Ejercicio 15 – Uso de DTO y proyecciones

En ejercicios anteriores hemos definido las clases entidad de nuestro modelo con todos sus atributos y las relaciones entre ellas. Además, hemos utilizado estas clases para cualquier operación que requiera la presencia de un objeto determinado.

En aplicaciones que requieren de intercambiar mensajes con terceros, una de las problemáticas más comunes es utilizar las clases de entidades en la capa de aplicación, lo que ocasiona que retornemos más datos de los necesarios o expongamos detalles poco convenientes de la base de datos.

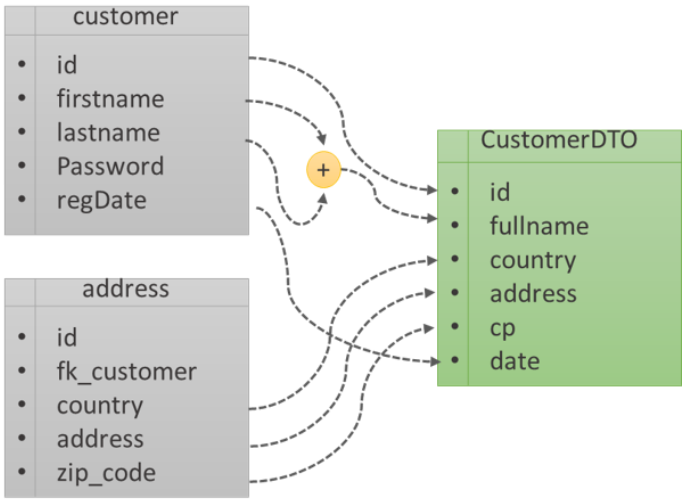
¿Qué es un DTO?

Un ***DTO*** (Data Transfer Object) es un patrón de arquitectura propuesto por Martin Fowler en su libro *Patterns of Enterprise Application Architecture*, cuya finalidad es crear objetos que ofrezcan una versión "resumida" de las entidades según las necesidades de las entradas y salidas de las operaciones.

Entre los beneficios de utilizar el patrón DTO encontramos:

- Reducir la cantidad de información que transferimos entre las capas.
- Adaptar los datos a una salida específica.
- Desacoplar los modelos de dominio de la capa de presentación.

Ejemplo de DTO



En la imagen vemos como dos entidades que mapean dos tablas de la base de datos, la tabla Customer y la tabla Address. La tabla Address tiene una columna que hace referencia al Customer, formando una relación One To One.

Podemos ver que hemos creado un nuevo objeto llamado CustomerDTO, en el cual podemos agregar libremente cuantos atributo requiramos, incluso, podemos asignarle valores de diferentes fuentes de datos.

Debido a que el DTO es una clase creada únicamente para una determinada respuesta, es posible modificarla sin mucho problema, pues no tiene un impacto en la capa de servicios o de datos, ya que en estas capas se trabaja con las Entidades.

Características de un DTO

Si bien un DTO es simplemente un objeto plano, sí que tiene que cumplir algunas reglas para poder considerar que hemos creado un DTO correctamente implementado:

- ***Solo lectura***: Dado que el objetivo de un DTO es utilizarlo como un objeto de transferencia entre el cliente y el servidor, solo deberemos de tener *getters* y *setters* de los atributos del DTO.
- ***Serializable***: Si los objetos tienen que viajar por la red, el propio objeto y sus atributos deben ser serializables.

En este contexto, es interesante el uso de los ***Records***, un tipo especial de clase que actúa como un objeto de datos y que es inmutable.

Mejora del rendimiento de consultas y proyecciones

En prácticas anteriores, hemos trabajado con entidades y sus asociaciones a la hora de hacer consultas a la base de datos. Esto es ineficiente en operaciones de lectura, donde tiene sentido únicamente obtener de la base de datos la información imprescindible.

