17-DTO & ModelMapper

Author: Vincent Lau

Note: This material is intended for educational purposes only. All rights reserved. Any unauthorized sharing or copying of this material, in any form, to any individual or party, for any use without prior permission, is strictly prohibited.

Learning Objectives

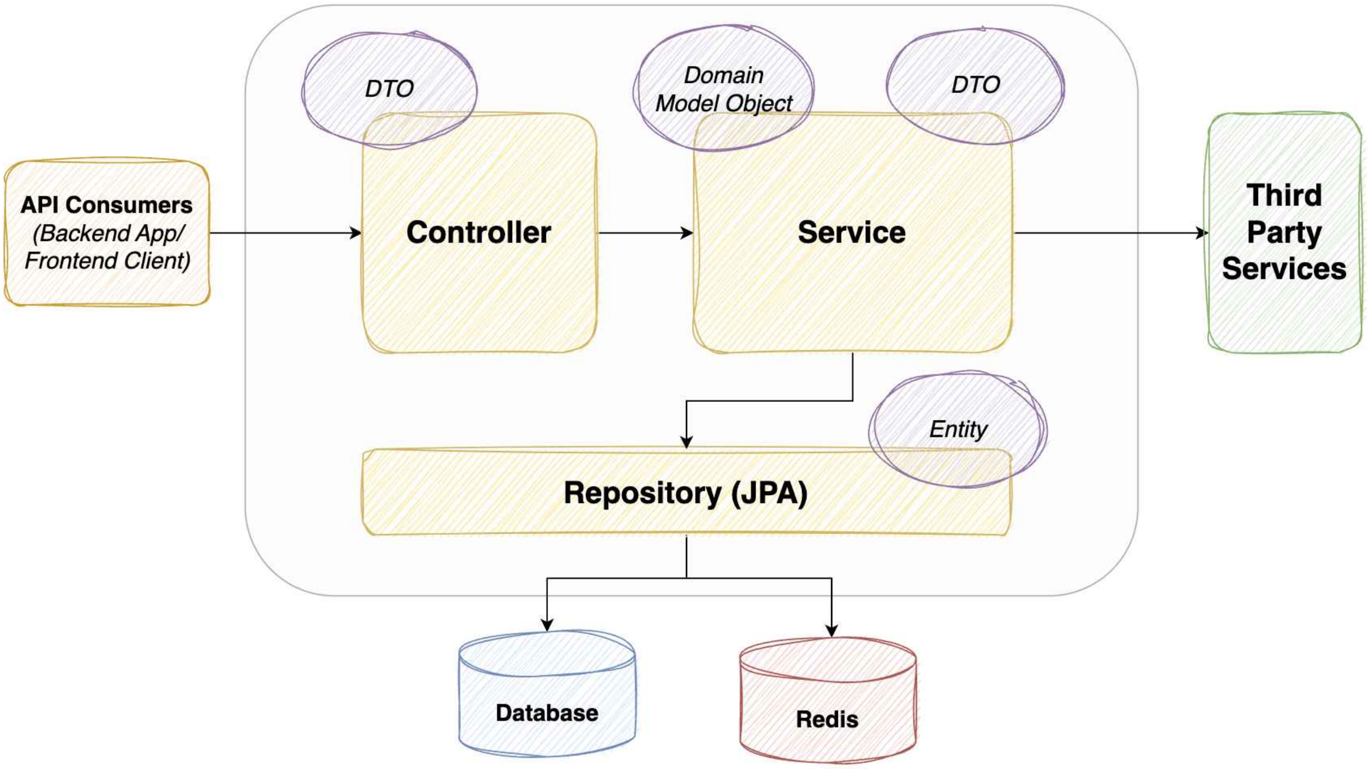
> Understand the usage of Data Transfer Object (DTO) and its benefits

> How to use it in Spring Boot Project.

