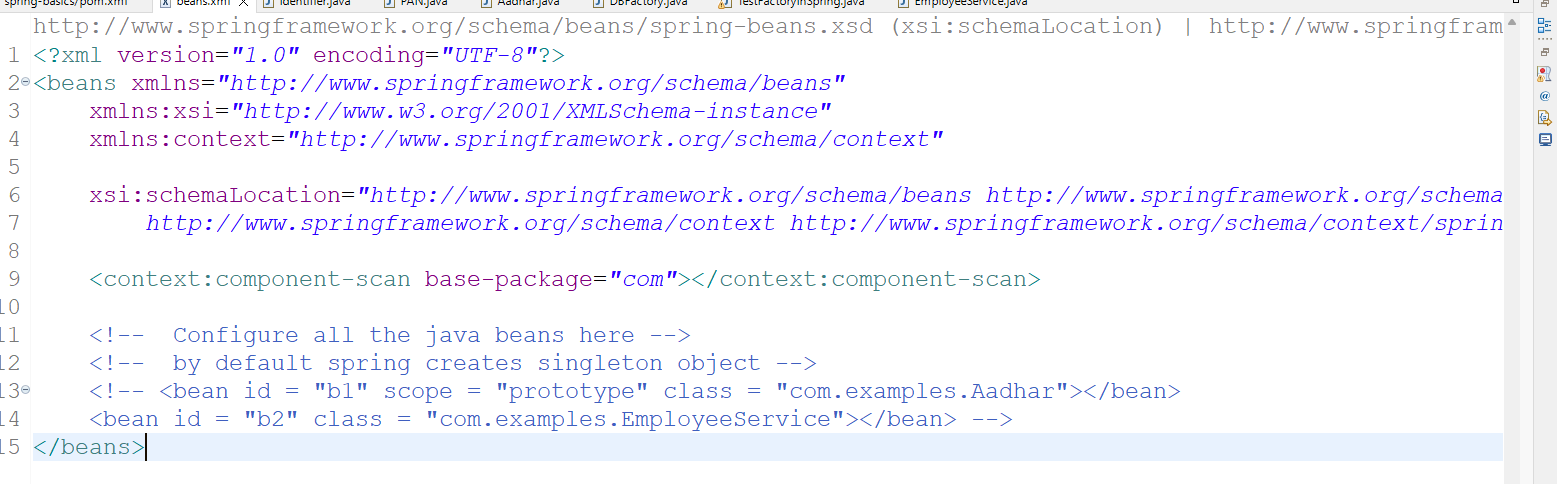
@Repository  
public class EmployeeDao { }   
This is equal to <bean id = “employeeDao” class = “com.examples.EmployeeDao”>

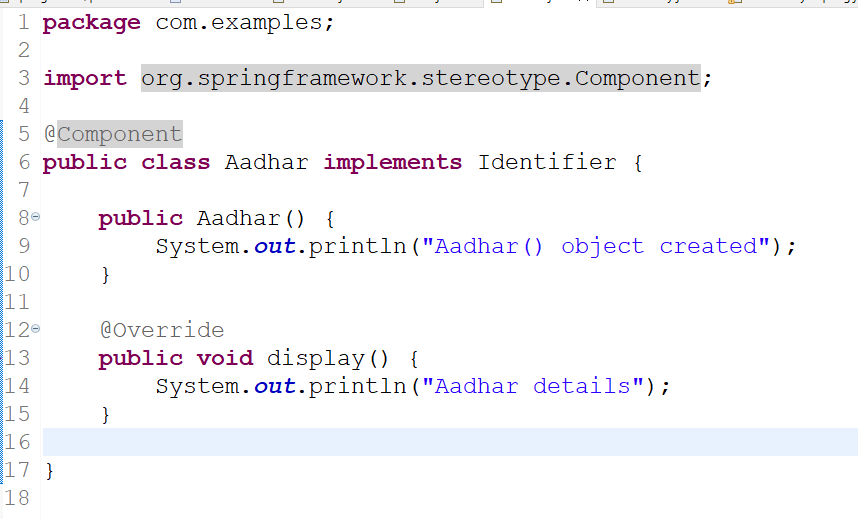
@Component is a base annotation where as @Service, @Repository, @RestController, @Configuration, @Controller are all derived from @Component