En este contexto, surge el concepto de ***proyección***. Una proyección es una forma de especificar qué campos de una entidad (o diversas entidades) deseamos recuperar de la base de datos.

Para ilustrar el concepto de proyección y ver cómo implementarlo en Spring, vamos a tomar como referencia el siguiente código de ejemplo:

```
public class Customer {
    @Id Long id;
    String firstname, lastname;
    String password;
    LocalDate regDate;
    Address address;
}

public class Address {
    @Id Long id;
    String country, address, zipcode;
}

public interface CustomerRepository extends Repository<Customer, Long> {
    List<Customer> findByLastname(String lastname);
}
```

Vamos a imaginar que solo queremos consultar los atributos de nombre y apellidos del Customer. Para ello, existen diversos tipos de proyecciones.

Proyección a través de clases (DTOs)

Una forma de definir proyecciones es mediante el uso de DTO que contengan los atributos a recuperar. El siguiente ejemplo muestra un DTO proyectado:

```
record CustomerNameDTO(String firstname, String lastname) {
}
```

Podemos utilizar fácilmente esta clase de tipo record como proyección en consultas derivadas y consultas JPQL personalizadas.

Uno de los problemas de la proyección con DTOs es la dificultad para realizar proyección anidadas.

Proyección por interfaz

Con las proyecciones basadas en interfaz, creamos una interfaz que define los campos que deseamos recuperar y Spring Data genera una implementación de esa interfaz en tiempo de ejecución.

```
public interface CustomerName {
  String getFirstname();
  String getLastname();
}
```

En este tipo de proyecciones, el nombre de los métodos debe coincidir con los nombres de los atributos de la entidad raíz del repositorio. Al hacerlo, se puede agregar un método de consulta de la siguiente manera:

```
public interface CustomerRepository extends Repository<Customer, Long> {
    List<CustomerName> findByLastname(String lastname);
}
```

Con interfaces, las proyecciones se pueden utilizar de forma anidada. Si también queremos incluir parte de la información de la dirección, podemos crear una interfaz de proyección para ello y devolver esa interfaz desde la declaración de getAddress(), como se muestra en el siguiente ejemplo:

```
public interface CustomerName {
    String getFirstname();
    String getLastname();
    AddressCountry getAddress();
}

public interface AddressCountry {
    String getCountry();
}
```

Utilizando este tipo de proyecciones Spring puede optimizar la ejecución de la consulta porque conoce todos los atributos para crear el proxy de la interfaz.

En el caso de requerir cálculos de nuevos valores, se pueden utilizar las llamadas proyecciones abiertas, utilizando la anotación ***@Value***.

```
public interface CustomerName {
  @Value("#{target.firstname + ' ' + target.lastname}")
  String getFullName();
  …
}
```

La entidad raíz está disponible en la variable "target". Una interfaz de proyección que utiliza @Value es una proyección abierta. Spring Data no puede aplicar optimizaciones en la consulta, porque la expresión utilizada en @Value (que utiliza SpEL, o Spring Expression Language) podría usar cualquier atributo de la entidad raíz.

Las expresiones utilizadas en @Value deben ser expresiones muy simples. Una alternativa a esta anotación podría ser los métodos default (introducidos en Java 8), como se muestra en el siguiente ejemplo:

```
public interface CustomerName {
    String getFirstname();
    String getLastname();

    default String getFullName() {
        return getFirstname().concat(" ").concat(getLastname());
    }
}
```

Este enfoque requiere implementar lógica basada exclusivamente en los otros métodos de acceso expuestos en la interfaz de proyección.

Una segunda alternativa, más flexible, es implementar la lógica en un bean de Spring y luego invocarla desde la expresión SpEL:

```
@Component
public class CustomerUtil {
    String getFullName(Customer customer) {
        return customer.getFirstname().concat("
").concat(customer.getLastname());
    }
}

public interface CustomerName {
    @Value("#{@customerUtil.getFullName(target)}")
    String getFullName();
    …
}
```