Map Entity to DTO using ModelMapper

In enterprise applications, we use RESTful services to establish the communication between client and server. The general idea is that the client sends the request to the server and the server responds to that request with some response. Generally, we have three different layers in most of the application Web Layer, Business Layer, and Database Layer. The objects in these layers are mostly different from each other. For example, The web layer object is completely different from the same object in the database layer. As database objects may contain fields that are not required in the web layer object such as auto-generated fields, password fields, etc.

What is DTO?
DTO stands for Data Transfer Object, these are the objects that move from one layer to another layer. DTO can be also used to hide the implementation detail of database layer objects. Exposing the Entities to the web layer without handling the response properly can become a security issue. For example, If we have an endpoint that exposes the details of entities class called User. The endpoint handles the GET request. If the response is not handled properly with the GET endpoint one can get all the fields of the User class even the password also, which is not a good practice for writing restful services. To overcome these problems DTO came into the picture, with DTO we can choose which fields we need to expose to the web layer.

ModelMapper is a maven library which is used for the conversion of entities object to DTO and viceversa.

In this example, we will create a restful application for user services that uses model mapper library conversion of entity to DTO.

Create Entity Class

Now, we need to create our entity class. For our application, we will be using the User class as our entity class. It will have the following fields id, name, email, and password.

User.java

```
@Entity
public class User {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name="user-id")
	private int id;
	@Column(name="user-name")
	private String name;
	@Column(name="user-email")
	private String email;
	@Column(name="user-password")
	private String password;

}
```

Create a User Repository

In this step, we will create an interface and name it as UserRepository and extends this class to the JPA repository. So, we can have CRUD operations easily available.

UserRepository.java

```
import org.springframework.data.jpa.repository.JpaRepository;

import com.geeksforgeeks.ModelMapper.data.User;

// In the arguments we need to pass our model class and
// the second argument is the type of id we have used
// in our model class
public interface UserRepository extends JpaRepository<User, Integer> {

}
```

Create User Service

Now, we will create a service interface and name it as UserService. We will add only two methods to it. One to add the user and the other to get the user.

UserService.java

```
import com.geeksforgeeks.ModelMapper.data.User;

public interface UserService {

      public User createUser(User user);
      public User getUser(int userId);

}
```

After this, we will add the implementation of our user service interface.

UserServiceImpl.java

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

```
import com.geeksforgeeks.ModelMapper.data.User;
import com.geeksforgeeks.ModelMapper.repository.UserRepository;

@Service
public class UserServiceImpl implements UserService {

      @Autowired
      private UserRepository userRepository;

      @Override
      public User createUser(User user) {
            User userSavedToDB = this.userRepository.save(user);
            return userSavedToDB;
      }
      @Override
      public User getUser(int userId) {
            User user = this.userRepository.getById(userId);
            return user;
      }

}
```

Create a Controller

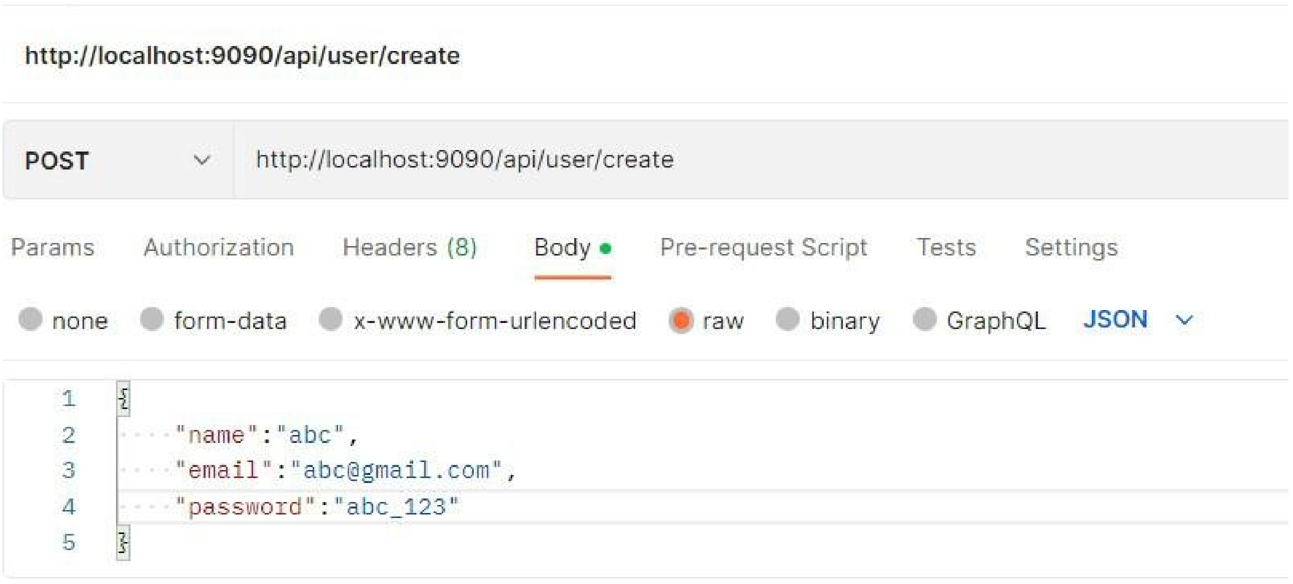
In this step, we will create a user controller for handling and mapping our request.