Note: If spring needs to search the classes having these annotation then in XML you need to use one tag to scan all the classes having these annotation

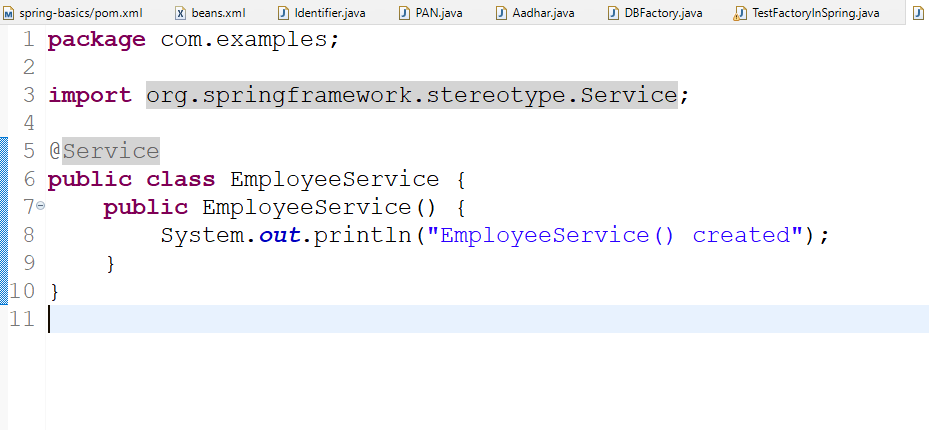
<context: component-scan base-package = “com.examples” />

beans.xml

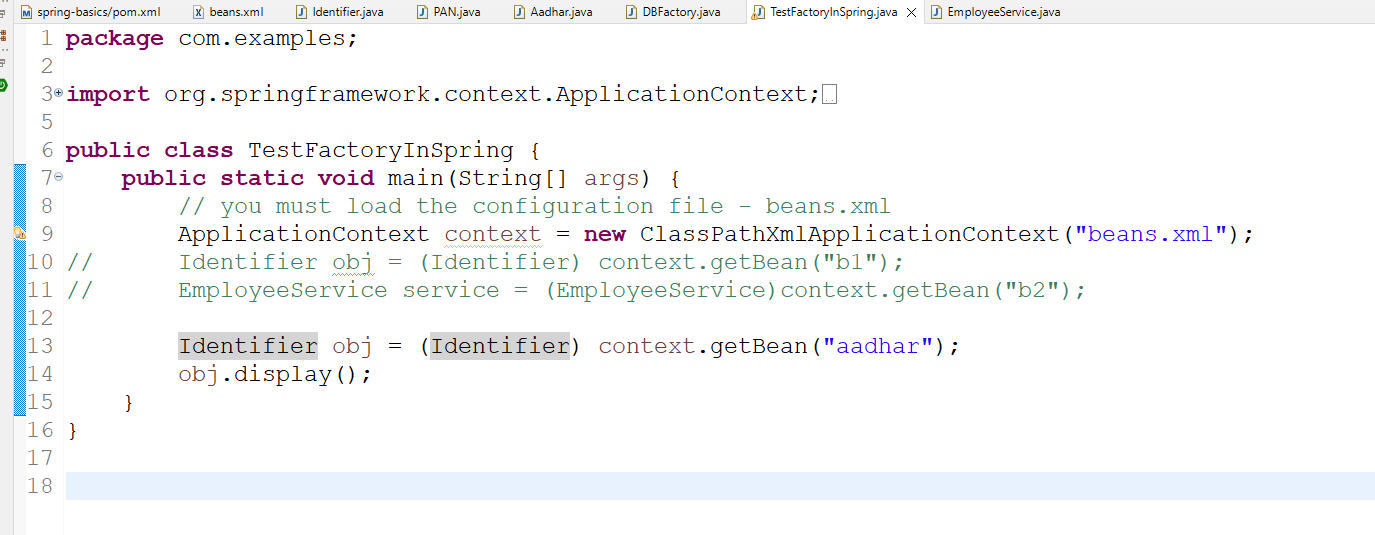




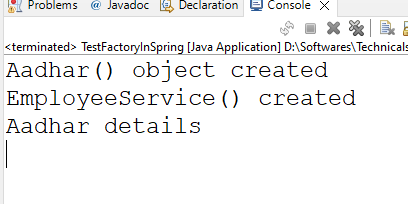
EmployeeService.java



TestFactoryInSpring.java



Output:



Spring is going to scan all the classes having @Component or their derived annotations to create the object.

