Lesson 03 Demo 02

Create Microservice with MongoDB

Objective: To create a microservice using Spring Boot and MongoDB to perform CRUD operations on a user collection in a MongoDB database

Tool required: Eclipse IDE, MongoDB Atlas, and Postman

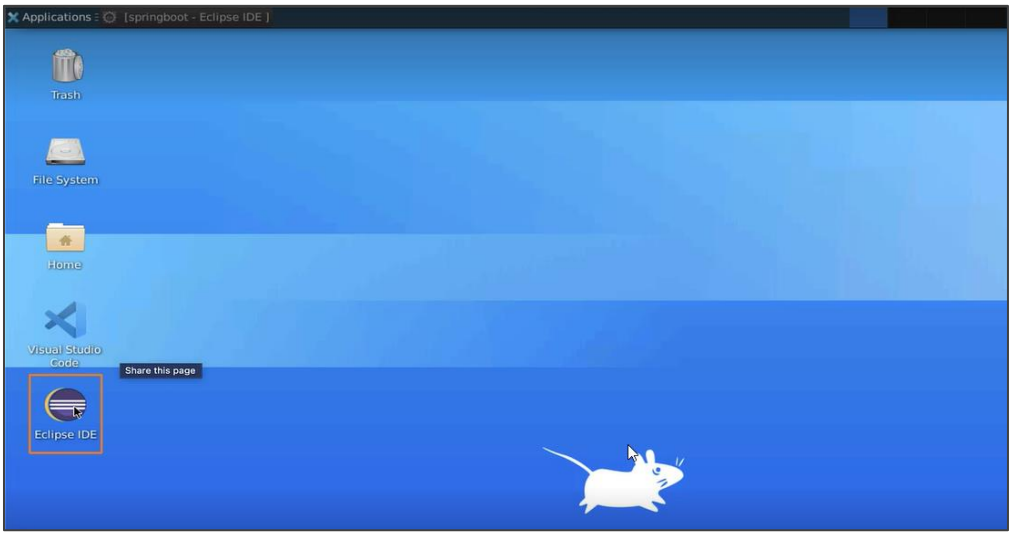
Prerequisites: None

Steps to be followed:

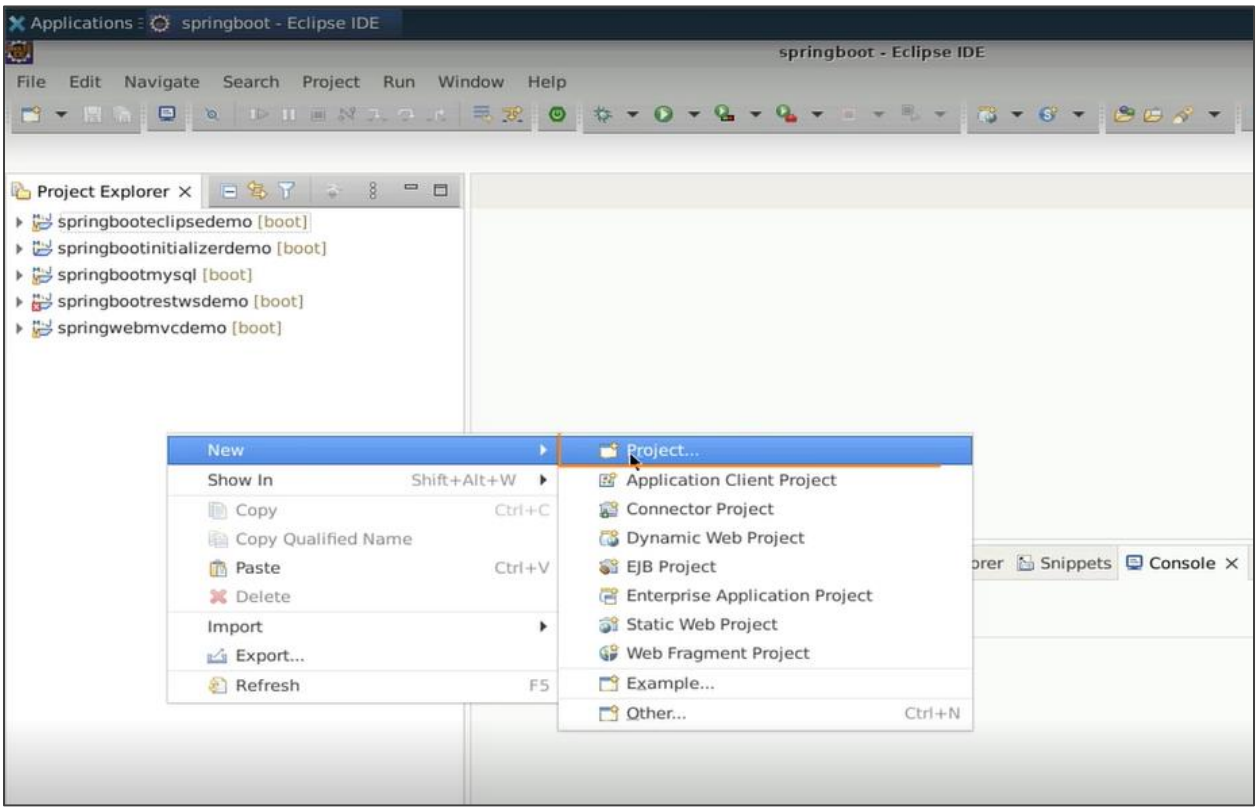
1. Creating a new Spring Starter project
2. Creating the Address Model class
3. Creating the User Model class
4. Configuring the MongoDB database
5. Setting up the database configuration
6. Creating the UserRepository interface
7. Creating the UserController class
8. Creating the Response class
9. Configuring the CRUD methods
10. Running and testing the application

Step 1: Creating a new Spring Starter project

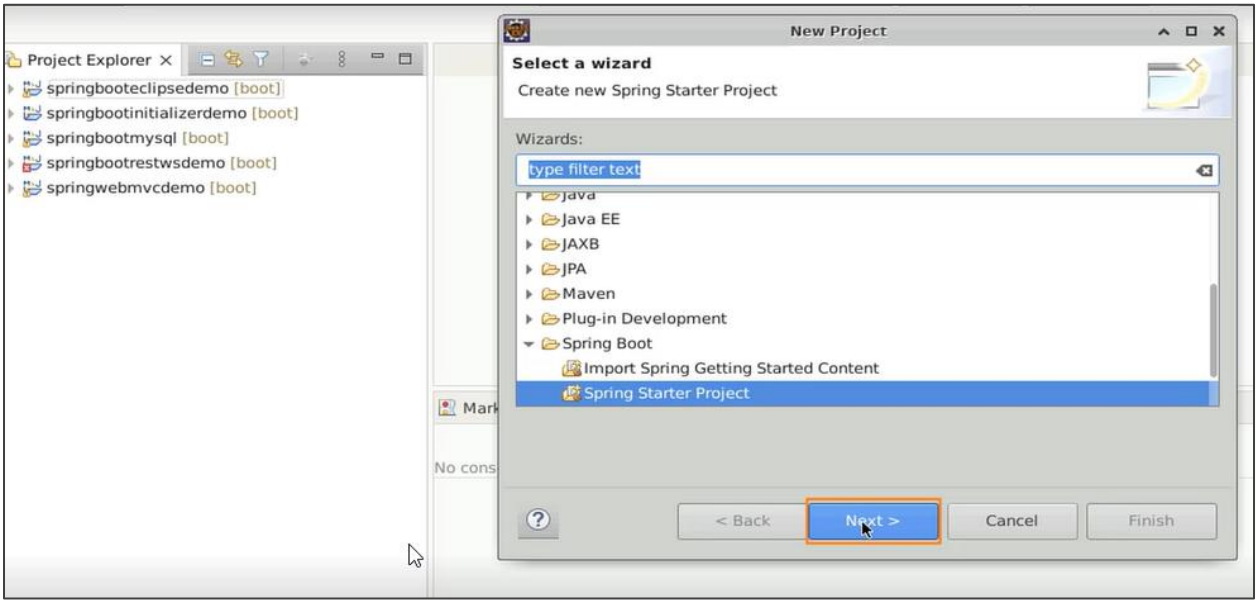
1.1 Open **Eclipse IDE**



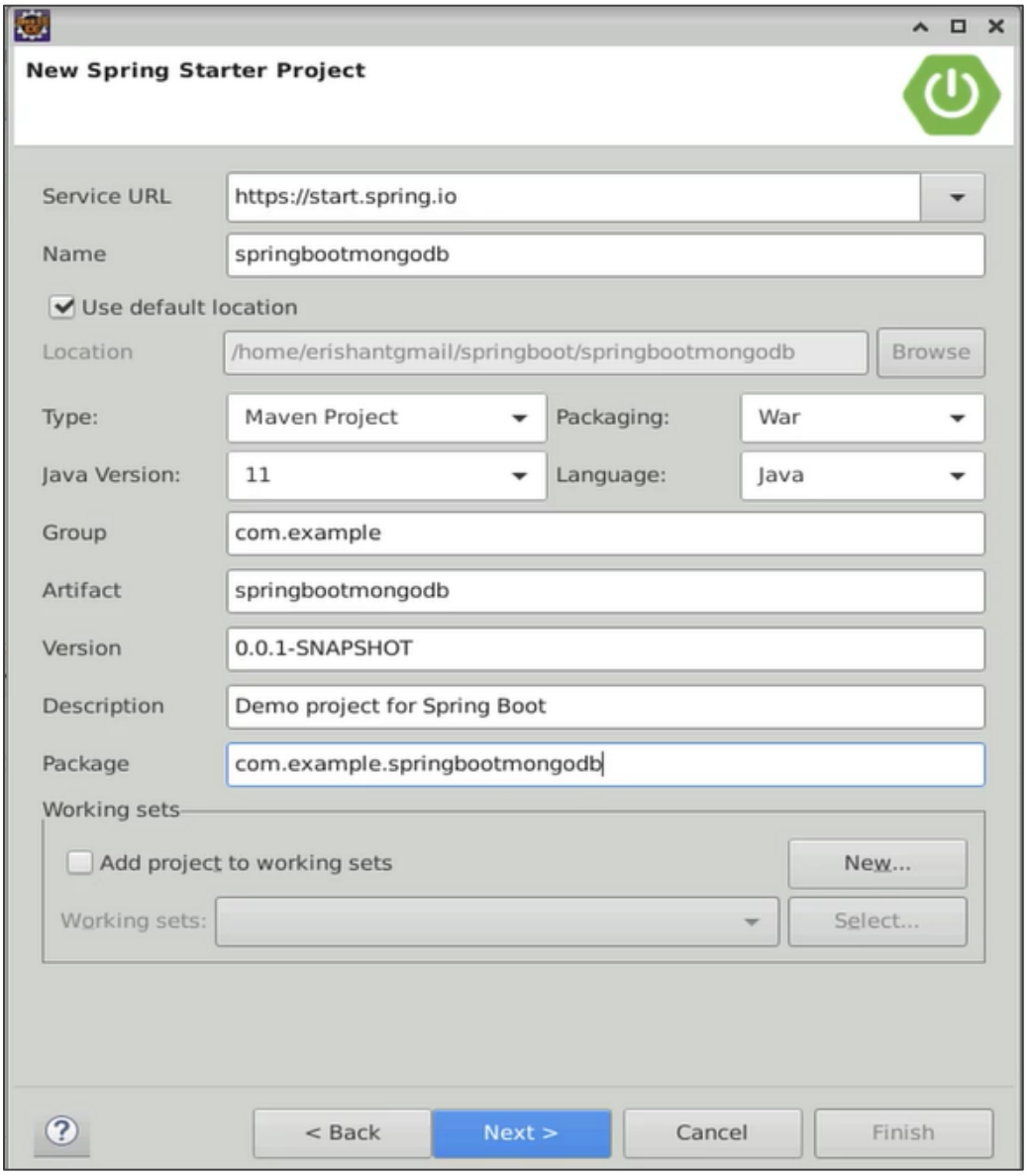
1.2 In the Project Explorer, right-click and select **New** > **Project**