UserController.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.geeksforgeeks.ModelMapper.data.User;
import com.geeksforgeeks.ModelMapper.repository.UserRepository;

@Service
public class UserServiceImpl implements UserService {

      @Autowired
      private UserRepository userRepository;

      @Override
      public User createUser(User user) {
            User userSavedToDB = this.userRepository.save(user);
            return userSavedToDB;
      }

      @Override
      public User getUser(int userId) {
            User user = this.userRepository.findById(userId).get();
            return user;
      }

}
```

Run the Application

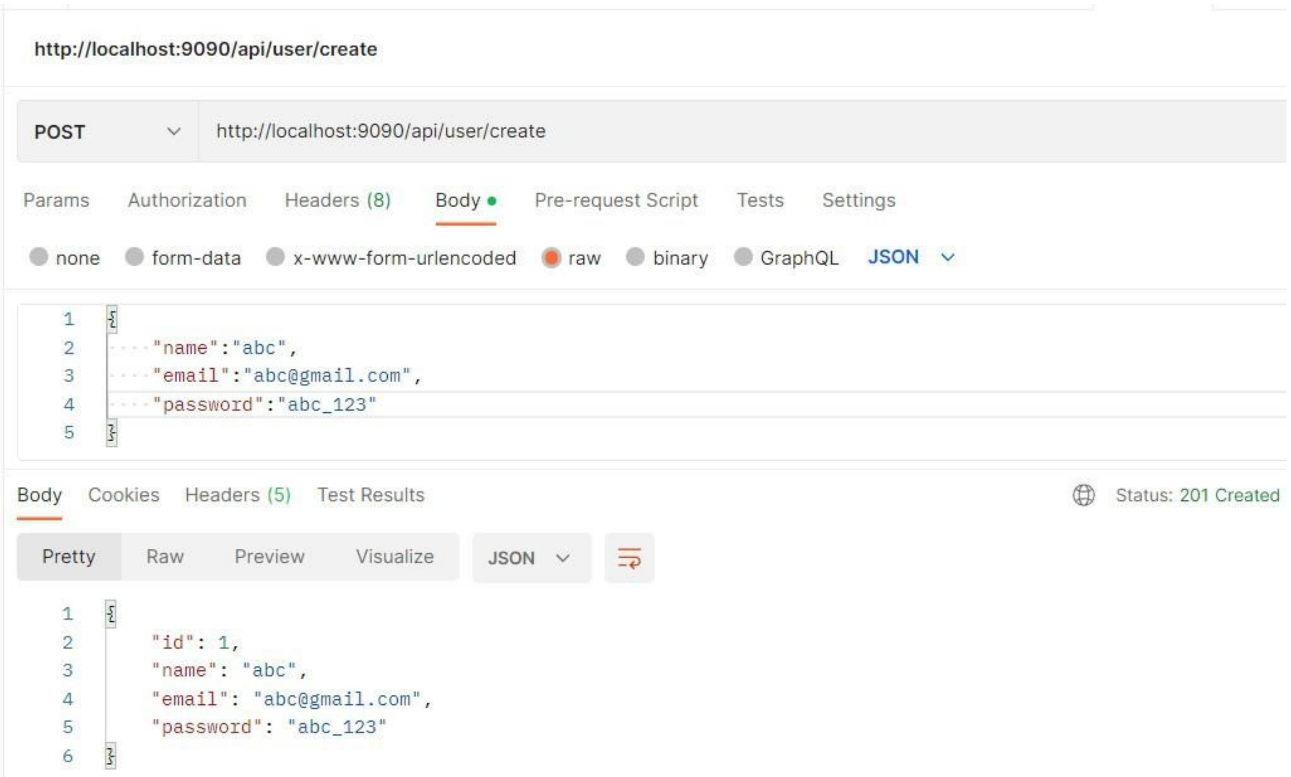
In this step, we will run our application using postman and test our restful services.


http://localhost:9090/api/user/create

POST http://localhost:9090/api/user/create

