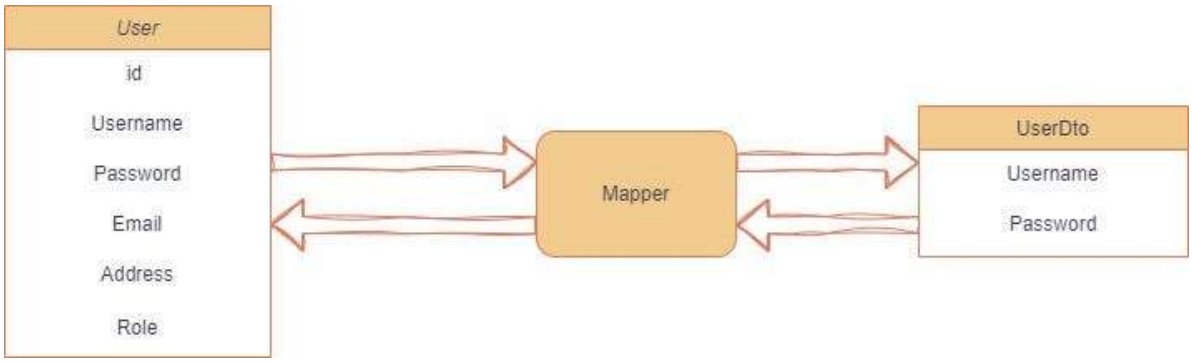
Understanding DTO in Java Spring Boot: A Step-by-Step Tutorial



Introduction

The topic of DTOs is among the important subjects in the software world and carries significant importance. Now, I will explain the best practices related to DTOs and several ways to implement DTOs. Afterwards, we will cover why we should use DTOs, the benefits they provide, and in which situations we should use them. Regardless of your level of expertise, I believe this article will be very useful. If you're ready, let's get started.

Why should we use DTO ?

In our Spring Boot application, we have an entity layer that contains important fields. In an application that does not use DTOs, these entities interact directly with the service and controller layers to perform necessary operations. This can pose security risks and other issues for various reasons. This is not an ideal situation for us. What we should do instead is create a class that is almost identical to the original entity class and use this new class to transfer data. This class is called a Data Transfer Object (DTO).

Advantages of Using DTOs

1. **Data Encapsulation**: By introducing a DTO layer instead of interacting directly with entities, data integrity is preserved and security is improved.

2. **Layer Independence**: DTOs help reduce dependencies between controller and service layers, allowing classes to work more independently and efficiently.

3. **Data Transfer Optimization**: when DTOs only include the necessary information, data transfer becomes faster and more efficient. This approach avoids transferring unnecessary data, which boosts performance.

Two Approaches to Implementing DTOs

1.Using Libraries: Two of the most popular libraries for mapping are ModelMapper and MapStruct. These libraries facilitate the mapping process by automating the conversion between objects. ModelMapper provides dynamic and flexible mapping, but it might be slower in terms of performance. However, if your project is not very large, the performance difference between ModelMapper and MapStruct may not be important. On the other hand, MapStruct performs mapping at compile-time, generally offering better performance. Pick one of them according to your project needs.

2.Manual Mapping: Another approach is to create a `mapper` class directly in your code and perform the conversion manually. This method is preferred by some because it avoids adding external dependencies to your application. While this approach is usually fine for personal projects, it might not be supported by your your company's policies if they restrict the use of external libraries.

1.Manuel Mapping

Now, we need to create our entity classes first. We must decide which fields in our entity class should be exposed, and this depends on our application requirements. For example, let's create a User and a UserDto class. In UserDto, I only want to expose the username and password fields, so I will include only these fields in my class.

```
@Entity
@Table(name = "tbl_user")
@Getter
@Setter
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String username;
    private String password;
    private String fullName;
    private String email;
    private String address;
}
@Getter
@Setter
public class UserDto {
    private String username;
    private String password;
}
```

Now that we have created our User and UserDto classes, we can perform the Dto conversions in our UserMapper class.

```
public class UserMapper {
    public static User toEntity(UserDto userDto){   // UserDto -> User
        if (userDto == null){
            return null;
        }
        User convUser = new User();
```

```
        convUser.setUsername(userDto.getUsername());
        convUser.setPassword(userDto.getPassword());

        return  convUser;
    }

    public static UserDto toDto(User user){      // User -> UserDto
        if(user == null){
            return null;
        }
        UserDto userDto = new UserDto();
        userDto.setUsername(user.getUsername());
        userDto.setPassword(user.getPassword());

        return userDto;
    }
}
public UserDto createUser(User user){
      User myUser = userRepository.save(user);
      return UserMapper.toDto(myUser);
    }
public List<UserDto> getAllUser(){
        List<User> users = userRepository.findAll();
        return  users.stream()
              .map(user ->UserMapper.toDto(user))
              .collect(Collectors.toList());
}
```

In the code above, the DTO conversion is done manually. The reason the conversion methods in the `UserMapper` class are defined as static is that this allows us to call these methods directly using the class name. This way, we can easily use `toDto` or `toEntity` methods within our code to perform the conversions.

You will see that using **ModelMapper** can make this process much simpler. Depending on the specifics of our project scenario, we can choose to use either approach.

2.ModelMapper

To integrate **ModelMapper** into our code, we first need to add its dependency to the `pom.xml` file and then create a bean in the `Configuration` class. After that, we can inject it into our classes using dependency injection and use it directly.

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.1.1</version>
</dependency>
@Configuration
public class ModelMapperConfig {

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

}
```

Now that we have added the necessary dependencies, I will show you how ModelMapper works within a service class.

```
public class UserService{

  private final ModelMapper modelmapper;    //Dependency Injection

  public UserService(ModelMapper modelMapper) {
        this.modelMapper = modelMapper;
    }
}
public UserDto createUser(User user){
            User tempUser = userRepository.save(user);
            UserDto userDto = modelMapper.map(tempUser,UserDto.class);
            return userDto;
    }
```

```
public List<UserDto> getAllUser(){
        List<User> users = userRepository.findAll();
        return users.stream()
            .map(user -> modelMapper.map(user, UserDto.class))
            .collect(Collectors.toList());
  }
```

In the code examples above, you can see different use cases for DTOs: the `createUser`method handles a single object, while the `getAllUser` method returns a list. The difference in the code arises from this. When using ModelMapper, after calling `modelMapper.map`, the first parameter is the class of the existing object, and the second parameter is the class you want to convert to. ModelMapper handles these conversions for you, making it straightforward to map DTOs to your classes.