Spring Boot

It simplifies developing the spring applications by auto-configuring your application based on the library you add

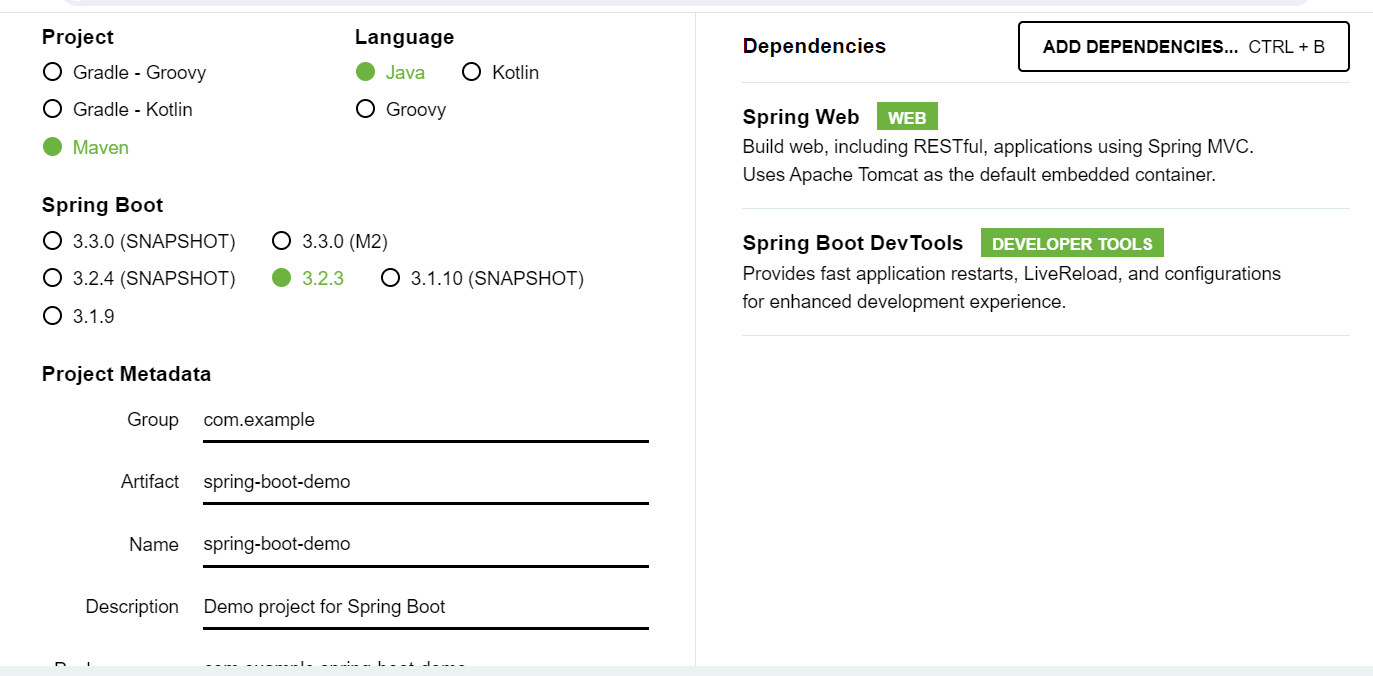
Spring boot takes care of configuring all the generic setup

* Server configurations
* Database configuration
* Component scanning
* Front controller configuration