```
{
    "name":"abc",
    "email":"abc@gmail.com",
    "password":"abc_123"
}
```

Once, we send our request. We will get the following output.



http://localhost:9090/api/user/create

POST http://localhost:9090/api/user/create

Params | Authorization | Headers (8) | Body ● | Pre-request Script | Tests | Settings

none | form-data | x-www-form-urlencoded | ● raw | binary | GraphQL | JSON

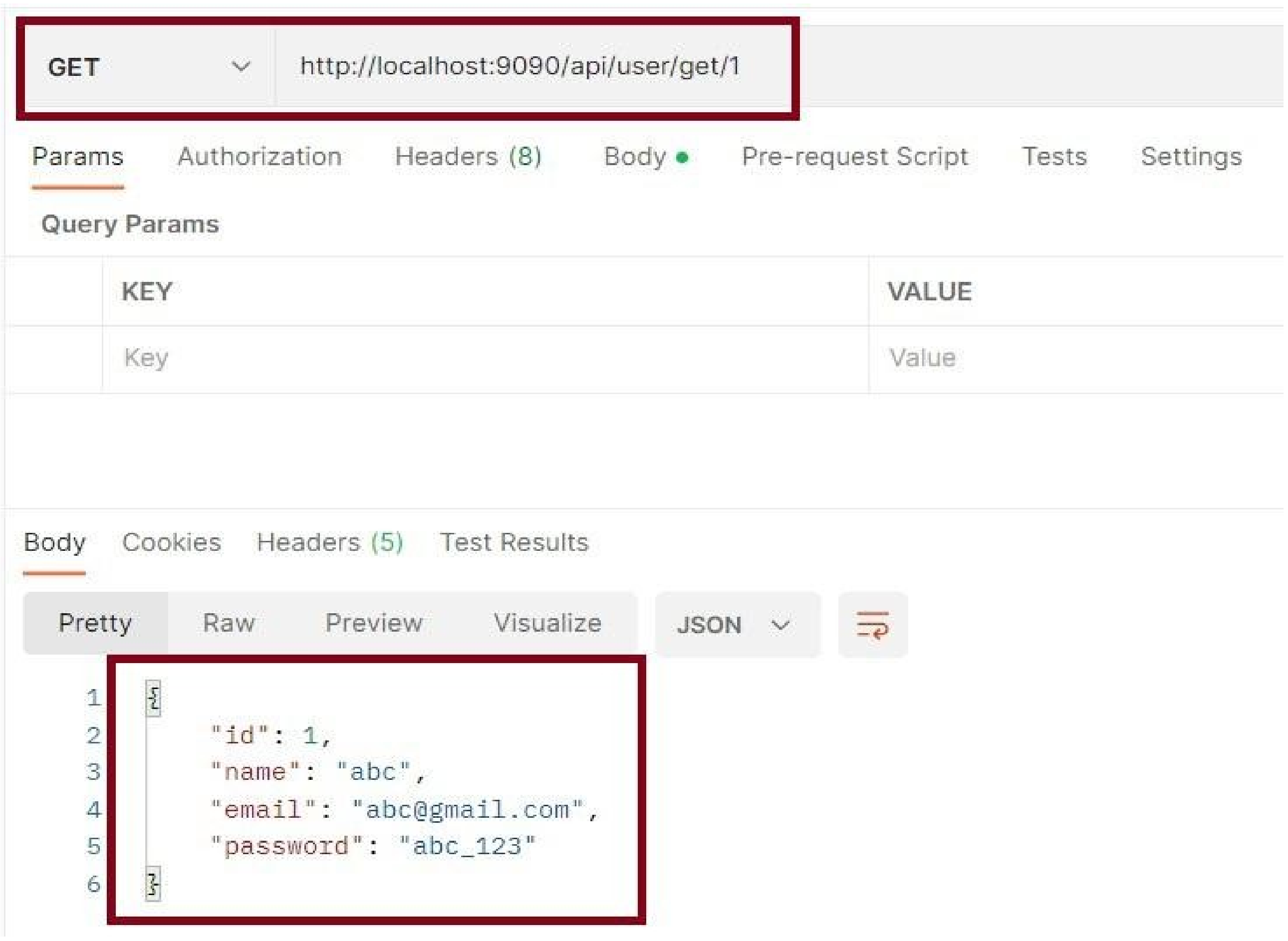
```
{
    "name":"abc",
    "email":"abc@gmail.com",
    "password":"abc_123"
}
```

Body | Cookies | Headers (5) | Test Results — Status: 201 Created

Pretty | Raw | Preview | Visualize | JSON