1.3 Select **Spring Starter Project** and click **Next**

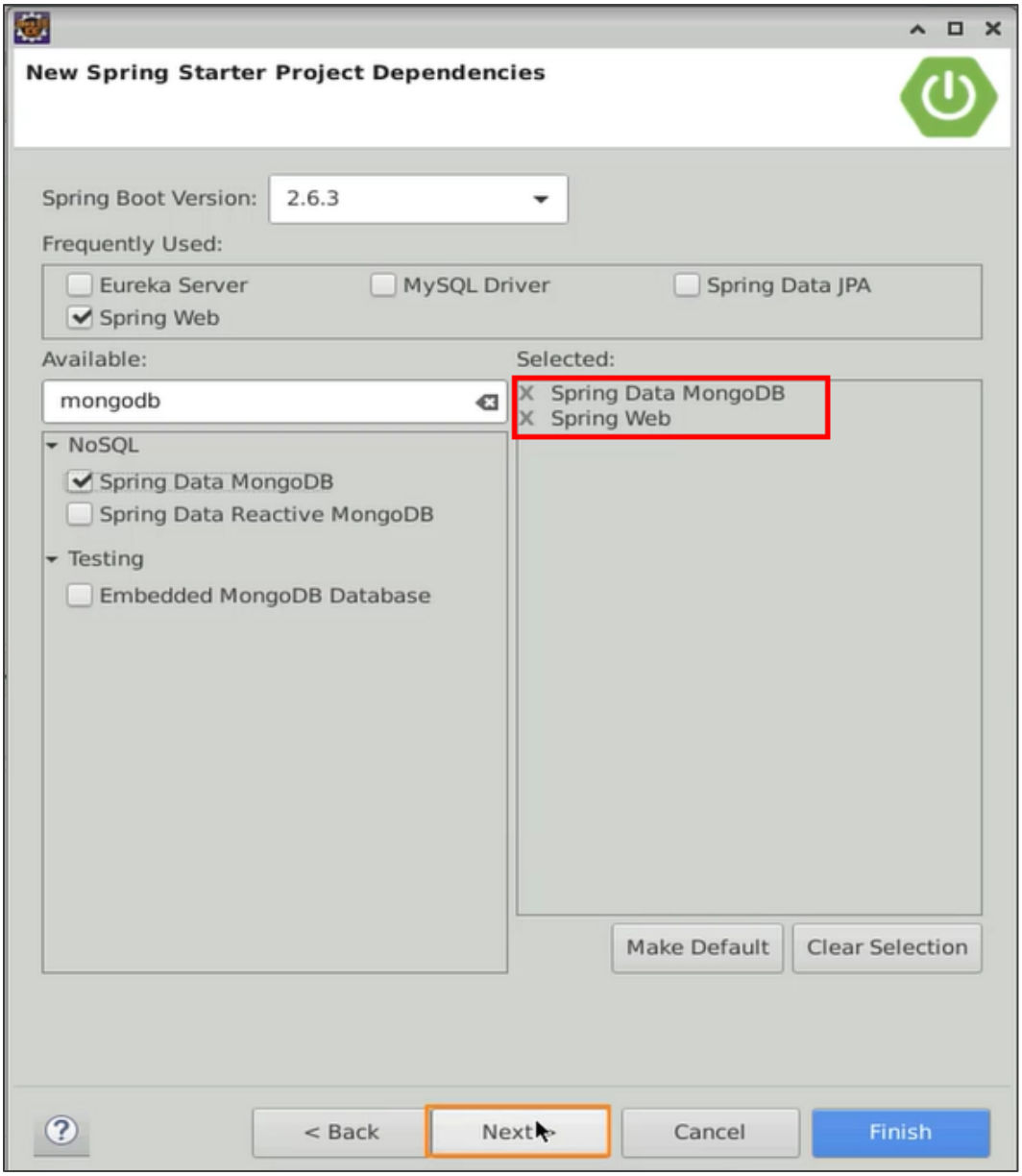


1.4 Provide a project name, such as **springbootmongodb**, configure the project with Maven as the build tool, and choose the packaging as **War**

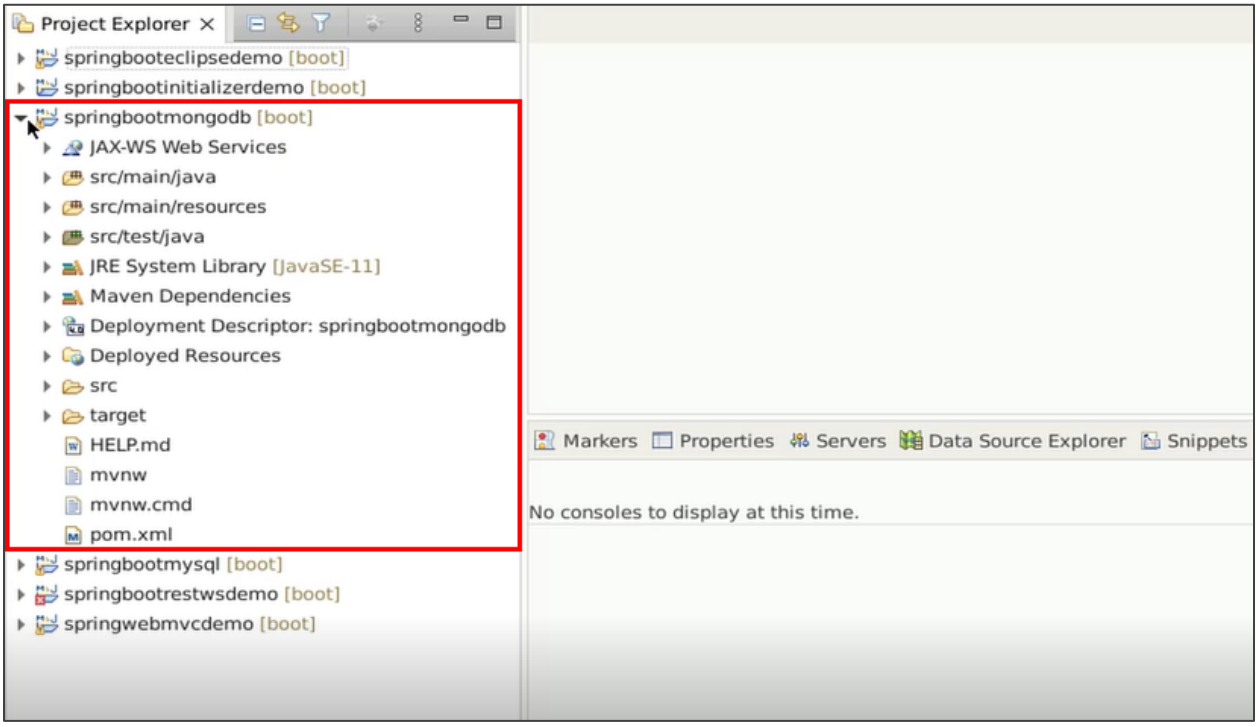


1.5 Click **Next** and add the following dependencies:

- Spring Data MongoDB
- Spring Web



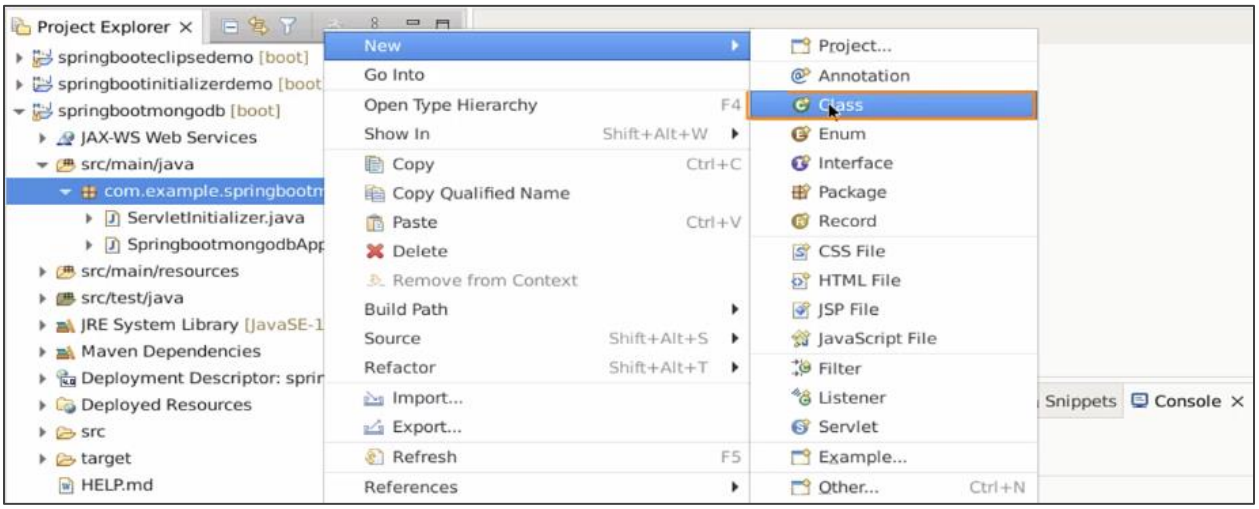
1.6 Click **Finish** to create the project with the specified dependencies



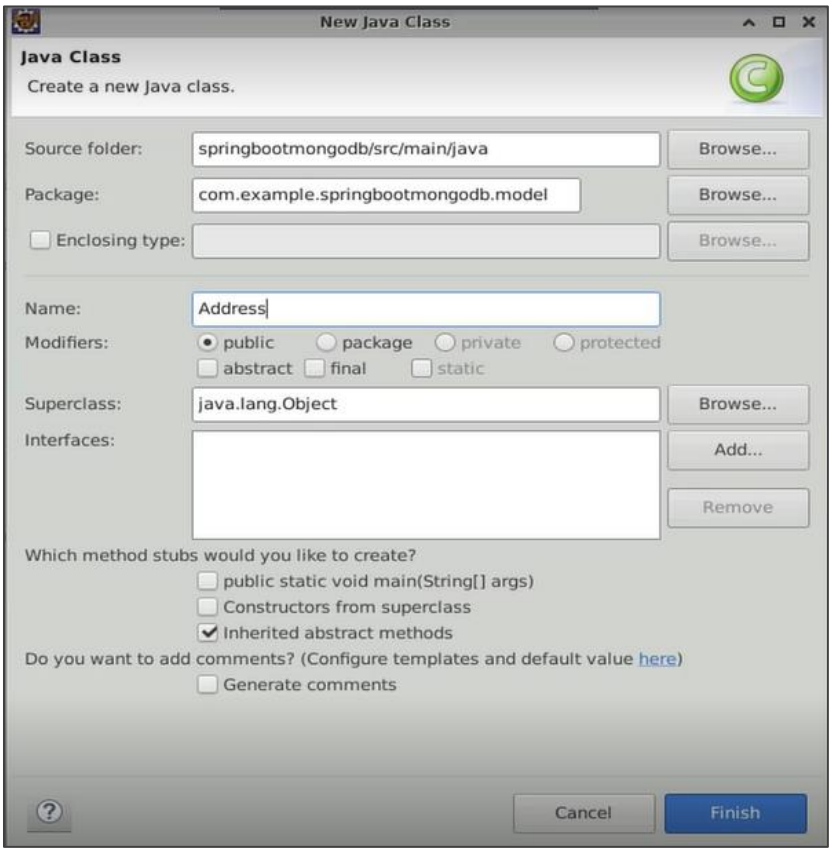
The eclipse will generate the project structure and download the necessary dependencies.