Spring boot does this using the libraries which will have name as spring boot starter

ex: spring boot starter web, spring boot data jpa,

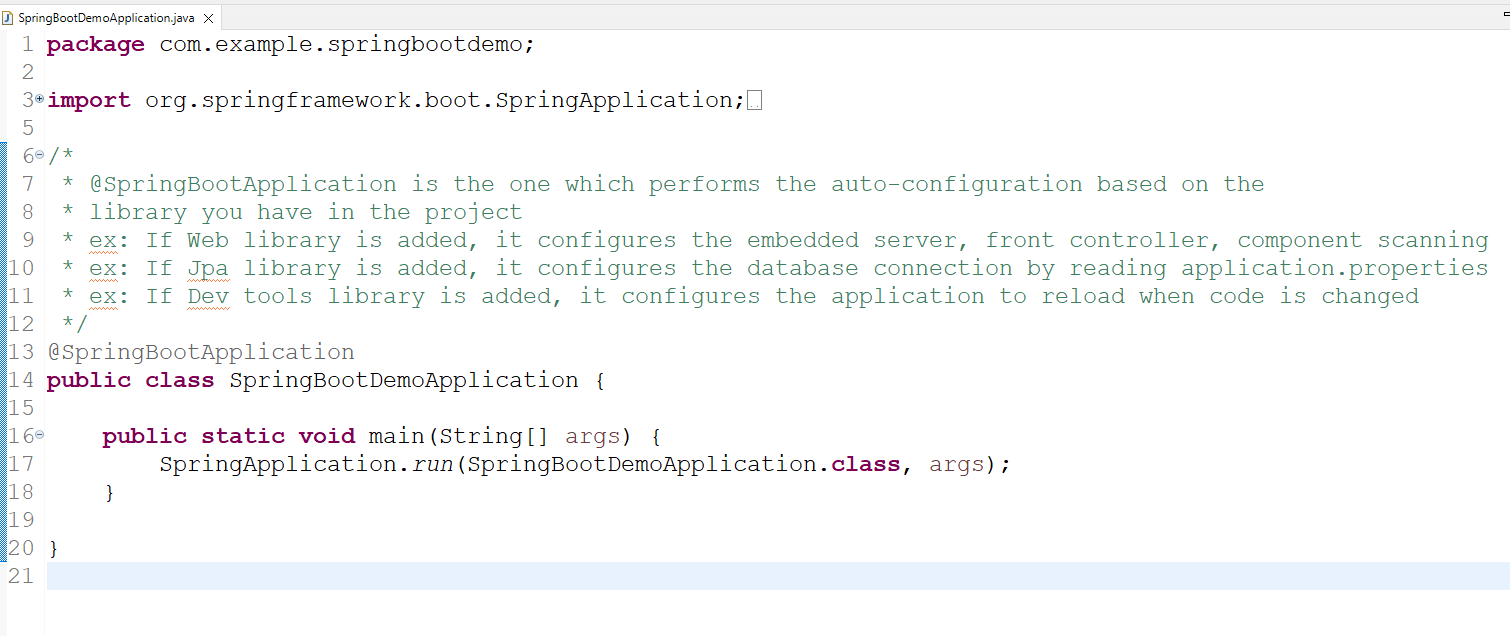
spring initializr: it is a official website to download empty spring boot project



When you download you will get a zip file, which you can directly open from eclipse, by following these steps

File -> Open Projects from file system -> In Import Source -> Archive -> zip file -> Select maven project & uncheck the folder not having maven

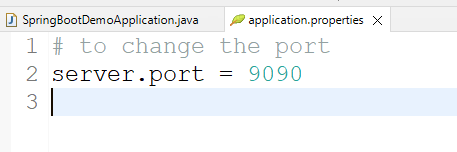
Every project of spring boot will have a main class with @SpringBootApplication that does the following job which is mentioned in the comments



You can get all the properties of spring boot in the below URL

<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.data>

application.properties



Webservices:

These are online services that helps heterogenous applications to share the data

ReSTful webservice exchanges the data in a common format (JSON)