```
{
    "id": 1,
    "name": "abc",
    "email": "abc@gmail.com",
    "password": "abc_123"
}
```

We will be using the GET endpoint and userid to retrieve users from the database.



GET http://localhost:9090/api/user/get/1

Params | Authorization | Headers (8) | Body ● | Pre-request Script | Tests | Settings

Query Params

KEY	VALUE
Key	Value

Body | Cookies | Headers (5) | Test Results

Pretty | Raw | Preview | Visualize | JSON

```
{
    "id": 1,
    "name": "abc",
    "email": "abc@gmail.com",
    "password": "abc_123"
}
```

As we can see in the above response, we will also receive passwords which is not a good practice to write restful APIs. To overcome this problem we will use DTO.

Create DTO

In this step, we will create UserDTO class that will contain only those fields that are required and necessary for the web layer.

UserDto.java

```
public class UserDto {
      private String name;
      private String email;

}
```

Adding Model Mapper Dependency

We need to add the following dependency in our pom.xml file.

```
<dependency>
            <groupId>org.modelmapper</groupId>
            <artifactId>modelmapper</artifactId>
            <version>3.1.1</version>
</dependency>
```

Modify Classes

Now, to use UserDto, we need to modify our UserService, UserServiceImpl, and UserController class.

UserService.java

```
import com.geeksforgeeks.ModelMapper.data.User;
import com.geeksforgeeks.ModelMapper.dto.UserDto;

public interface UserService {

      public User createUser(User user);

      // updated it with UserDto
      public UserDto getUser(int userId);

}
```

UserServiceImpl.java

```
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.geeksforgeeks.ModelMapper.data.User;
import com.geeksforgeeks.ModelMapper.dto.UserDto;
import com.geeksforgeeks.ModelMapper.repository.UserRepository;
```

```java
@Service
public class UserServiceImpl implements UserService {

	@Autowired
	private UserRepository userRepository;

	@Autowired
	private ModelMapper modelMapper;

	@Override
	public User createUser(User user) {
		User userSavedToDB = this.userRepository.save(user);
		return userSavedToDB;
	}

	// update it with UserDto
	@Override
	public UserDto getUser(int userId) {
		User user = this.userRepository.findById(userId).get();
		UserDto userDto = this.modelMapper.map(user, UserDto.class);
		return userDto;
	}