> What is ModelMapper and how convenient it is.
>
> Design Mapper Class & it's Methods, converting between DTO, Domain Object & Entity.
>
> Three types of Mapper Class - MapStruct, ModelMapper, Custom Mapper

Introduction

In Spring Boot development, a Data Transfer Object (DTO) is a design pattern used to transfer data between the application's layers or different parts of the application. It is particularly useful when you want to decouple the internal domain model (used within the application) from the data that needs to be sent to clients or external systems, such as a frontend application or a RESTful API.



Key concepts and benefits of using DTOs

Separation of Concerns

DTOs help separate the concerns of your application by providing a clear distinction between the internal domain model and the data presented to external clients. This decoupling ensures that changes to the internal model don't impact the external interfaces.

In this example, we'll demonstrate how DTOs separate the concerns by transferring data from a domain object to a DTO.

```java
// Domain object
public class User {
    private Long id;
    private String username;
    private String email;
    // Getters and setters
}

// Data Transfer Object (DTO)
public class UserDTO {
    private String username;
    // Getters and setters
}
```

Reduced Overhead

When you send domain objects directly to clients, you might expose sensitive information or unnecessary details. DTOs allow you to include only the data that is relevant to the client, reducing data transfer overhead.

In the above example, you use a DTO to reduce data transfer overhead by including only the necessary data. For instance, consider a situation where the `User` entity has a lot of additional fields, but we only want to expose the username.

Data Transformation

DTOs enable you to transform complex domain objects into simpler structures tailored to the needs of clients. For instance, you can flatten nested objects, remove computed fields, or format data differently.

DTOs allow you to transform data to suit client needs. In this example, we format a `Date` into a user-friendly string for the client.

```
public class EventDTO {
    private String name;
    private String location;
    private String formattedDate; // Formatted date for the client

    public void setEventDate(Date date) {
        // Format the date as a string for the client
        this.formattedDate = new SimpleDateFormat("yyyy-MM-dd").format(date);
    }
}
```

Versioning and Evolution

As your application evolves, the structure of your domain objects may change. DTOs help you maintain backward compatibility with existing clients by providing consistent data structures even as the internal model changes.

As your domain model evolves, DTOs can help maintain backward compatibility. Let's consider a change in the `User` entity where a new field, `phoneNumber`, is added.

```
// Updated User entity
public class User {
    private Long id;
    private String username;
    private String email;
    private String phoneNumber; // New field
    // Getters and setters
}

// Updated UserDTO to maintain backward compatibility
public class UserDTO {
    private String username;
    private String email;
    // Getters and setters
}
```

Validation and Security

You can apply specific validation and security checks on the data within DTOs to ensure data integrity and enforce business rules before processing it.

DTOs allow you to validate data before processing. For instance, you can ensure that certain fields are not empty before saving data.

```
public class RegistrationRequestDTO {
    private String username;
    private String password;
    private String email;

    public boolean isValid() {
        return !username.isEmpty() && !password.isEmpty() && !email.isEmpty();
    }
}
```

Optimized Querying

DTOs can be used to fetch data from your database or other sources more efficiently by retrieving only the necessary fields.

You can use DTOs to fetch data more efficiently from the database by selecting only the necessary fields. In this example, we use a DTO to select specific fields from the `User` entity for a search operation.

```
// User entity with many fields
public class User {
    private Long id;
    private String username;
    private String email;
    private String address;
    private Date registrationDate;
    // Getters and setters
}

// UserSearchResultDTO for optimized querying
public class UserSearchResultDTO {
    private Long id;
    private String username;
    private String email;
}
```

Implement DTO between Controller & Service Layers

In a Spring Boot application, Data Transfer Objects (DTOs) are commonly used in both the Controller layer and the Service layer, each serving different purposes and contributing to the separation of concerns in your application. Let's explore how DTOs are typically used in both layers:

1. Controller Layer

Request Handling

In the Controller layer, DTOs are primarily used for receiving and responding to client requests. When a client sends data to the server, the incoming data is often in the form of a DTO.