ReST stands for Representational State Transfer

ReSTful uses two things to allow applications to exchange the data

1. URL: To locate the webservices
2. HTTP methods: To map the operations using GET, POST, PUT & DELETE

To create webservices in Spring we have some annotations

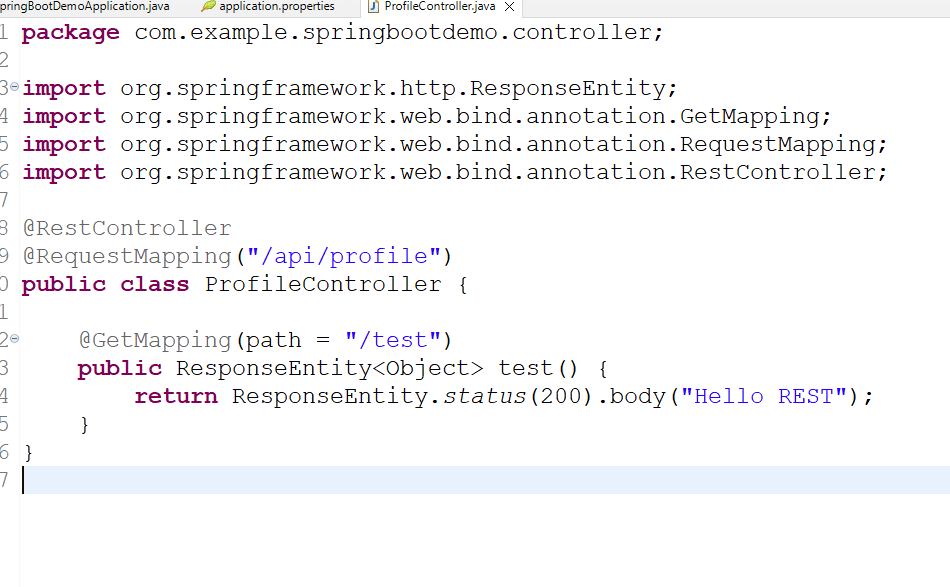
1. @RestController: This is to create a controller class which will have webservice code
2. @RequestMapping: This is to configure the URL for your controller
3. @GetMapping: This is to map HTTP GET
4. @PostMapping: This is to map HTTP POST
5. @PutMapping: This maps HTTP PUT
6. @DeleteMapping: This maps HTTP DELETE

@RestController  
@RequestMapping(“/api/profile”)

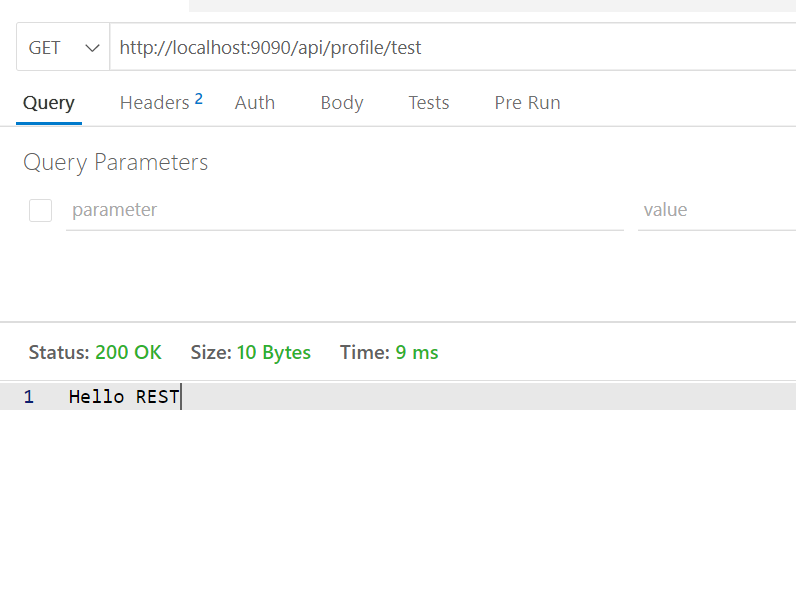
class ProfileController {   
 @GetMapping(path = “/test”)  
 public ResponseEntity test() {   
 return ResponseEntity.status(200).body(“some data”);  
 }  
}

Client uses: GET http://ip:port/api/profile/test

ProfileController.java



Output:



Different ways of passing the data to the webservice

1. URL: You can pass data via url path – simple data
2. Body: You can pass data via request body – JSON

api/profile/100

api/profile/200

api/profile/300

@GetMapping(“/api/profile/{x}”), @PostMapping(“/api/profile/{x}”)  
@PathVariable(“x”) int id;

@PathVariable(“x”) is going to extract the value associated with x and injects to the variable