Step 2: Creating the Address Model class

2.1 Right-click on the source package and select **New** > **Class**



2.2 Name the class **Address** and add **.model** to the package name and click **Finish**



2.3 Add the required attributes for the Address class, such as **adrsLine**, **city**, **state**, and **zipCode**

```
User.java    *Address.java ×
1  package com.example.springbootmongodb.model;
2
3  public class Address {
4
5      String adrsLine;
6      String city;
7      String state;
8      Integer zipCode;
9
10 }
11
```

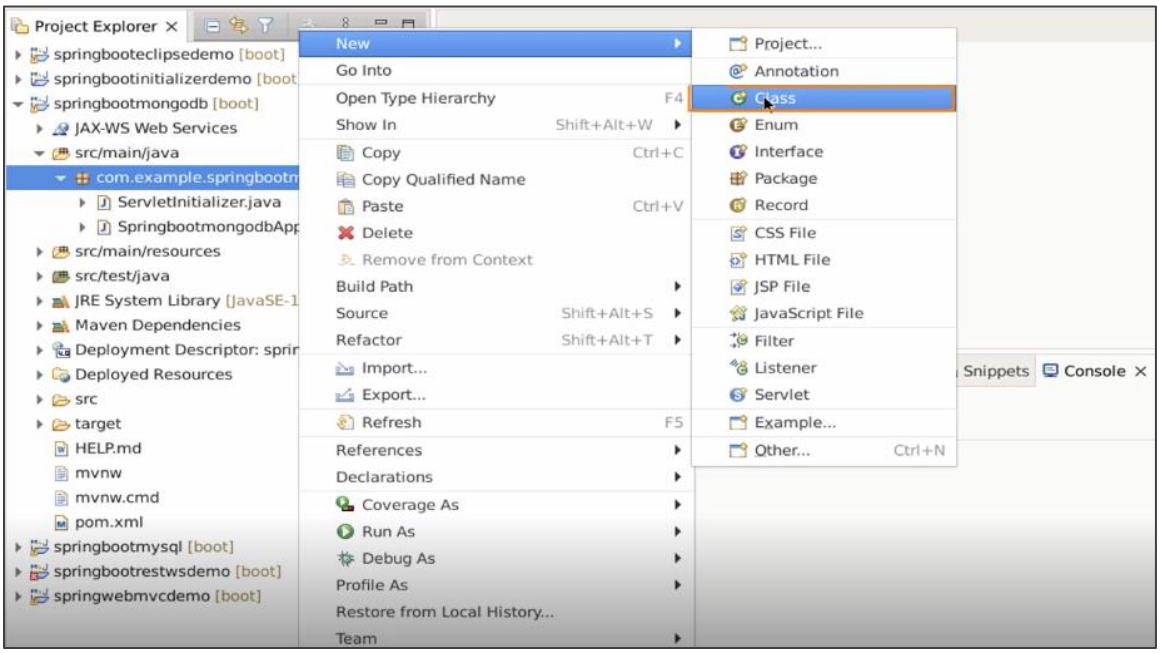
2.4 Generate the default constructor, parameterized constructor, getters, setters, and a **toString()** method for the Address class

```
User.java    *Address.java ×
1  package com.example.springbootmongodb.model;
2
3  public class Address {
4
5      String adrsLine;
6      String city;
7      String state;
8      Integer zipCode;
9
10     public Address() {
11         // TODO Auto-generated constructor stub
12     }
13
14     public Address(String adrsLine, String city, String state, Integer zipCode) {
15         this.adrsLine = adrsLine;
16         this.city = city;
17         this.state = state;
18         this.zipCode = zipCode;
19     }
20
21
```

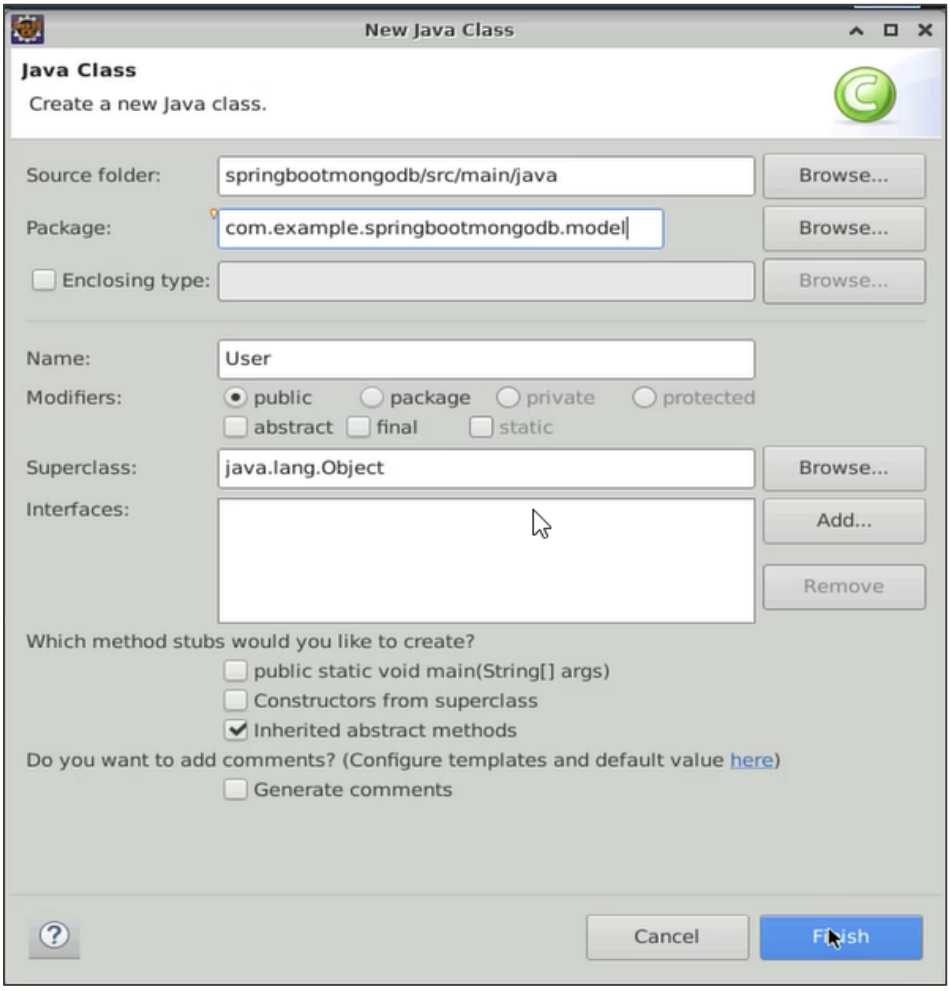
```
User.java    *Address.java ×
25  public void setAdrsLine(String adrsLine) {
26      this.adrsLine = adrsLine;
27  }
28
29  public String getCity() {
30      return city;
31  }
32
33  public void setCity(String city) {
34      this.city = city;
35  }
36
37  public String getState() {
38      return state;
39  }
40
41  public void setState(String state) {
42      this.state = state;
43  }
44
45  public Integer getZipCode() {
46      return zipCode;
47  }
48
49  public void setZipCode(Integer zipCode) {
50      this.zipCode = zipCode;
51  }
52
53  @Override
54  public String toString() {
55      return "Address [adrsLine=" + adrsLine + ", city=" + city + ", state=" + state + ", zipCode=" + zipCode + "]";
56  }
57
```

Step 3: Creating the User Model class

3.1 Right-click on the source package and select **New** > **Class**



3.2 Name the class **User** and add **.model** to the package name and click **Finish**



3.3 Add fields such as **id** (annotated with **@Id**), **name**, **phone**, **email**, and an **address** object

User.java

```java
package com.example.springbootmongodb.model;

import org.springframework.data.annotation.Id;

public class User {

    @Id
    String id;

    String name;
    String phone;
    String email;

}
```

User.java | Address.java

```java
package com.example.springbootmongodb.model;

import org.springframework.data.annotation.Id;

public class User {

    @Id
    String id;

    String name;
    String phone;
    String email;

    Address address; // HAS-A Relationship

}
```

3.4 Generate the default constructor, parameterized constructor, getters, setters, and a **toString()** method for the User class

*User.java × Address.java

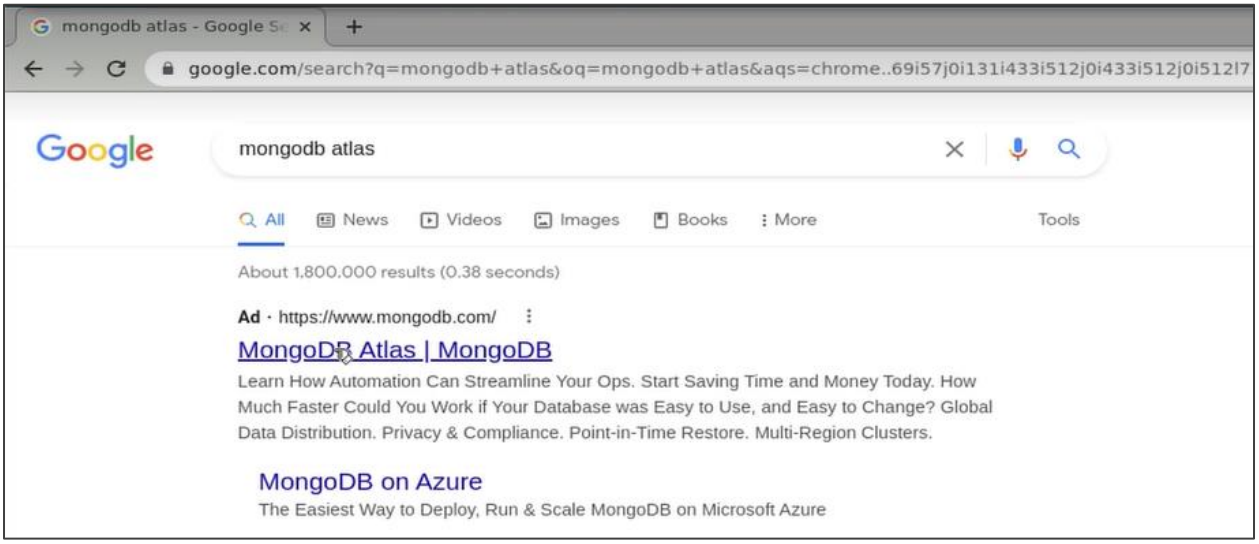
```java
package com.example.springbootmongodb.model;

import org.springframework.data.annotation.Id;

public class User {

    @Id
    String id;

    String name;
    String phone;
    String email;

    Address address; // HAS-A Relationship | 1 to 1

    public User() {
        // TODO Auto-generated constructor stub
    }

    public User(String id, String name, String phone, String email, Address address) {
        this.id = id;
        this.name = name;
        this.phone = phone;
        this.email = email;
        this.address = address;
    }
```