Request Validation

DTOs can be used to validate incoming data, ensuring it conforms to the expected format and meets the application's business rules.

Response Transformation

After processing a request, the Controller may use DTOs to transform the data from the internal domain model into a format that is suitable for the client. This transformation ensures that the client receives only the necessary data in the desired structure.

Request and Response Example

```
@RestController
@RequestMapping("/users")
```

```
public class UserController {
    @Autowired
    private UserService userService;

    @PostMapping
    public ResponseEntity<UserDTO> createUser(@RequestBody UserRequestDTO userRequest) {
        // Controller receives a UserRequestDTO from the client and validates it
        // Calls the UserService to create a user
        UserDTO createdUser = userService.createUser(userRequest);
        // Transforms the result into a UserDTO before sending the response
        return ResponseEntity.ok(createdUser);
    }
}
```

2. Service Layer

Business Logic

In the Service layer, DTOs are often used to transfer data between service methods and to contain the output of these methods. These DTOs can be used to separate the internal domain model from the service methods.

Data Transformation

DTOs may be employed to transform data between entities and to encapsulate the results of business operations, ensuring that the response is in a suitable format for the Controller layer.

Service Layer Example

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public UserDTO createUser(UserRequestDTO userRequest) {
        // Transform the UserRequestDTO into an internal User entity
        User user = UserMapper.mapToEntity(userRequest);

        // Perform business logic and create a user
        User createdUser = userRepository.save(user);

        // Transform the created User entity into a UserDTO for the Controller
        return UserMapper.mapToDTO(createdUser);
    }
}
```

In both the Controller and Service layers, using DTOs promotes a clear separation of concerns and ensures that data is correctly formatted and validated at the appropriate stage of request handling. This separation is crucial for building maintainable, secure, and efficient Spring Boot applications.

3. RestTemplate (Cross-Server Service Call)

When using `RestTemplate` to call a third-party API in a Spring Boot application, it's common practice to use DTOs (Data Transfer Objects) for both sending requests to the API and processing the responses received from the API. DTOs help in structuring the data appropriately and provide a clear separation of concerns.

Assuming you have a Spring Boot project set up, follow these steps:

Create the DTOs for Request and Response

Create DTOs for the request and response objects. In this example, we'll retrieve a list of resources.

```
// Request DTO
public class ResourceRequestDTO {
    // Request-specific fields, if needed
}

// Response DTO
public class ResourceResponseDTO {
    private String resourceId;
    private String resourceName;
    // Getters and setters
}
```

Create a Service to Make the API Call

Create a service that uses `RestTemplate` to make the API call. You can define a method that retrieves a list of resources from the API.

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

```

```java
@Service
public class ResourceService {
    private final String apiUrl = "https://api.example.com/resources";

    @Autowired
    private RestTemplate restTemplate;

    public List<ResourceResponseDTO> getResources(ResourceRequestDTO requestDTO) {
        ResponseEntity<ResourceResponseDTO[]> responseEntity = restTemplate.getForEntity(apiUrl, ResourceResponseDTO[].class);
        if (responseEntity.getStatusCode() == HttpStatus.OK) {
            return Arrays.asList(responseEntity.getBody());
        }
        // Handle other status codes or exceptions as needed
        return Collections.emptyList();
    }
}
```

Configure RestTemplate in Your Application

Make sure you have `RestTemplate` properly configured in your Spring Boot application. You can use `RestTemplateBuilder` to create and configure an instance of `RestTemplate`.

```java
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class RestTemplateConfig {

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build(); // or you can new RestTemplate();
    }
}
```

Use the Service in Your Controller

Finally, you can use the `ResourceService` in your controller to handle API requests and responses.