The above code is sending the data via URL path

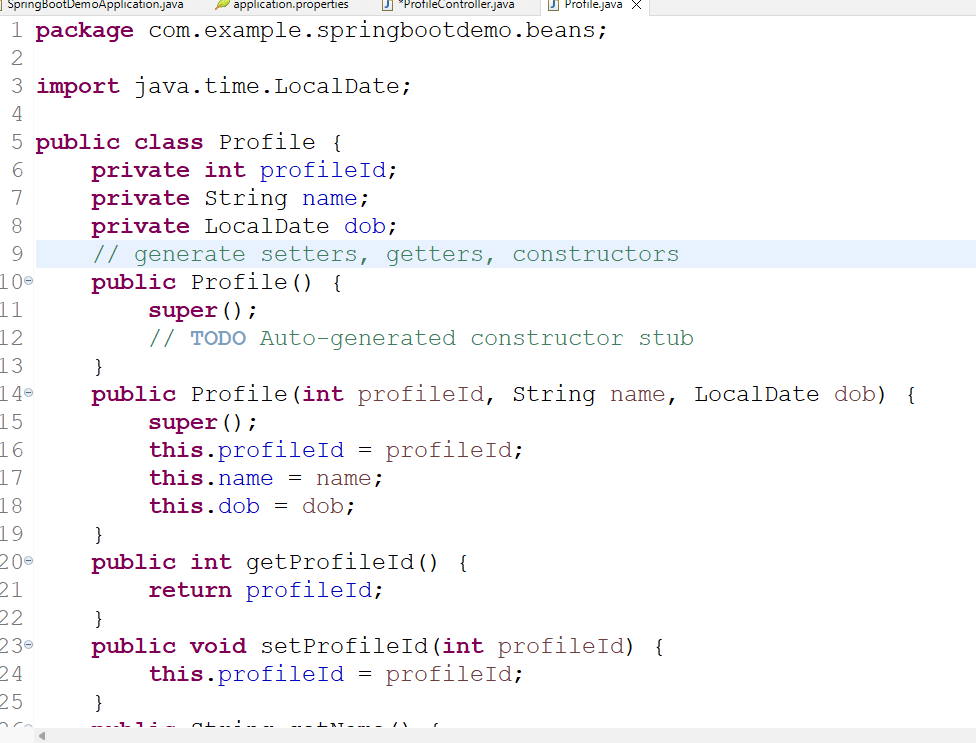
When you are entering complex data it is send in JSON format  
{ “name” : “Abc”, “phone”:99999, “dob”:”1998-10-25” }

@RequestBody: This annotation extracts the data from the request body, and maps to the java object

