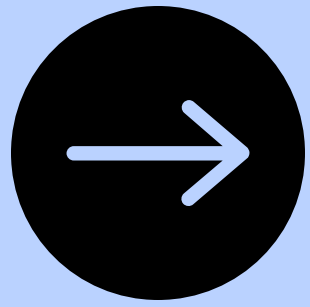
DATA TRANSFER OBJECTS (DTO)

SPRING BOOT

Rohan Thapa

thaparohan2019@gmail.com



What is a DTO?

A **Data Transfer Object (DTO)** is a simple, plain object that is used to **transfer data** between different parts of a system, particularly between the client and server layers in an application.
DTOs are often used to **encapsulate data**, ensuring that only the necessary information is exposed and transmitted.

Why use DTOs?

- **Data Encapsulation: DTOs encapsulate data** and ensure that only the required data is exposed to the client, **protecting sensitive information** from being inadvertently shared.
- **Decoupling:** DTOs decouple the internal data models (e.g., **entities in a database**) from the API or service layer, allowing flexibility in the API without affecting the underlying database schema.
- **Efficiency:** By sending only the necessary data, DTOs help reduce the payload size, making data transfer more efficient, especially over network requests.

Why use DTOs?

- **Validation:** DTOs can be used to validate incoming data before it is processed by the application, ensuring that only valid data reaches the business logic layer.

- **Security:** DTOs prevent over-posting attacks by allowing only specific fields to be updated, thus providing an additional layer of security.

When to Use DTOs?

- **API Development:** DTOs are commonly used in RESTful APIs to represent the data that should be sent to and received from clients.
- **Service Layer:** In a **service-oriented architecture**, DTOs are used to transfer data between different services or layers within an application.
- **Complex Data Mapping:** When data needs to be transformed or aggregated before being sent to the client, DTOs help in managing and structuring this data effectively.

Implement DTOs

Define the DTO Class:

- A DTO class typically contains only the fields that need to be transferred. It often includes **getters, setters**, and sometimes **validation annotations.**

```java
import jakarta.validation.constraints.NotBlank;
import lombok.Data;

@Data  10 usages
public class ClassRequestDto {
    @NotBlank(message = "Class name is required")
    private String className;

    @NotBlank(message = "Class Description is required")
    private String description;

    @NotBlank(message = "Academic year is required")
    private String academicYear;
}
```

Mapping Between Entities and DTOs:

- When using **DTOs**, data from an entity (e.g., a **JPA entity**) is mapped to the DTO before being sent to the client. Similarly, data from the client (DTO) is mapped back to the entity before processing or saving.
- This mapping can be done manually or using libraries like **MapStruct** or **ModelMapper**.

Mapping Between Entities and DTOs:

```
public ClassEntity mapDtoToEntity(ClassRequestDto dto){  1 usage
    ClassEntity entity = new ClassEntity();
    entity.setClassName(dto.getClassName());
    entity.setDesciption(dto.getDescription());
    entity.setAcademicYear(dto.getAcademicYear());
    return entity;
}

public ClassResponseDTO mapEntityToDto(ClassEntity entity){  4 usages
    ClassResponseDTO dto = new ClassResponseDTO();
    dto.setClassId(entity.getClassId());
    dto.setClassName(entity.getClassName());
    dto.setAcademicYear(entity.getAcademicYear());
    dto.setDescription(entity.getDesciption());
    return dto;
}
}
```

Mapping DTO to Entity and vice-versa

DTOs are used as **input parameters** or **return types** to manage the **data flow between the client and the server**.

```java
@Service
public class ClassServicesImpl implements ClassServices {

    private final ClassRepository classRepository;  7 usages

    public ClassServicesImpl(ClassRepository classRepository){  no usages
        this.classRepository = classRepository;
    }

    @Override  1 usage
    public ClassResponseDTO createClass(ClassRequestDto newClass){
        ClassEntity entity = mapDtoToEntity(newClass);
        ClassEntity savedResult = classRepository.save(entity);
        return mapEntityToDto(savedResult);
    }

    @Override  1 usage
    public ClassResponseDTO updateClass(Long classId, ClassRequestDto classRequestDto) {
        ClassEntity existingClass = classRepository.findById(classId).orElseThrow(()-> new ResourceNotFoundException("Class Not Found"));

        existingClass.setClassName(classRequestDto.getClassName());
        existingClass.setAcademicYear(classRequestDto.getAcademicYear());
        existingClass.setDesciption(classRequestDto.getDescription());

        ClassEntity updatedClass = classRepository.save(existingClass);
        return mapEntityToDto(updatedClass);
    }
```

Usage in Controllers:

In a RESTful controller, DTOs are used as **input parameters** or **return types** to manage the **data flow between the client and the server**.

```java
import java.util.List;

@RestController  no usages
@RequestMapping("/api/admin")
public class ClassController {

    private final ClassServices classServices;  6 usages

    public ClassController (ClassServices classServices){  no usages
        this.classServices = classServices;
    }

    @GetMapping("/class")  no usages
    public ResponseEntity<List<ClassResponseDTO>> getAllClasses(){
        return new ResponseEntity<>(classServices.getAllClass(), HttpStatus.OK);
    }

    @GetMapping("/class/{id}")  no usages
    public ResponseEntity<ClassResponseDTO> getClassById(@PathVariable Long id){
        return new ResponseEntity<>(classServices.getClassById(id), HttpStatus.OK);
    }

    @PostMapping("/class")  no usages
    public ResponseEntity<ClassResponseDTO> createClass(@Valid @RequestBody ClassRequestDto classRequestDto){
        return new ResponseEntity<>(classServices.createClass(classRequestDto),HttpStatus.OK);
    }
```

Best Practices with DTOs

- **Keep DTOs Simple:** DTOs should only contain fields that are necessary for the specific operation. Avoid adding business logic or complex methods.
- **Use Validation:** Use validation annotations (e.g., @NotNull, @Size) on DTO fields to ensure data integrity.
- **Automate Mapping:** Use mapping frameworks like **MapStruct** to reduce boilerplate code and avoid manual errors during mapping.
- **Separate Read and Write DTOs:** Consider using different DTOs for read and write operations, especially if the data requirements differ significantly.

DTOs in Different Contexts

- **Microservices:** DTOs are crucial in **microservices architecture** for **data exchange between services**. They help in defining clear contracts between services and managing versioning.

- **CQRS (Command Query Responsibility Segregation):** In CQRS, DTOs are used for read operations **(Query DTOs)** and write operations **(Command DTOs)** to separate concerns and optimize the data flow.

Common Pitfalls to Avoid

- **Overusing DTOs:** Not every data transfer requires a DTO. Overusing them can lead to unnecessary complexity.

- **Ignoring Validation:** Without proper validation, DTOs can **introduce vulnerabilities**, especially when accepting user input.

- **Incorrect Mapping:** Improper mapping between entities and DTOs can lead to **data loss or inconsistencies**. It's crucial to ensure that all necessary fields are correctly mapped.

Conclusion

DTOs are a powerful tool for managing data transfer in modern applications, especially when dealing with APIs, microservices, and complex data structures.

They provide a clean and efficient way to encapsulate data, reduce coupling between layers, and improve security and maintainability.

Properly using DTOs can significantly enhance the architecture and performance of your applications.

Thank You

Rohan Thapa

thaparohan2019@gmail.com