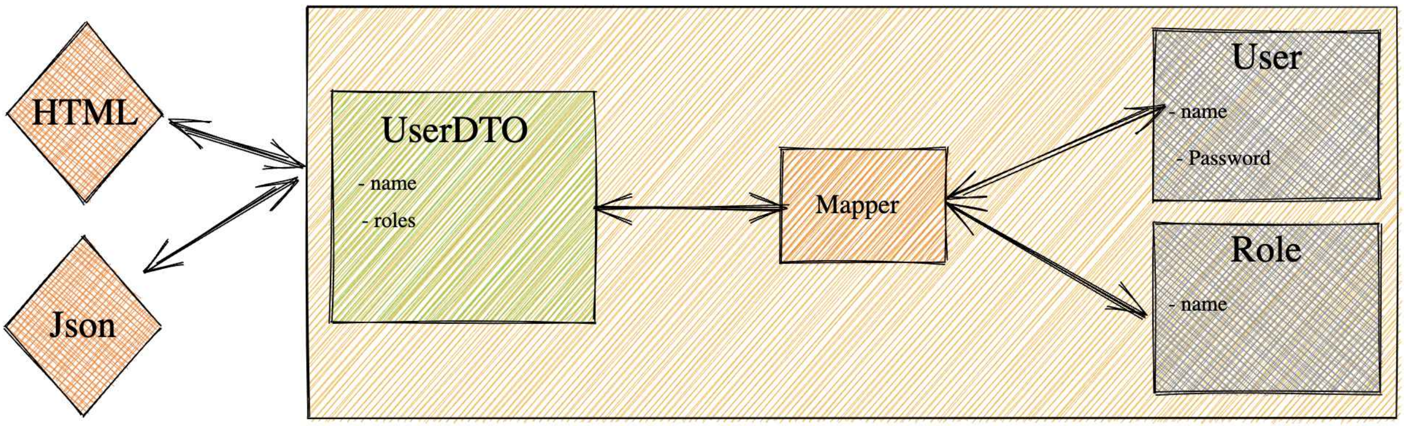
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/resources")
public class ResourceController {
    @Autowired
    private ResourceService resourceService;

    @GetMapping
    public List<ResourceResponseDTO> getResources(ResourceRequestDTO requestDTO) {
        return resourceService.getResources(requestDTO);
    }
}
```

In this example, the `ResourceService` makes a GET request to the fictional API's URL and retrieves a list of resources in the form of `ResourceResponseDTO`. You can customize the request and response handling based on your specific API's requirements.

Make sure to add the necessary dependencies for Spring Boot and `RestTemplate` to your `pom.xml` file.

Mapper For Data Transfer Object

When working with DTOs, you often need to convert between DTOs and domain entities or other data structures. This conversion can become complex, especially in applications with a large number of DTOs and entities. Mapper classes are used to address this complexity and provide a clear, structured way to convert data between different representations.



Model Mapper

ModelMapper is a Java library that simplifies the mapping of objects between different data structures, such as domain entities and DTOs (Data Transfer Objects). It's a popular choice for object-to-object mapping because it offers a high degree of flexibility and is easy to use.

Key Features of ModelMapper

Automatic Mapping

ModelMapper can automatically map fields with the same names and compatible types, reducing the need for manual mapping.

Custom Mapping

You can define custom mappings for specific fields or complex data transformations.

Nested Mapping

ModelMapper supports nested object mapping, making it easy to map complex object hierarchies.

Mapping Configuration

You can configure the behavior of the mapping process, such as specifying field names and type conversion strategies.

Code Example

First, make sure to include the ModelMapper library as a dependency in your project. You can add the following Maven dependency to your `pom.xml`:

```xml
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.4.0</version> <!-- Use the latest version available -->
</dependency>
```

Next, create a `Customer` entity class:

```java
public class Customer {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;

    // Getters and setters
}
```

Create a `CustomerDTO` class that represents the data transfer object:

```java
public class CustomerDTO {
    private String firstName;
    private String lastName;
    private String email;

