Material descargable

egg

API REST con Spring

¿Qué es un patrón DTO?

El patrón DTO (Objeto de Transferencia de Datos) es una técnica utilizada para transportar información de manera eficiente entre diferentes capas de un sistema, sin lógica de negocio involucrada. Los DTO son objetos simples cuya única responsabilidad es transmitir datos de un punto a otro, por ejemplo, del cliente al servidor.

Una de las principales ventajas de los DTO es que permiten crear representaciones de datos independientes del modelo subyacente de la aplicación. Esto posibilita crear diferentes vistas para una misma entidad, proteger datos sensibles (como contraseñas) y evitar errores cuando el modelo de datos cambia.

Además, los DTO permiten recopilar datos de diversas fuentes y organizarlos de manera coherente. Por ejemplo, una solicitud puede contener información proveniente de varias tablas en la base de datos.

En Java, los DTO son simplemente clases conocidas como POJOs (Plain Old Java Objects), las cuales contienen atributos y métodos básicos (como getters, setters, equals y hashcode), pero carecen de lógica de negocio. Por convención, las clases DTO se agrupan en un paquete específico y suelen tener el sufijo "DTO" en su nombre.

¿Por qué utilizar DTO?

Implementar DTOs proporciona una serie de beneficios clave:

- **Seguridad**: Evitan la exposición de datos sensibles en la URL, mejorando la seguridad al mantener estos datos fuera de la vista.
- **Optimización de solicitudes**: Permiten enviar y recibir datos de manera estructurada, reduciendo el número de solicitudes necesarias y mejorando la eficiencia de la comunicación entre cliente y servidor.
- **Validación de datos**: Facilitan la validación de datos en el lado del servidor, asegurando que los datos recibidos sean correctos antes de procesarlos.

- **Modularidad y escalabilidad**: Separan la lógica de negocio de la capa de presentación, lo que mejora la modularidad y facilita la evolución del sistema.
- **Flexibilidad**: Permiten adaptar la estructura de los datos según las necesidades específicas de cada solicitud o respuesta, brindando mayor flexibilidad en el diseño de la API.

¿Cómo implementamos los DTO en nuestra API?

Puedes implementar DTOs en cualquier método que requiera transferir datos, lo que permite obtener o manipular únicamente la información necesaria. Por ejemplo, si tienes un método en tu API que solo necesita ciertos datos de una entidad, puedes crear un DTO con esos datos específicos.

Recomendaciones para Implementar Clases DTO

- **Organizar las clases en un paquete específico**: Crea un paquete dedicado a las clases DTO dentro del proyecto. Esto facilita su localización y mantenimiento.
- **Nombrar las clases de manera significativa**: Asigna nombres descriptivos a los DTOs para reflejar claramente su propósito y tipo de datos.
- **Incluir solo los atributos necesarios**: Declara solo los atributos necesarios para el propósito del DTO, evitando sobrecargarlo con datos innecesarios.
- **Utilizar datos de referencia**: En lugar de almacenar objetos completos, usa identificadores (IDs) para representar relaciones, reduciendo la complejidad y evitando problemas con la serialización.

Ejemplo: Implementación del patrón DTO para la creación de un libro

Entidad LIBRO: La entidad `Libro` es la representación del objeto en la base de datos. Se configura con anotaciones JPA, como `@Entity`, `@Table`, `@Id`, `@ManyToOne`, etc., para definir cómo se mapearán las columnas de la tabla en la base de datos con los atributos de la clase Java. Esta entidad es responsable de realizar las operaciones CRUD directamente en la base de datos.

```
package com.egg.libreriaapi.entidades;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
```

```java
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@Entity
@Table(name = "libro")
public class Libro {

    @Id
    @Column(name = "id_libro")
    private Long idLibro;

    @Column(name = "titulo")
    private String titulo;

    @Column(name = "ejemplares")
    private Integer ejemplares;

    @Column(name = "libro_activo")
    private boolean libroActivo;

    @ManyToOne
    @JoinColumn(name = "id_autor")
    private Autor autor;

    @ManyToOne
    @JoinColumn(name = "id_editorial")
    private Editorial editorial;

}
```

.