*User.java × Address.java

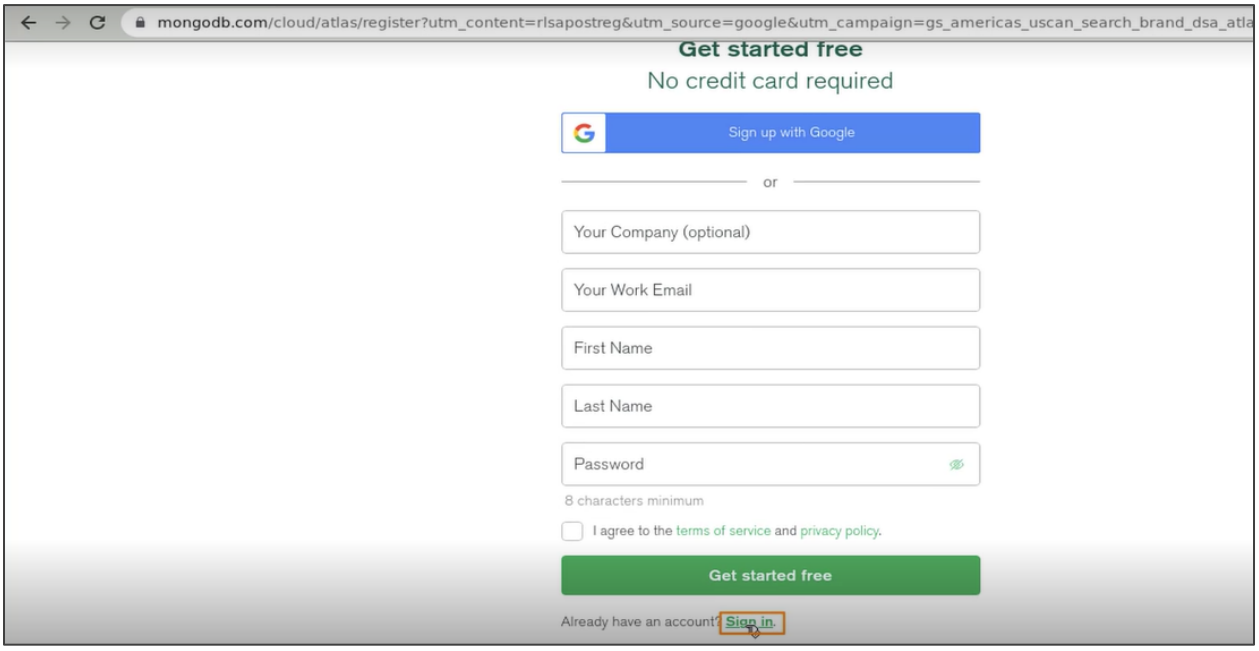
```java
    public void setName(String name) {
        this.name = name;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "User [id=" + id + ", name=" + name + ", phone=" + phone + ", email=" + email + ", address=" + address
                + "]";
    }
```

Step 4: Configuring the MongoDB database

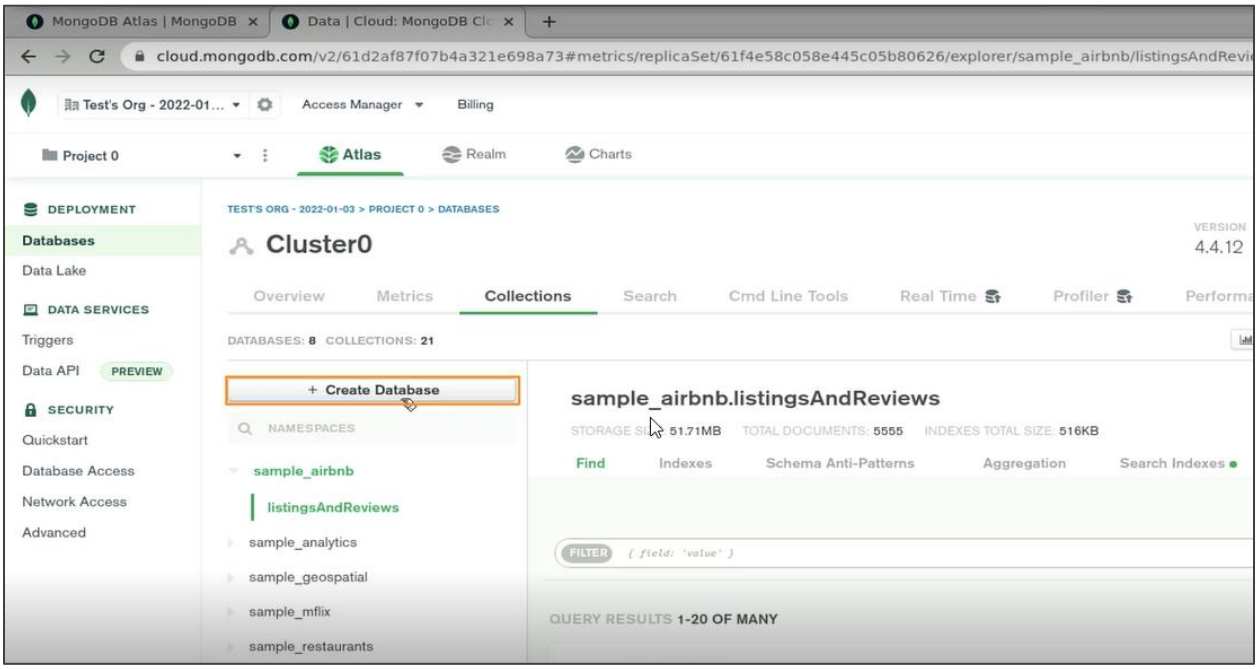
4.1 Open a web browser and navigate to the URL **https://www.mongodb.com**