    // Getters and setters
}
```

Now, let's use ModelMapper to map between the `Customer` entity and `CustomerDTO`:

```java
import org.modelmapper.ModelMapper;

public class CustomerMapperExample {
    public static void main(String[] args) {
        // Create a ModelMapper instance
        ModelMapper modelMapper = new ModelMapper();

        // Create a Customer entity
        Customer customer = new Customer();
        customer.setFirstName("John");
        customer.setLastName("Doe");
        customer.setEmail("john@example.com");

        // Use ModelMapper to map the Customer entity to CustomerDTO
        CustomerDTO customerDTO = modelMapper.map(customer, CustomerDTO.class);

        // Print the mapped CustomerDTO
        System.out.println("Mapped CustomerDTO: " + customerDTO);

        // You can also map in the reverse direction
        CustomerDTO anotherCustomerDTO = new CustomerDTO();
        anotherCustomerDTO.setFirstName("Jane");
        anotherCustomerDTO.setLastName("Smith");
        anotherCustomerDTO.setEmail("jane@example.com");

        Customer anotherCustomer = modelMapper.map(anotherCustomerDTO, Customer.class);

        // Print the mapped Customer
        System.out.println("Mapped Customer: " + anotherCustomer);
    }
}
```

In this example, we create a `ModelMapper` instance and use it to map a `Customer` entity to a `CustomerDTO` and vice versa. ModelMapper automatically maps fields with the same names and compatible types. Custom mappings can also be defined when needed.

ModelMapper simplifies the mapping process and is widely used in Java applications to reduce the manual effort required for mapping objects between different data structures.

MapStruct Mapper

MapStruct is a Java annotation-based code generation library that simplifies the mapping of objects between different data structures. It generates efficient mapping code during compilation, which eliminates the need for writing manual mapping code. MapStruct is highly customizable and supports various mapping scenarios, making it a popular choice for object-to-object mapping in Java applications.

Key Features of MapStruct

Annotation-Based

MapStruct uses annotations to define the mapping between objects, making it easy to use and understand.

Compile-Time Code Generation

Mapping code is generated at compile time, resulting in highly efficient and type-safe mappings.

Custom Mapping

You can provide custom mapping logic for specific fields or types.

Expression Mapping

MapStruct allows you to use expressions to define complex mappings.

Support for Complex Objects

It can handle complex objects and nested mappings, making it suitable for mapping hierarchies.

Code Example

First, make sure to include the MapStruct library as a dependency in your project. You can add the following Maven dependency to your `pom.xml`:

```xml
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>1.4.2.Final</version> <!-- Use the latest version available -->
</dependency>
```

Next, create a `Customer` entity class:

```java
public class Customer {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;

    // Getters and setters
}
```

Create a `CustomerDTO` class that represents the data transfer object:

```java
public class CustomerDTO {
    private String firstName;
    private String lastName;
    private String email;

    // Getters and setters
}
```

Now, let's create a MapStruct mapper interface for mapping between the `Customer` entity and `CustomerDTO`:

```java
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.factory.Mappers;

@Mapper
public interface CustomerMapper {
    // implicitly static final
    CustomerMapper INSTANCE = Mappers.getMapper(CustomerMapper.class);

    @Mapping(source = "firstName", target = "firstName")
    @Mapping(source = "lastName", target = "lastName")
    @Mapping(source = "email", target = "email")
    CustomerDTO map(Customer customer);

    @Mapping(source = "firstName", target = "firstName")
    @Mapping(source = "lastName", target = "lastName")
    @Mapping(source = "email", target = "email")
    Customer map(CustomerDTO customerDTO);
}
```

In this example, we define a `CustomerMapper` interface with two mapping methods: `customerToCustomerDTO` and `customerDTOToCustomer`. The `@Mapping` annotations specify the source and target fields for mapping.

Now, you can use the `CustomerMapper` to convert between `Customer` and `CustomerDTO` objects:

```
// Convert a Customer to a CustomerDTO
Customer customer = new Customer();
customer.setFirstName("John");
customer.setLastName("Doe");
customer.setEmail("john@example.com");

CustomerDTO customerDTO = CustomerMapper.INSTANCE.map(customer);

// Convert a CustomerDTO to a Customer
CustomerDTO customerDTO = new CustomerDTO();
customerDTO.setFirstName("Jane");
customerDTO.setLastName("Smith");
customerDTO.setEmail("jane@example.com");

Customer customer = CustomerMapper.INSTANCE.map(customerDTO);
```

MapStruct simplifies the mapping process and generates efficient mapping code during compilation. It's a powerful tool for handling object-to-object mapping in Java applications and helps reduce the manual effort required for mapping objects between different data structures.

Custom Mapper

Key Features of Custom Mapper

Separation of Concerns

Custom Mapper class, same as Mapstruct and ModelMapper, separates the logic of mapping data between DTOs and other objects, which promotes clean code and maintainability.

Reusability

Mapper classes can be reused across different parts of your application, avoiding code duplication.

Complex Mapping

Some mappings may be complex due to differences in data structure or business logic. Custom Mapper classes allow you to encapsulate this complexity.

Testing

Using mapper classes makes it easier to unit test the mapping logic in isolation from the rest of the application.

Flexibility

If you need to change the mapping logic or adapt to different scenarios (e.g., versioning changes), you can do so in one place in the mapper class.

Code Example

First, let's create a `Customer` entity class:

```java
public class Customer {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;

    // Getters and setters
}
```

Next, create a `CustomerDTO` class that represents the data transfer object:

```java
public class CustomerDTO {
    private String firstName;
    private String lastName;
    private String email;

    // Getters and setters
}
```

Now, let's create a custom `CustomerMapper` class for mapping between the `Customer` entity and `CustomerDTO`:

```java
public class CustomerMapper {
    public static CustomerDTO toCustomerDTO(Customer customer) {
        CustomerDTO customerDTO = new CustomerDTO();
        customerDTO.setFirstName(customer.getFirstName());
        customerDTO.setLastName(customer.getLastName());
        customerDTO.setEmail(customer.getEmail());
        return customerDTO;
    }

    public static Customer toCustomer(CustomerDTO customerDTO) {
        Customer customer = new Customer();
        customer.setFirstName(customerDTO.getFirstName());
        customer.setLastName(customerDTO.getLastName());
        customer.setEmail(customerDTO.getEmail());
        return customer;
    }
}
```

With this custom `CustomerMapper`, you can convert between `Customer` and `CustomerDTO` objects like this:

```java
// Convert a Customer to a CustomerDTO
Customer customer = new Customer();
customer.setFirstName("John");
customer.setLastName("Doe");
customer.setEmail("john@example.com");

CustomerDTO customerDTO = CustomerMapper.toCustomerDTO(customer);

// Convert a CustomerDTO to a Customer
CustomerDTO customerDTO = new CustomerDTO();
customerDTO.setFirstName("Jane");
customerDTO.setLastName("Smith");
customerDTO.setEmail("jane@example.com");

Customer customer = CustomerMapper.toCustomer(customerDTO);
```

This custom mapping approach is simple and doesn't require external libraries. However, keep in mind that for more complex mappings or large numbers of entities and DTOs, using a mapping library like MapStruct or ModelMapper can save you development time and help maintain cleaner code.

In Java, popular libraries like MapStruct and ModelMapper are often used to simplify the mapping between DTOs and domain objects. These libraries generate mapping code based on annotations or configuration, reducing the manual effort required for mapping.

To use mapper classes, you typically define interfaces or classes with methods for mapping specific DTOs to entities and vice versa. These methods encapsulate the logic of copying data from one object to another, and you can invoke them as needed in your application.

Overall, mapper classes for DTOs help manage the complexity of data transformation, promote clean code, and facilitate the development of robust and maintainable applications.