DTO - Data Transfer Object Mastery: Alta Performance e Otimização de Dados em Camadas Avançadas com Java

Em Java, o **Data Transfer Object (DTO)** é um padrão de projeto usado para transferir dados entre diferentes camadas de uma aplicação, especialmente entre a camada de apresentação (front-end) e a camada de serviço ou persistência (back-end). DTOs são objetos simples, geralmente compostos apenas de atributos e métodos getters/setters, sem lógica complexa.

Eles ajudam a reduzir o acoplamento entre as camadas e a melhorar a performance, transmitindo apenas os dados necessários.

Boas Práticas ao Utilizar DTOs em Java

1. **Imutabilidade**: Embora DTOs frequentemente utilizem getters e setters, adotar o padrão de DTOs imutáveis, onde os dados são passados através do construtor, pode aumentar a segurança e evitar modificações indesejadas.

   O uso de frameworks como Lombok pode facilitar a implementação de DTOs imutáveis, através da anotação @Value.

   ```
   @Value
   public class UserDTO {
       String name;
       int age;
   }
   ```

2. **Validação no DTO**: Para garantir que os dados transferidos estão dentro dos parâmetros esperados, é recomendável usar anotações de validação do **Bean Validation** (JSR-303) no DTO. Isso facilita a captura de erros logo nas camadas superiores.

   ```
   public class UserDTO {
       @NotNull
       private String name;

       @Min(18)
       private int age;

       // Getters e Setters
   }
   ```

3. **Conversões e Mapeamento**: O uso de bibliotecas como **MapStruct** ou **ModelMapper** facilita o mapeamento entre entidades de domínio (modelos do banco de dados) e DTOs, automatizando o processo de conversão e evitando código boilerplate.

   Exemplo com MapStruct

   ```
   @Mapper
   public interface UserMapper {
       UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);

       UserDTO toDTO(User user);
       User toEntity(UserDTO userDTO);
   }
   ```

4. **Segregação de DTOs**: Para evitar exposição desnecessária de informações, é importante criar DTOs específicos para diferentes contextos, como **Request DTOs** (para receber dados de requisições) e **Response DTOs** (para enviar dados de resposta), mantendo a segurança e clareza no fluxo de dados.

Boas Práticas nas Versões LTS do Java

Com a evolução das versões LTS (Long-Term Support) do Java, como as versões **Java 8**, **Java 11** e **Java 17**, surgiram diversas melhorias que podem ser aplicadas ao uso de DTOs para obter alta performance e manter a compatibilidade.

1. **Java 8 - Streams e Lambda Expressions**: O uso de **Streams API** e **expressões lambda** facilita a manipulação e conversão de coleções de entidades para DTOs. Isso resulta em um código mais conciso e eficiente.

   ```
   List<UserDTO> userDTOs = users.stream()
       .map(user -> userMapper.toDTO(user))
       .collect(Collectors.toList());
   ```

   Além disso, o uso de **Optional** pode ser aplicado ao retorno de DTOs, prevenindo o uso excessivo de verificações de null.

   ```
   Optional<UserDTO> userDTO = Optional.ofNullable(userMapper.toDTO(user));
   ```

2. **Java 11 - Strings e Tipos Primitivos Otimizados**: A versão Java 11 introduz várias melhorias em relação à manipulação de **Strings**, que pode ser bastante útil na formatação de dados dentro de DTOs. O método String::isBlank simplifica a verificação de strings vazias, por exemplo, em campos de formulários.

   ```
   if (userDTO.getName().isBlank()) {
       throw new IllegalArgumentException("Nome não pode estar em branco");
   }
   ```

Além disso, a versão 11 permite a utilização de tipos primitivos otimizados para manipulação de dados em DTOs, melhorando a eficiência de memória e performance.

3. **Java 17 - Sealed Classes e Pattern Matching**: Na versão Java 17, o uso de **Sealed Classes** pode ser interessante quando se deseja controlar que tipos de DTOs podem ser estendidos ou implementados, especialmente em cenários de segurança de dados e validação em arquiteturas mais complexas.

   ```
   public abstract sealed class AbstractDTO permits UserDTO, AdminDTO {
       // Definição de campos e métodos comuns
   }
   ```

   O **Pattern Matching for instanceof** também simplifica o tratamento condicional de tipos de DTO em várias partes do código, proporcionando maior legibilidade.

   ```
   if (obj instanceof UserDTO userDTO) {
       // Manipulação direta de userDTO
   }
   ```

4. **Utilização de Records (Java 16+)**: A partir da versão 16, e consolidada na versão 17 LTS, Java introduziu os **Records**, que são ideais para a criação de DTOs imutáveis com menos boilerplate. Isso simplifica a criação de DTOs e garante a imutabilidade sem a necessidade de frameworks adicionais como Lombok.

   ```
   public record UserDTO(String name, int age) {}
   ```

   Com Records, você obtém automaticamente os métodos toString, equals, hashCode, getters, além de construtores.

Conversões Possíveis

Existem várias conversões possíveis e úteis entre objetos no Java utilizando DTOs. As mais comuns são:

- **Entidade para DTO**: Para isolar a camada de domínio das camadas superiores, convertendo entidades para DTOs que serão enviados para o cliente.
- **DTO para Entidade**: Para criar ou atualizar uma entidade de domínio com base nos dados recebidos do cliente.
- **Listas e Coleções**: Muitas vezes, é necessário converter listas de entidades para listas de DTOs e vice-versa. Ferramentas como MapStruct lidam bem com essas conversões de coleções.

```
List<UserDTO> userDTOs = users.stream()
    .map(user -> userMapper.toDTO(user))
    .collect(Collectors.toList());
```

Exemplo Prático

Vamos considerar uma situação em que uma entidade User está sendo usada em um sistema, e o DTO correspondente é UserDTO.

Entidade User

```
public class User {
    private Long id;
    private String name;
    private int age;

    // Getters e Setters
}
```

DTO UserDTO

```
public class UserDTO {
    private String name;
    private int age;

    // Getters e Setters
```

Conversão Manual de User para UserDTO

```
public class UserConverter {

    public static UserDTO convertToDTO(User user) {
        UserDTO dto = new UserDTO();
        dto.setName(user.getName());
        dto.setAge(user.getAge());
        return dto;
    }

    public static User convertToEntity(UserDTO userDTO) {
        User user = new User();
        user.setName(userDTO.getName());
        user.setAge(userDTO.getAge());
        return user;
    }
}
```

Conversão Automática com MapStruct

```
@Mapper
public interface UserMapper {

    UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);

    UserDTO userToUserDTO(User user);

    User userDTOToUser(UserDTO userDTO);
}
```

Neste caso, a conversão seria realizada automaticamente pelo MapStruct, eliminando a necessidade de escrever código repetitivo.

Conclusão

O uso de DTOs em Java é uma prática essencial para melhorar a organização do código, separar responsabilidades entre camadas e otimizar o tráfego de dados. Seguir boas práticas, como a imutabilidade, a validação de dados e o uso de bibliotecas de mapeamento automático, garante que a implementação de DTOs seja eficiente e de fácil manutenção.

Além disso, o aproveitamento de recursos específicos das versões LTS do Java, como **Streams API**, **Records**, e **Pattern Matching**, aumenta a performance e moderniza as aplicações.

Este artigo proporciona uma visão técnica e detalhada, incluindo práticas modernas e recursos das versões LTS do Java, para desenvolvedores que buscam maximizar a performance e a eficiência em arquiteturas multicamadas.

EducaCiência FastCode para a comunidade.