store(@RequestBody Profile pr) { }

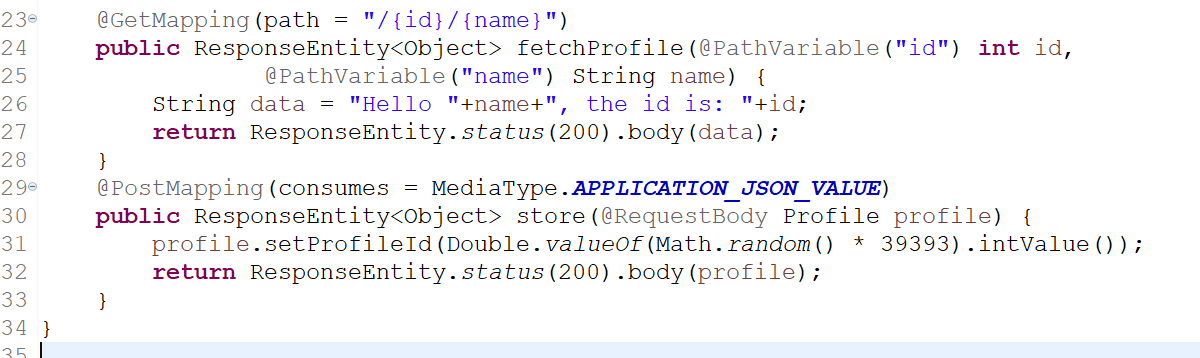
class Profile { name, phone, dob }

Profile.java

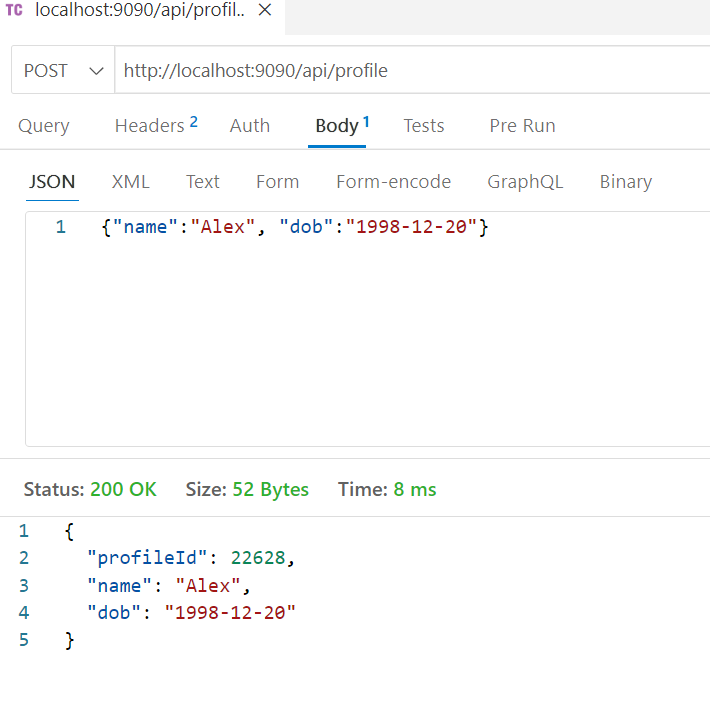


Create a webservice that accepts data in the url & the request body

ProfileController.java



Output:



Spring Data Jpa

It is a library which can automate DB connections & DB logics

For DB connections you must provide DB credentials in the application.properties

For DB logics it provides some inbuilt interfaces called Repository interfaces to that you must provide the entity class details

Entity Class; It is a class that provides table & primary information’s

Profile table : id, name, dob

@Entity  
@Table(name = “profile”) // optional if class name & tables names are same  
class Profile {   
 @Column(name = “id”) private int pid;  
 @Column(name = “name”) private String pname;  
 @Column(name = “dob”) private LocalDate pdob;  
}

You can name your properties same as column names to directly map the object to the table without using @Column

@Entity  
@Table(name = “profile”)  
class Profile {   
 @Id  
 private int id; // maps to the column having the name id  
 private String name; // maps to the column having the name ‘name’  
 private LocalDate dob; // maps to dob

}

@Id: We need to use this on a property that maps to the primary key column, because in spring data jpa DB logics are automated, many queries are generated based on the primary key like find by primary key, delete by primary key, update by primary key

Repository interfaces: These are the inbuilt interfaces that will make Spring Data JPA to automate the DB logics like CRUD operations based on the entity, these interfaces provide some inbuilt methods which can perform CRUD operations on the entity which reflects to the table

1. CrudRepository<T, ID>: gives methods like save, findById, deleteById, findAll
2. JpaRepository<T, ID>: extends CrudRepository and provides methods to sort, saving multiple objects and etc.

T: Is an entity class type

ID: Is a primary key class type

CrudRepository<T, ID> has following methods

1. save(T): Here it saves the entity
2. findById(ID): Here it accepts the id and returns the entity matching to the id
3. deleteById(ID): Here it accepts the id & deletes the entity matching to the id
4. findAll(): It returns all the entities in List<T>

Our job is to inherit either CrudRepository or JpaRepository, but spring boot automates the implementation of the interface you create.

interface ProfileRepository extends CrudRepository<Profile, Integer> {   
   
}

Now ProfileRepository gets all the methods of CrudRepository in the form as

save(T) = save(Profile)

findById(ID) = findById(Integer)

All the methods of ProfileRepository will auto-implemented so that when you call the methods like save, deleteById, findById the reflect to the profile entity i.e., profile table.

Service layer must only use the object of this auto-implemented class

@Autowired  
private ProfileRepository profileDao;

Things to create

1. Profile entity: Using @Entity, @Id, @GeneratedValue
2. ProfileRepository to extend CrudRepository<Profile, Integer> or JpaRepository<Profile, Integer>
3. Autowire the ProfileRepository in the service layer
4. @Service: Service layer must have methods to call ProfileRepository methods like save, deleteById, findById, findAll
5. Controller layer must autowire service layer
6. application.properties: Must have database credentials