}
```

UserController.java

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.geeksforgeeks.ModelMapper.data.User;
import com.geeksforgeeks.ModelMapper.dto.UserDto;
import com.geeksforgeeks.ModelMapper.service.UserServiceImpl;

@RestController
@RequestMapping("/api/user")
public class UserController {

	@Autowired
	private UserServiceImpl userServiceImpl;

	@PostMapping("/create")
	public ResponseEntity<User> createUser(@RequestBody User user){
		User userCreated = this.userServiceImpl.createUser(user);
		return new ResponseEntity<User>(userCreated, HttpStatus.CREATED);
	}

	// update it with UserDto
	@GetMapping("/get/{id}")
	public ResponseEntity<UserDto> getUser(@PathVariable("id") int userId){
		UserDto userDto = this.userServiceImpl.getUser(userId);
		return new ResponseEntity<UserDto>(userDto, HttpStatus.OK);
	}

}
```

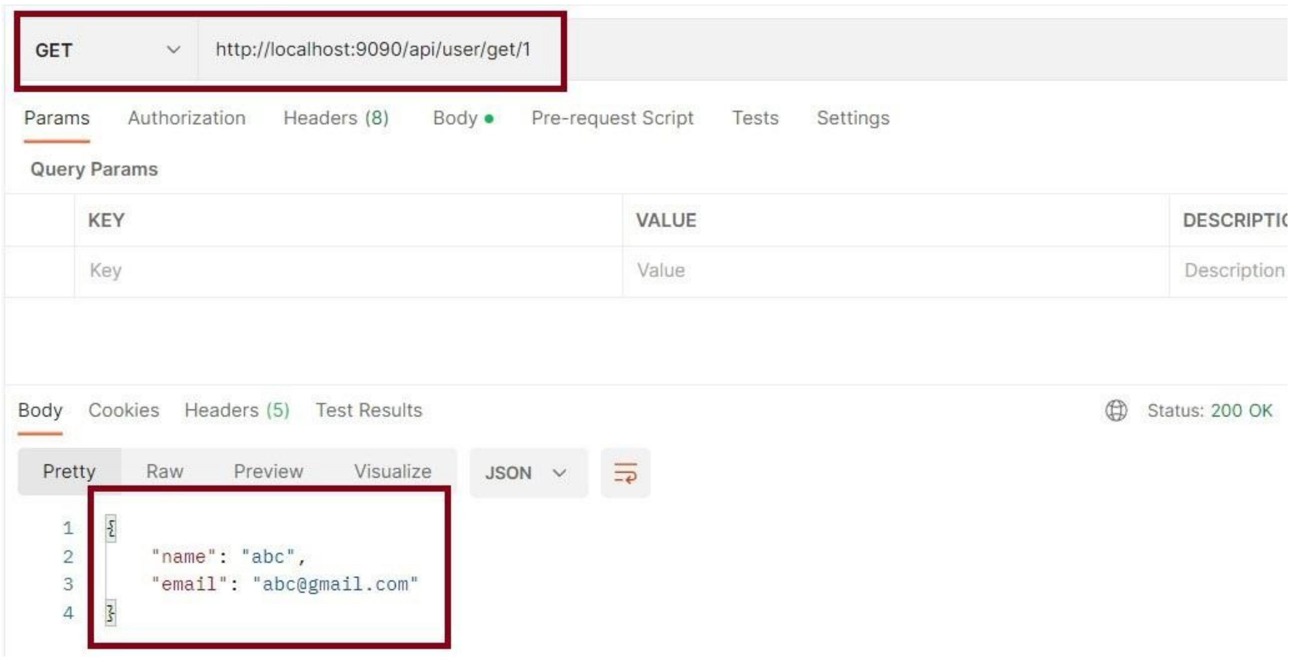
Add Model Mapper Bean

In this step, we will add model mapper bean to our main spring boot class.

```
import org.modelmapper.ModelMapper;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class ModelMapperApplication {

      public static void main(String[] args) {
            SpringApplication.run(ModelMapperApplication.class, args);
      }

      @Bean
      public ModelMapper getModelMapper() {
            return new ModelMapper();
      }

}
```

Run the Application

Now, we will again run our application and use the GET endpoint to see the response.



```
{
    "name": "abc",
    "email": "abc@gmail.com"
}
```

As we can see in the above response, we are getting only the necessary fields required by the web layer. Although, we will create a new User with all the fields but will send only the required fields to the web layer.