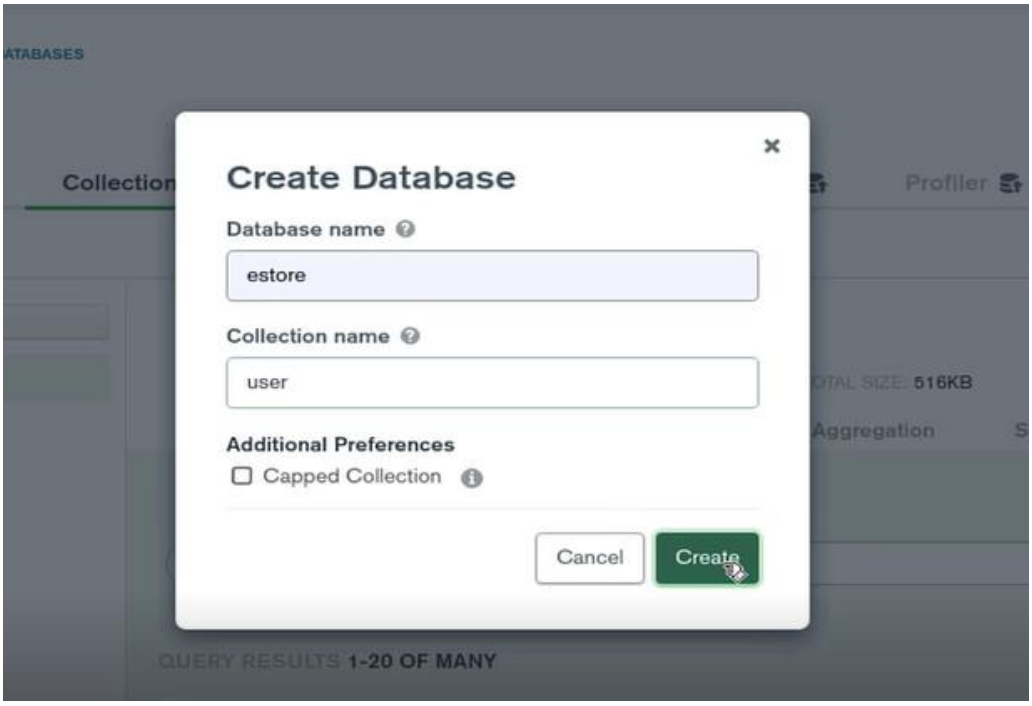
4.2 Register a new account and sign in

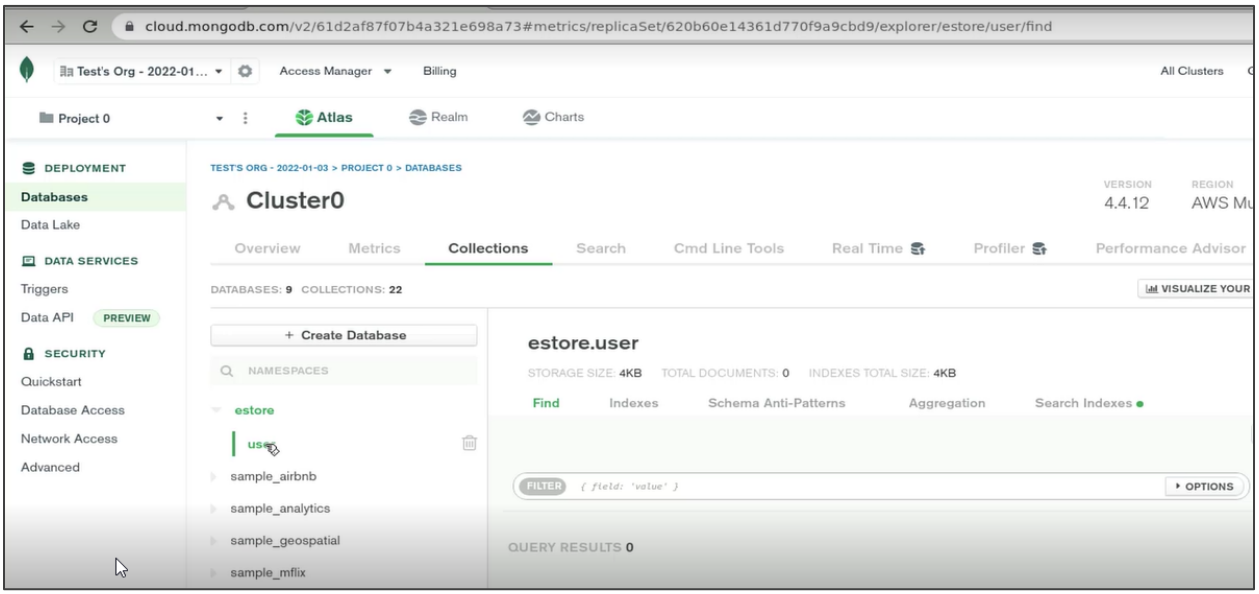


4.3 Create a new database by navigating to the **Collections** tab and clicking on **Create Database**



4.4 Name the database as **estore** and the collection as **user**. Now, click **Create**

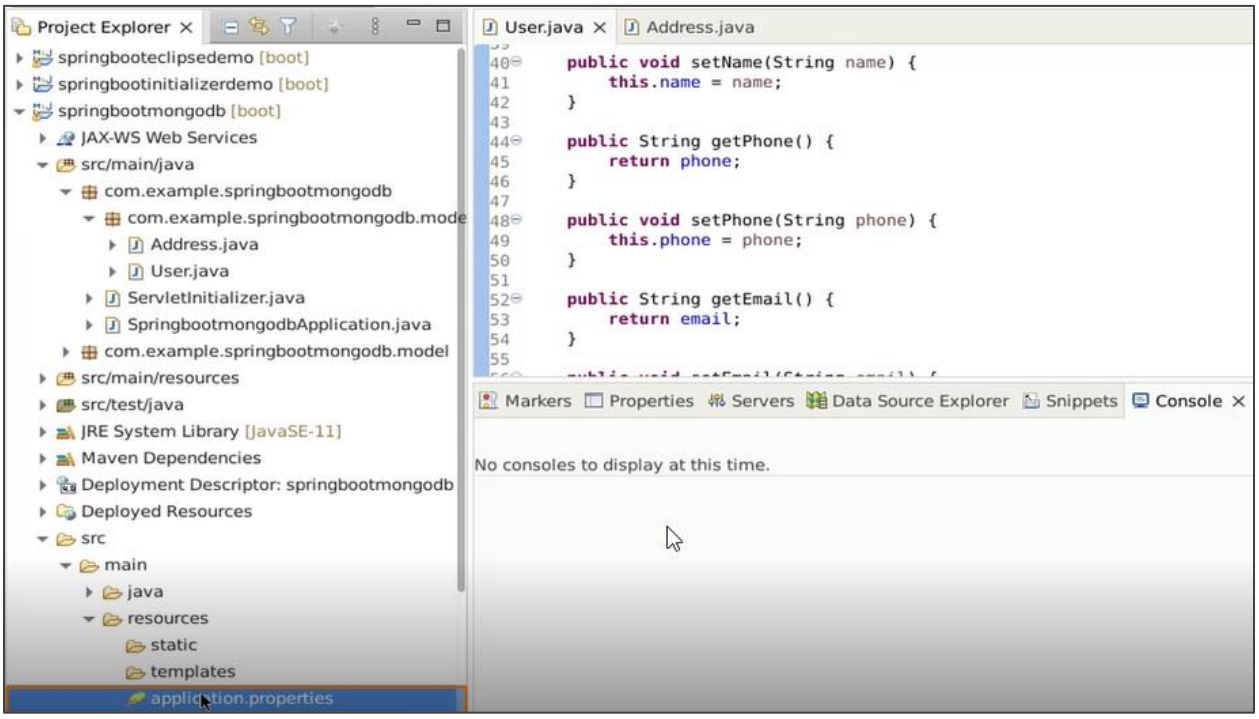




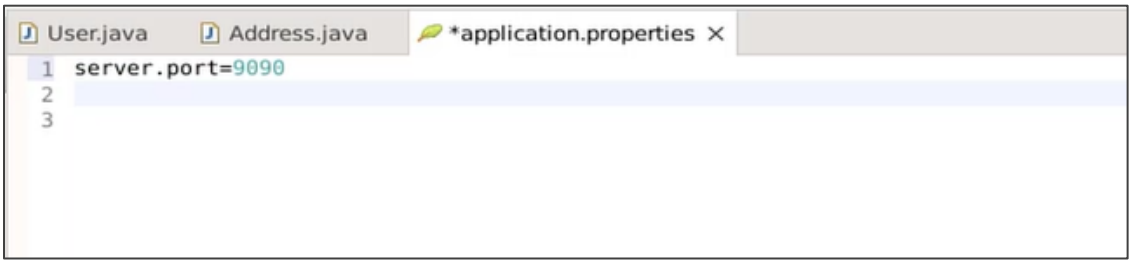
The **estore** database has been created, but currently, there are no records in it.

Step 5: Setting up the database configuration

5.1 Open the **application.properties** file located in the **src/main/resources** directory



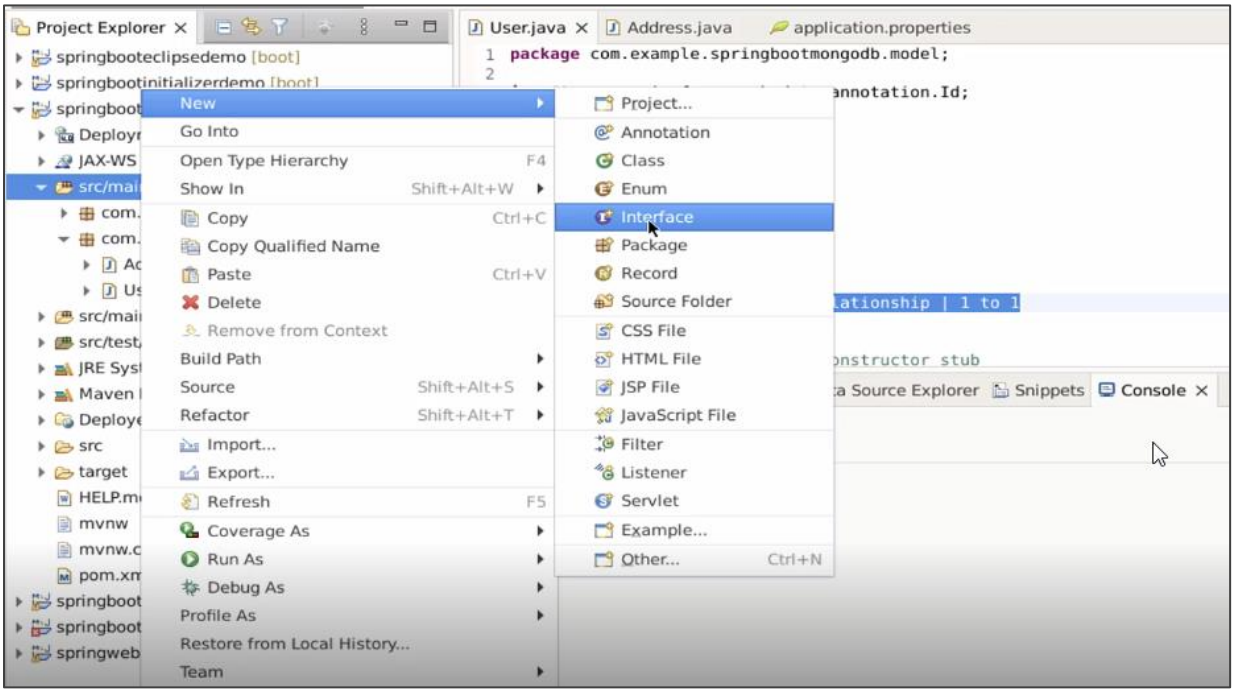
5.2 Configure the server port by adding the property **server.port=9090** (choose any port number)



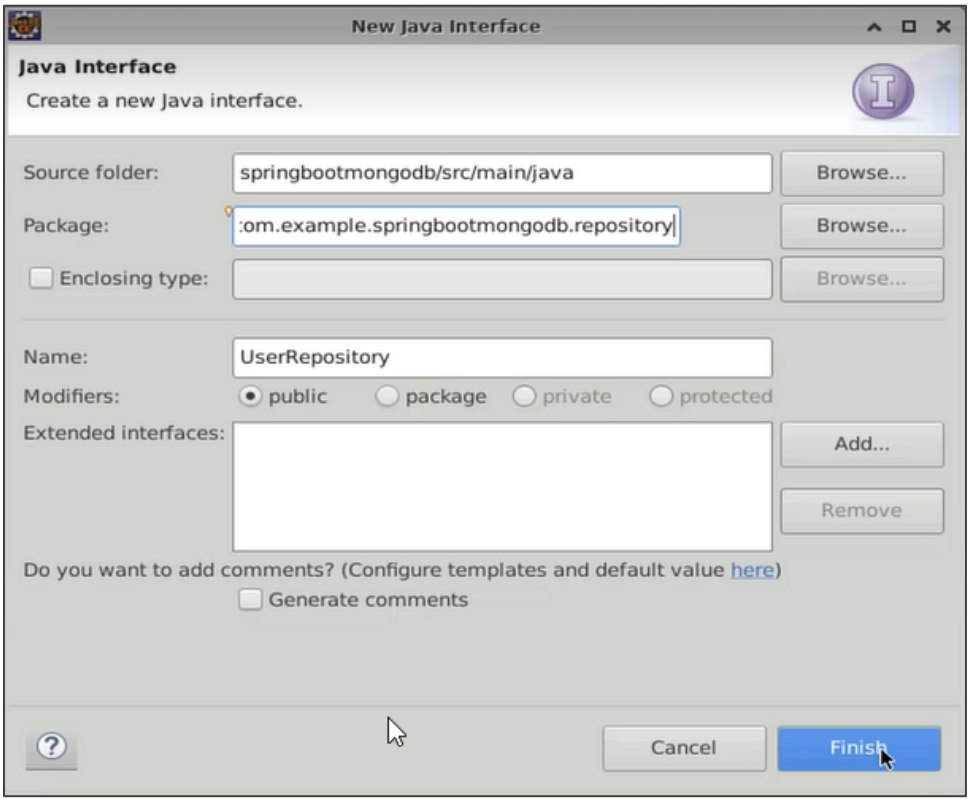
5.3 Specify the database name using **spring.data.mongodb.database** and set the MongoDB URI using **spring.data.mongodb.uri** (obtained from MongoDB Atlas)

Step 6: Creating the UserRepository interface

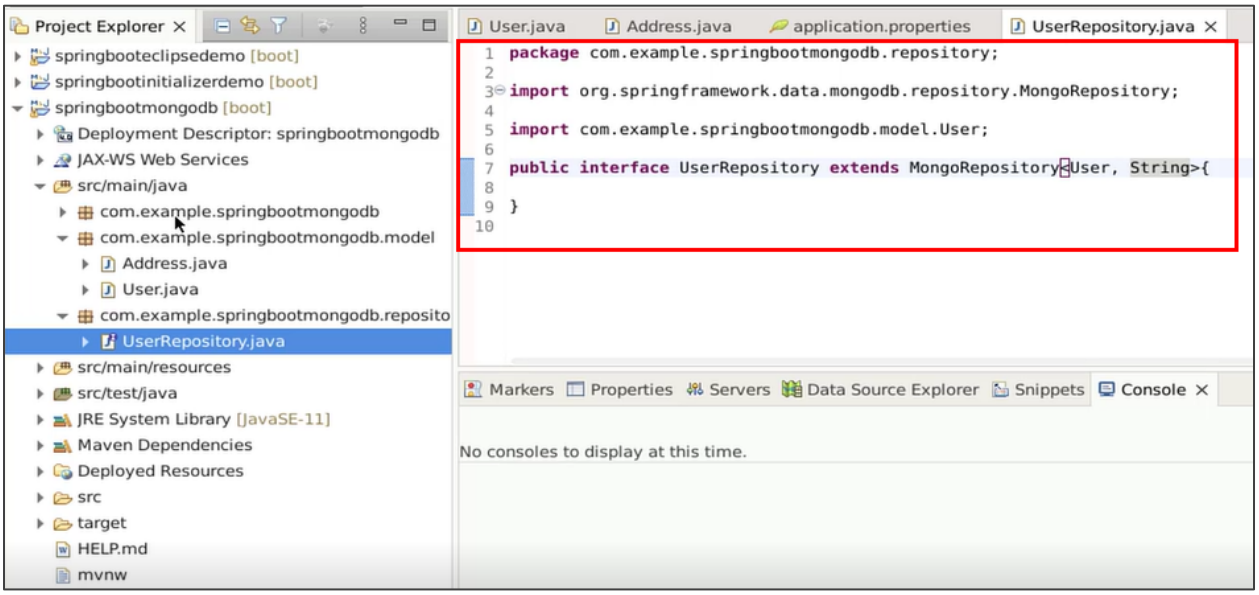
6.1 Right-click on the source package and select **New** > **Interface**