Clase LibroCreateDTO: El `DTO` es un objeto que se utiliza para transferir solo los datos necesarios entre las capas de la aplicación. En este caso, `LibroCreateDTO` contiene únicamente los atributos relevantes para crear un nuevo libro. A diferencia de la entidad `Libro`, que tiene relaciones completas con `Autor` y `Editorial`, el DTO sólo contiene los identificadores (`idAutor` e `idEditorial`) para estos objetos.

```java
package com.egg.libreriaapi.modelos;

import lombok.Data;

@Data
public class LibroCreateDTO {
```

```
    private Long isbn;//Este dato será utilizado como idLibro
    private String titulo;
    private Integer ejemplares;
    private String idAutor;
    private String idEditorial;
    private boolean libroActivo;

}
```

En el servicio, el método `crearLibro` recibe el `DTO` como parámetro, lo convierte en una entidad `Libro`, y luego lo guarda en la base de datos. El servicio utiliza los datos del DTO para establecer los atributos correspondientes en la entidad `Libro` y luego resuelve las relaciones (`Autor` y `Editorial`) utilizando los identificadores pasados en el DTO.

```
package com.egg.libreriaapi.servicios;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.egg.libreriaapi.entidades.Autor;
import com.egg.libreriaapi.entidades.Editorial;
import com.egg.libreriaapi.entidades.Libro;
import com.egg.libreriaapi.modelos.LibroCreateDTO;
import com.egg.libreriaapi.modelos.LibroListarActivosDTO;
import com.egg.libreriaapi.repositorios.LibroRepositorio;

@Service
public class LibroServicio {
    @Autowired
    private LibroRepositorio libroRepositorio;

    @Autowired
    private AutorServicio autorServicio;

    @Autowired
    private EditorialServicio editorialServicio;

    @Transactional
    public void crearLibro(LibroCreateDTO libroCreateDTO) {
        Libro libroNvo = new Libro();
        libroNvo.setIdLibro(libroCreateDTO.getIsbn());
        libroNvo.setTitulo(libroCreateDTO.getTitulo());
        libroNvo.setEjemplares(libroCreateDTO.getEjemplares());
       libroNvo.setLibroActivo(libroCreateDTO.isLibroActivo());
        Autor autor = autorServicio.getOne(libroCreateDTO.getIdAutor());
        if (autor != null) {
            libroNvo.setAutor(autor);
        }
```

```
        Editorial editorial =
editorialServicio.getOne(libroCreateDTO.getIdEditorial());
        if (editorial != null) {
            libroNvo.setEditorial(editorial);
        }
        libroRepositorio.save(libroNvo);
    }

}
```

El controlador maneja las solicitudes HTTP. Recibe el `DTO` a través del cuerpo de la solicitud (`@RequestBody`), lo pasa al servicio para crear el libro, y luego devuelve una respuesta HTTP. Si todo va bien, devuelve un código `200 OK(en este caso)`; si ocurre un error, devuelve un mensaje de error.

```
 @PostMapping("/crear")
    public ResponseEntity<Object> crearLibro(@RequestBody LibroCreateDTO
libroDTO) {
        try {
            libroServicio.crearLibro(libroDTO);
            return new ResponseEntity<>(HttpStatus.OK);
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                    .body("{\"Algun dato no es correcto o es nulo,
revisar.\"}");
        }
    }
```

Uso de DTO en Consultas del Repositorio

Los DTO también se pueden utilizar para personalizar las consultas en la base de datos y evitar cargar entidades completas. En el caso del repositorio `LibroRepositorio`, se utiliza un `@Query` para recuperar sólo los datos necesarios (como el título y el número de ejemplares) en lugar de toda la entidad `Libro`. Esto mejora el rendimiento de las consultas.

```
package com.egg.libreriaapi.repositorios;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import com.egg.libreriaapi.entidades.Libro;
import com.egg.libreriaapi.modelos.LibroListarActivosDTO;
```

```java
@Repository
public interface LibroRepositorio extends JpaRepository<Libro, Long> {

   @Query("SELECT new
com.egg.libreriaapi.modelos.LibroListarActivosDTO(l.titulo, l.ejemplares) " +
           "FROM Libro l WHERE l.libroActivo = true")
    List<LibroListarActivosDTO> encontrarActivos();
}
```