6.2 Name the interface **UserRepository** and add **.repository** to the package name and click **Finish**

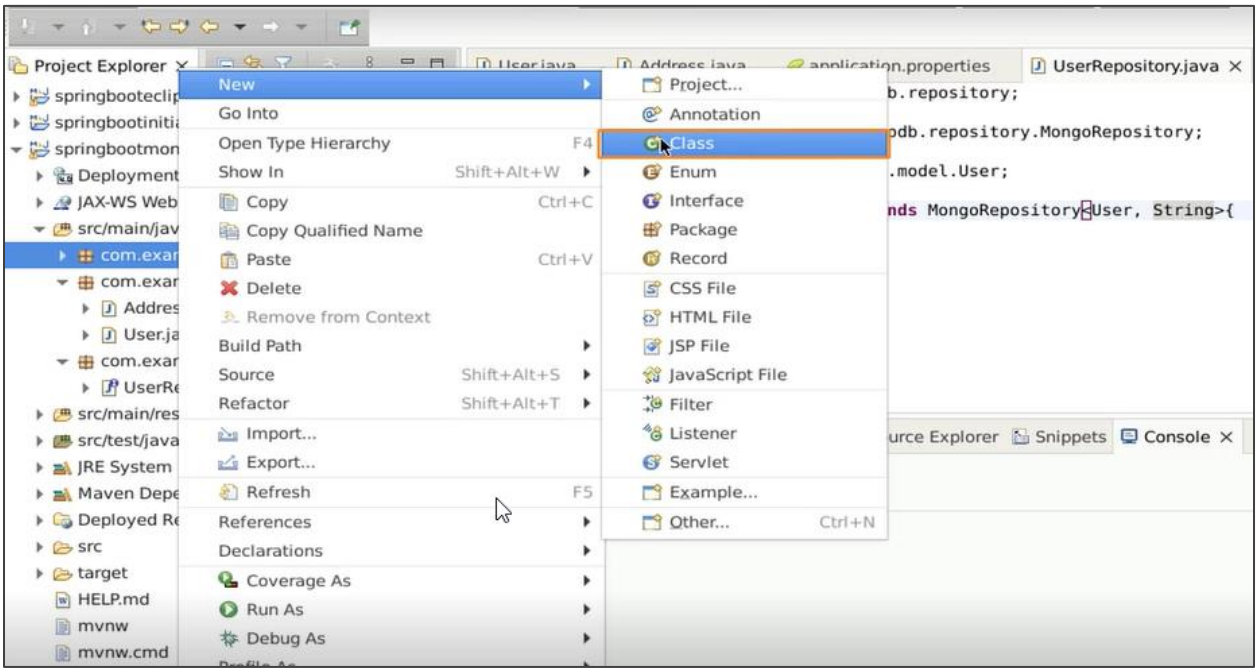


6.3 Extend the **MongoRepository<User, String>** interface to inherit the CRUD operations for the **User** entity

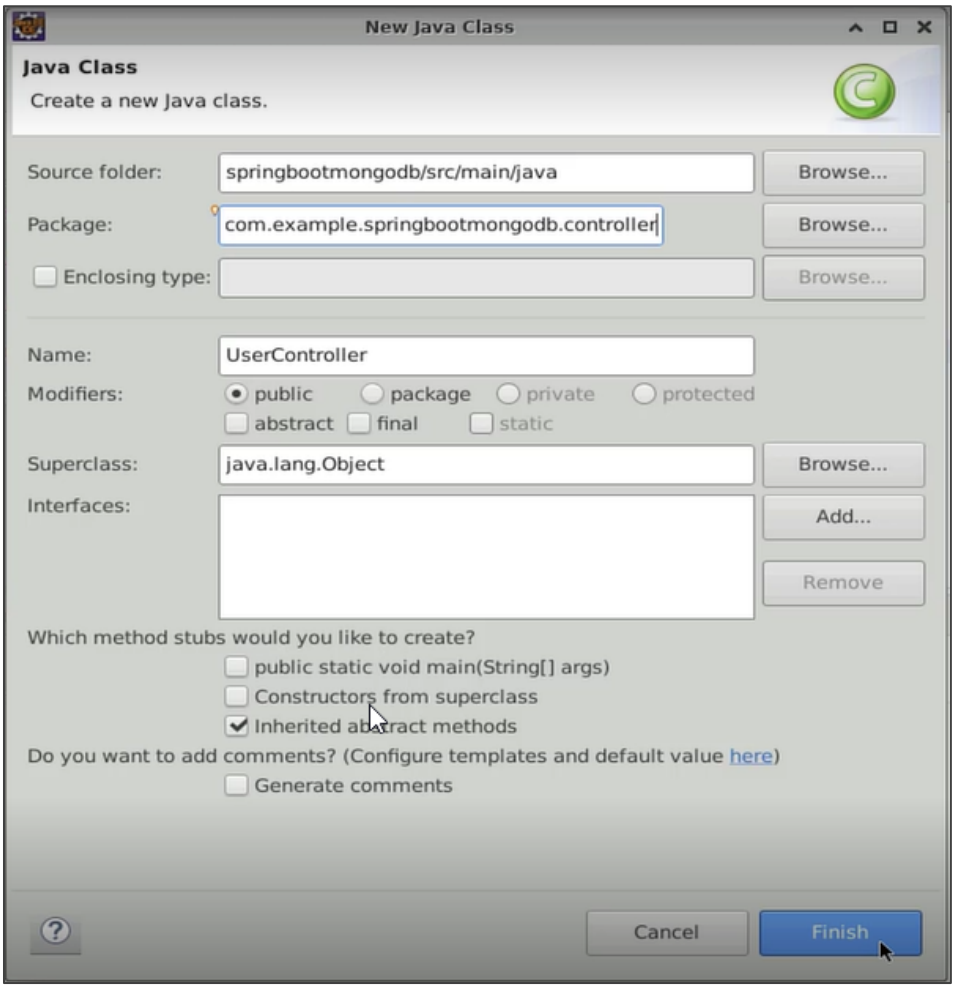


Step 7: Creating the UserController class

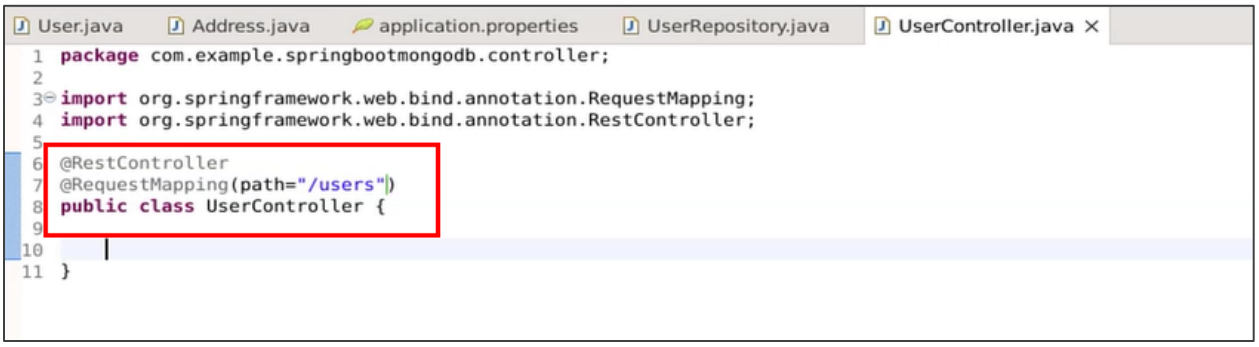
7.1 Right-click on the source package and select **New** > **Class**



7.2 Name the class **UserController** and add **.controller** to the package name and click **Finish**



7.3 Annotate the class with **@RestController** to indicate it is a controller for handling RESTful requests. Add **@RequestMapping** with the path set to **/users**

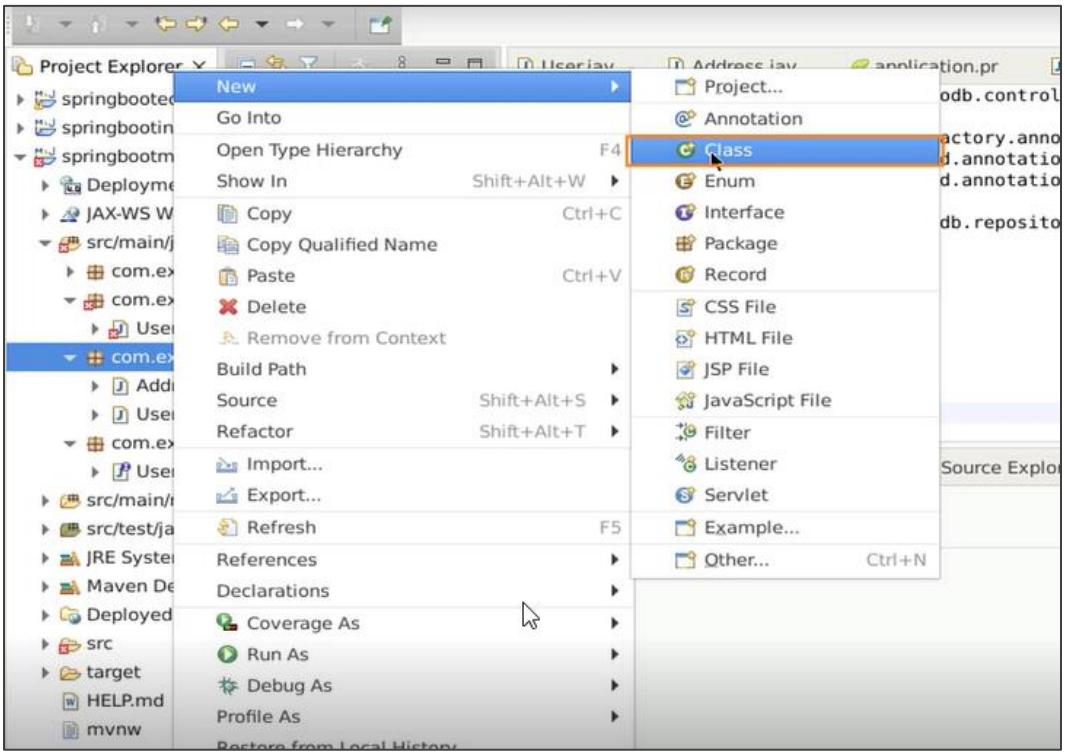


7.4 Autowire the **UserRepository** into the controller

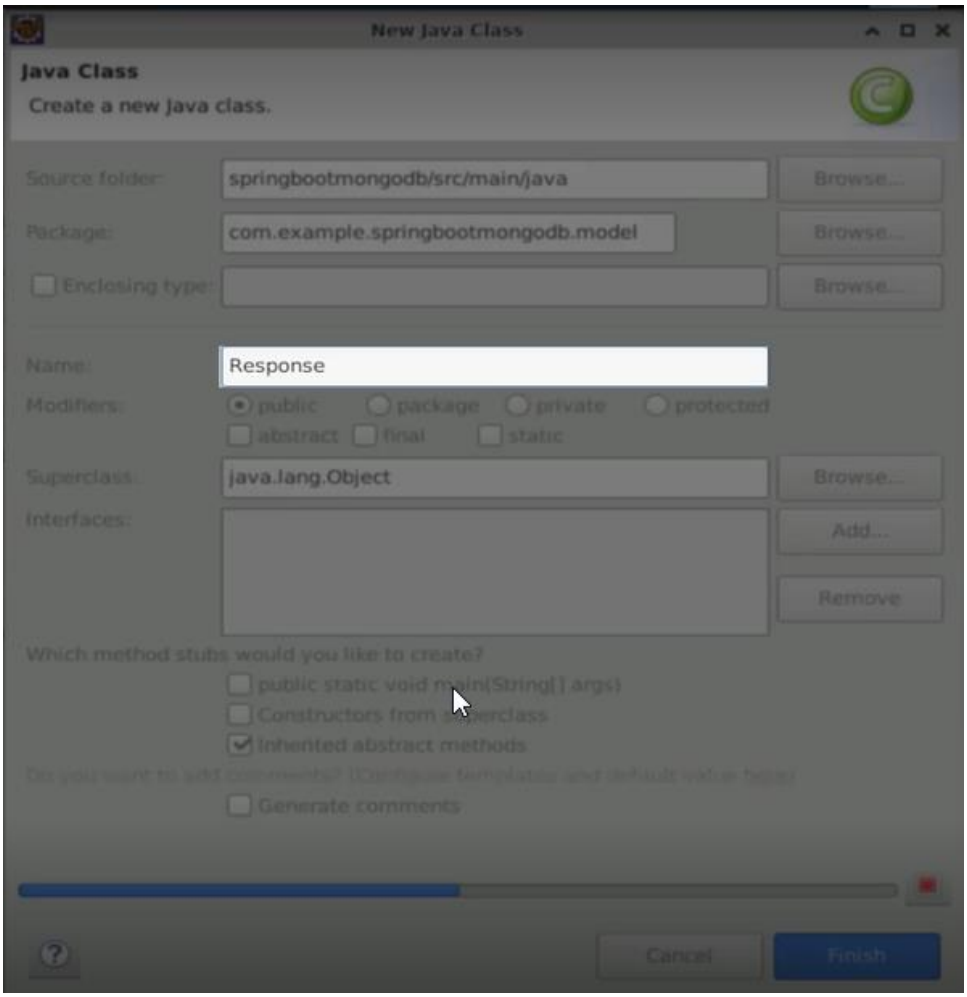
```java
package com.example.springbootmongodb.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.example.springbootmongodb.repository.UserRepository;

@RestController
@RequestMapping(path="/users")
public class UserController {

    @Autowired
    UserRepository repository;

}
```

Step 8: Creating the Response class

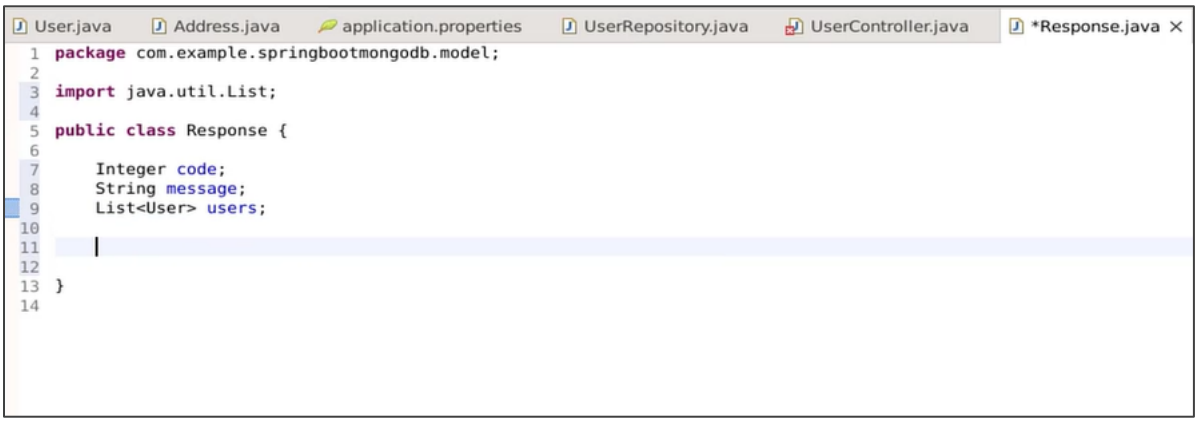
8.1 Right-click on the controller package and select **New** > **Class**



8.2 Name the class **Response** and click **Finish**



8.3 Define the class with the necessary fields for the response, such as **code, message,** and a **List<User>** for users



8.4 Generate the default constructor, parameterized constructor, getters, setters, and a **toString()** method for the Response class

User.java | Address.java | application.properties | UserRepository.java | UserController.java | *Response.java

```java
package com.example.springbootmongodb.model;

import java.util.List;

public class Response {

    Integer code;
    String message;
    List<User> users;

    public Response() {
        // TODO Auto-generated constructor stub
    }

    public Response(Integer code, String message, List<User> users) {
        this.code = code;
        this.message = message;
        this.users = users;
    }
```

User.java | Address.java | application.properties | UserRepository.java | UserController.java | *Response.java

```java
        this.message = message;
    }

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = code;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public List<User> getUsers() {
        return users;
    }

    public void setUsers(List<User> users) {
        this.users = users;
    }

    @Override
    public String toString() {
        return "Response [code=" + code + ", message=" + message + ", users=" + users + "]";
    }
```

Step 9: Configuring the CRUD methods

9.1 In the **UserController.java** class, add a method called **addUser** and annotate it with **@PostMapping** to set the endpoint as **/add**

```
User.java   Address.java   application.properties   UserRepository.java   UserController.java ×   Response.java

import com.example.springbootmongodb.model.Address;
import com.example.springbootmongodb.model.Response;
import com.example.springbootmongodb.model.User;
import com.example.springbootmongodb.repository.UserRepository;

@RestController
@RequestMapping(path="/users")
public class UserController {

    @Autowired
    UserRepository repository;

    @PostMapping(path="/add")
    public ResponseEntity<Response> addUser(@RequestParam String name, @RequestParam String phone, @RequestParam String email,
            @RequestParam String adrsLine, @RequestParam String city, @RequestParam String state, @RequestParam Integer zipCode) {
```

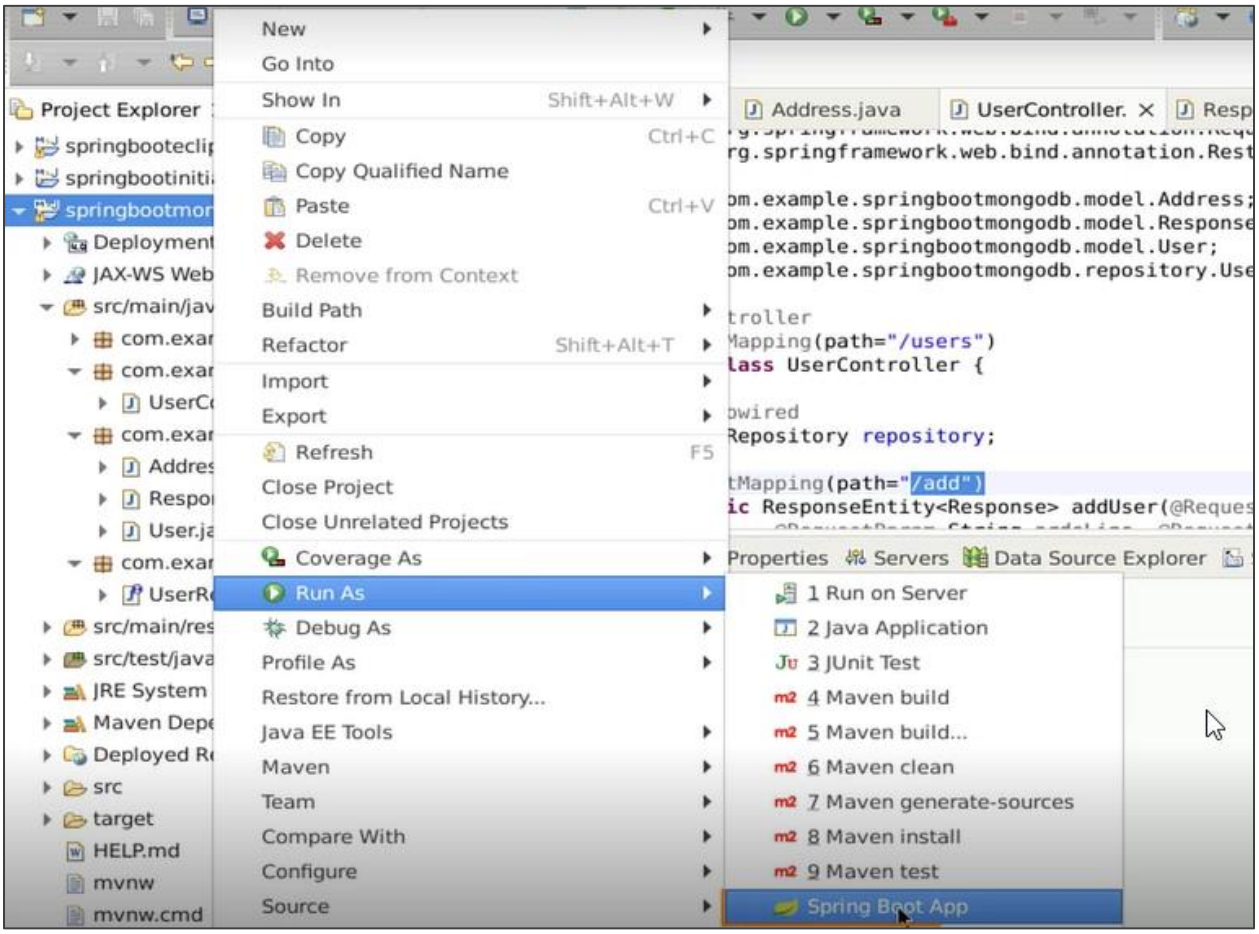
9.2 Implement the logic to save the user details in the database and return a status code of **101**. If any exceptions occur, catch them using a try-catch block and return a status code of **301**

```
User.java   Address.java   application.properties   UserRepository.java   UserController.java ×   Response.java

import com.example.springbootmongodb.model.Address;
import com.example.springbootmongodb.model.Response;
import com.example.springbootmongodb.model.User;
import com.example.springbootmongodb.repository.UserRepository;

@RestController
@RequestMapping(path="/users")
public class UserController {

    @Autowired
    UserRepository repository;

    @PostMapping(path="/add")
    public ResponseEntity<Response> addUser(@RequestParam String name, @RequestParam String phone, @RequestParam String email,
            @RequestParam String adrsLine, @RequestParam String city, @RequestParam String state, @RequestParam Integer zipCode) {

        try {

            Address address = new Address(adrsLine, city, state, zipCode);
            User user = new User(null, name, phone, email, address);

            repository.save(user);

            Response response = new Response(101, " User "+name+" Saved Successfully at " + new Date());
            return new ResponseEntity<Response>(response, HttpStatus.INTERNAL_SERVER_ERROR);

        } catch (Exception e) {
            Response response = new Response(301, "OOPS!! Something Went Wrong: "+e.getMessage());
            return new ResponseEntity<Response>(response, HttpStatus.INTERNAL_SERVER_ERROR);
        }

    }
```

Similarly, you can implement other methods for update, delete, and read operations for the user collection.