En este ejemplo, la consulta crea una instancia del DTO `LibroListarActivosDTO` directamente desde la base de datos, recuperando solo los datos necesarios (`titulo` y `ejemplares`), sin necesidad de cargar la entidad `Libro completa`.

¡Explorar el mundo de la construcción de API con Spring es realmente fascinante! Existen numerosas prácticas recomendadas que pueden elevar tus habilidades de desarrollo al siguiente nivel. Te invitamos a adentrarte en conceptos como @MappedSuperclass, la implementación de código genérico y el uso de mappers. Estas herramientas no solo mejorarán la claridad de tu código, sino que también fomentarán la reutilización y contribuirán significativamente a la estructura general de tu proyecto. ¡Sumérgete en estas áreas para perfeccionar tus habilidades y llevar tus proyectos a nuevas alturas!

CONVENCIONES SOBRE LAS URLS EN APLICACIONES REST

Las URLs son fundamentales en las aplicaciones REST, ya que permiten identificar y acceder a los recursos de manera eficiente. A continuación, se detallan las mejores prácticas para estructurar las URLs de tu API:

Aspecto	Lo que hay que buscar	Lo que se debe hacer	Lo que no se debe hacer	Ejemplo
Significancia y descripción de las URLs	URLs significativas y descriptivas	Mantener las URLs claras y comprensibles	Evitar URLs ambiguas o poco descriptivas	/users (para representar una colección de usuarios)

Pluralidad de sustantivos en los recursos	Sustantivos en plural para los recursos	Utilizar nombres de recursos en plural	Evitar nombres de recursos en singular	/users (en lugar de /user para representar una colección de usuarios)
Rutas anidadas para relaciones	Relaciones anidadas en rutas	Representar relaciones de manera clara	Evitar rutas demasiado complejas	/users/123/orders (para obtener los pedidos del usuario con el ID 123)
Estructura jerárquica	Estructura jerárquica en las URLs	Mantener una jerarquía lógica	Evitar jerarquías profundas	/departments/123/employees (para representar empleados vinculados a departamentos)
Separación de palabras en las URLs	Guiones bajos o guiones para separar palabras	Utilizar guiones para mayor legibilidad	Evitar URLs con nombres de recursos sin separación	/product-reviews (para acceder a revisiones de productos)
Longitudes excesivas de URL	URLs de longitud razonable	Mantener las URLs cortas y manejables	Evitar URLs excesivamente largas	/users/123/orders/321/products/456/reviews (evitar URLs muy largas)
Uso de verbos en las URLs	Ausencia de verbos en las URLs	Utilizar verbos para acciones específicas	Evitar URLs con acciones en lugar de recursos	/users/123 (para obtener un usuario)
Parámetros complejos en la URL	Parámetros de consulta en la URL	Utilizar parámetros de consulta de forma clara	Evitar rutas con demasiados parámetros complicados	/products?category=beauty&sort=price,desc (para filtrar productos por categoría y ordenarlos por precio)
Sufijos de archivo en las URLs	Ausencia de sufijos de archivo	Utilizar representaciones de recursos	Evitar extensiones de archivo en las URLs	/products (en lugar de /products.json)
Uso de mayúsculas en las URLs	Utilización de letras minúsculas	Mantener consistencia en el uso de minúsculas	Evitar mayúsculas en las URLs	/products (en lugar de /Products)
Anidamientos o jerarquías de más de dos niveles	Evitar anidamientos excesivos	Mantener una profundidad de anidamiento adecuada	Evitar rutas con múltiples niveles de anidamiento	/users/123/orders/321/products/456/reviews (evitar anidar demasiado las rutas)