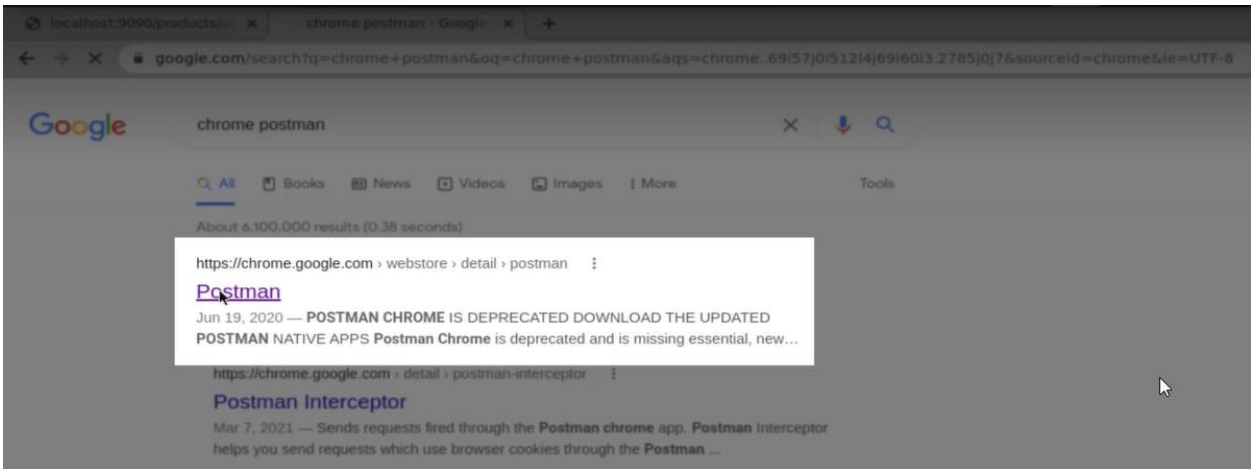
Step 10: Running and testing the application

10.1 Right-click on the project and select **Run As** > **Spring Boot App** to start the application

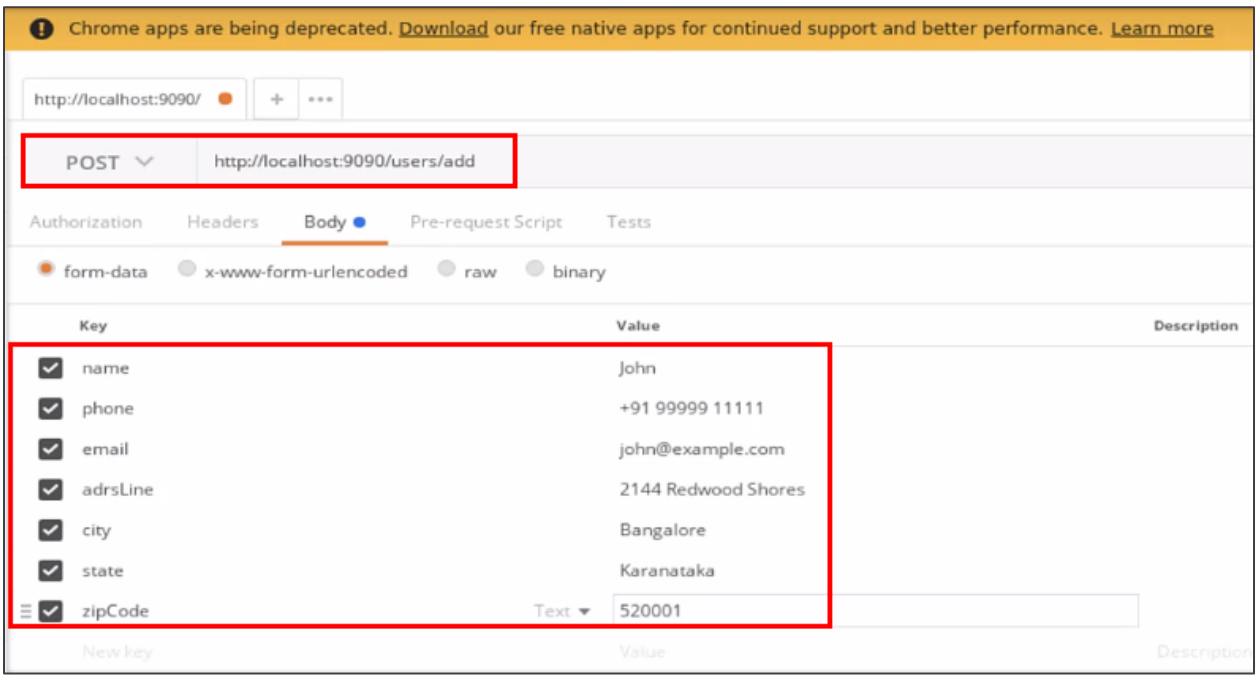


Spring Boot will automatically start the embedded Tomcat Server and deploy the application at **localhost:9090**

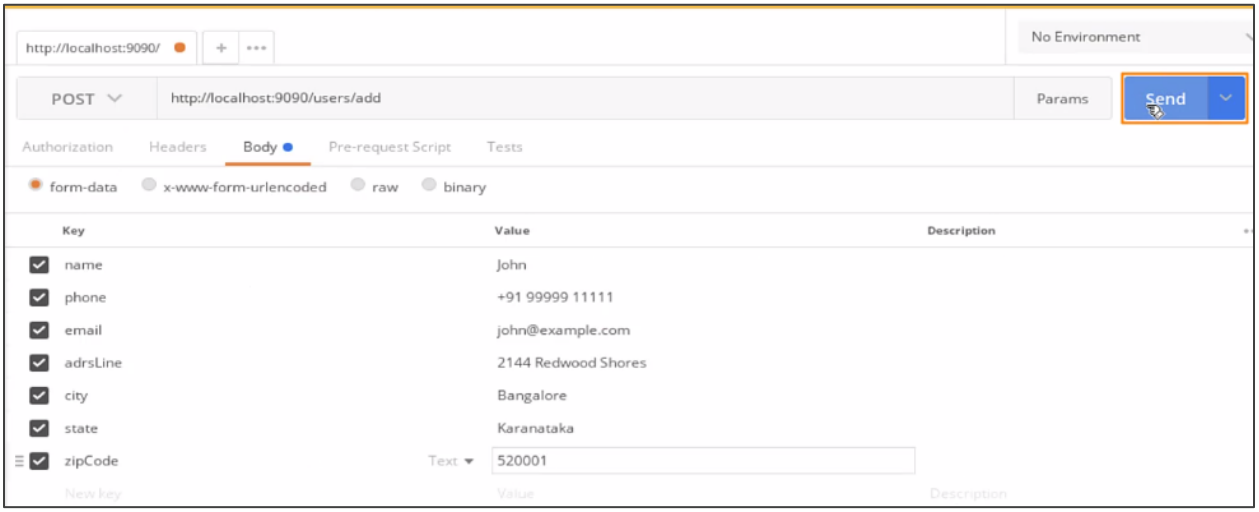
10.2 To test the **addUser** method, open Postman in a web browser

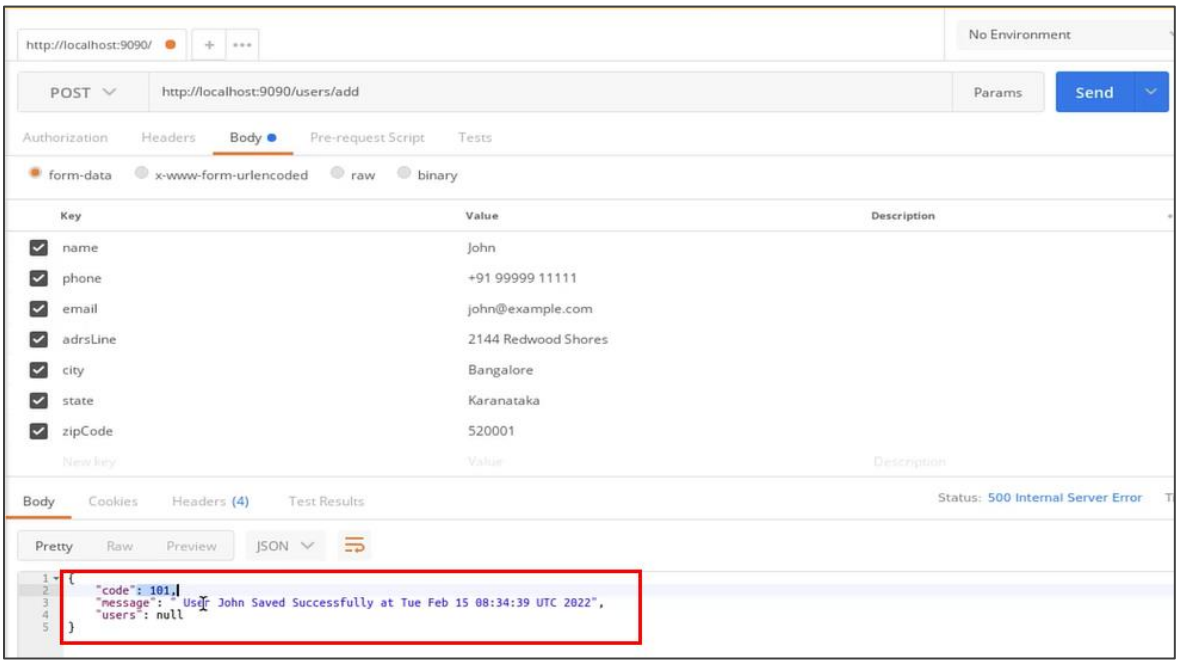


10.3 On the main page, send a POST request to the URL **http://localhost:9090/users/add**, which forwards the request to the **addUser()** controller method. Define the data for the user collection using key-value pairs under the **Body** section



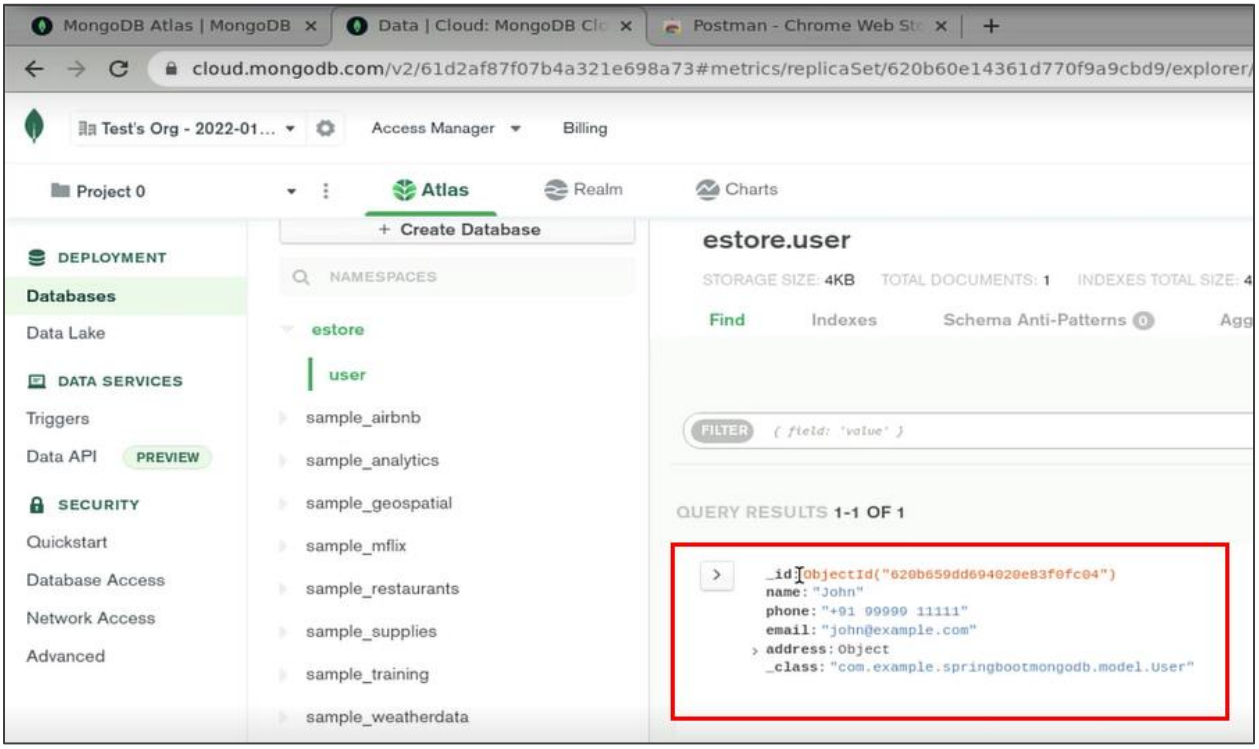
10.4 Click **Send**





You will see the JSON output with a message stating **User John Saved Successfully** and a status code of **101**.

10.5 Return to the user collection in the MongoDB Atlas and refresh the page



You can see that the user **John** has been added to the user collection as specified in Postman.

Similarly, you can implement other methods to read, update, and delete users in the **UserController.java** class.