

Mapeos

Hay algunas tecnologías que tienen un comportamiento que, en ocasiones, cambia incluso la forma en la que programas. Sigue siendo Java, pero ya no se hace de la misma manera. Ocurre con muchos Frameworks. Y no queda otra que aprenderse esas peculiaridades.

Cuando trabajamos con **Spring**, **REST** y **ORM**, (y tecnologías similares), pasa esto. Digamos que tiene una regla de oro:

“Las Entidades de BBDD no deberían ser las mismas que usas para enviarlas por HTTP por el REST”

Esto quiere decir, que si tú tienes una tabla de cuentas (T_ACCOUNTS), y usas Hibernate para leer las cuentas de un cliente, harás uso de estas dos clases:

- **AccountDao**: Una clase con los métodos de acceso a BBDD.
- **EntityAccount**: Que define cómo se va a hacer el mapeo con la tabla.

Supongamos que ahora tenemos que enviar esas cuentas por un REST. Lo que hacemos entonces es usar la clase:

- **AccountResource**: Contiene los endpoints de la Api.

En esta clase definimos un método anotado con @GET que devuelva esa lista de cuentas y un HTTP 200. Y ya estaría. ¿No?

PUES NO

Lo que debería hacerse es una clase **Account** idéntica a **EntityAccount**, hacer una conversión (mapeo) entre ambas, y enviar por el REST la lista de **Account**.

Lo que NO se debería hacer es usar el **EntityAccount** y meterlo en un REST.

No es buena idea que un objeto del Modelo llegue hasta la Vista.

Esto es así por cómo funciona JTA / Hibernate. No es buena idea mezclar cosas.

Cómo hacer este mapeo fácil

A mano. Es una opción.

O podemos utilizar **MapStruct** para hacer una conversión de objetos de un tipo a otro de forma automática en tiempo de compilación. Solamente hay que hacer una clase dedicada a estos mapeos, anotarla si hay casos especiales, y después utilizarla.

Para añadir **MapStruct**, primero ponemos las dependencias en el pom.xml:

El plugin, dentro de build/plugins:

```
<!-- MapStruct annotation processor -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>21</source>
        <target>21</target>
        <annotationProcessorPaths>
            <path>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>${mapstruct.version}</version>
            </path>
        </annotationProcessorPaths>
    </configuration>
</plugin>
```

Y después la dependencia:

```
<!-- MapStruct runtime -->
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>${mapstruct.version}</version>
</dependency>
```

Construye de nuevo el proyecto, y ya estamos listos para usar el **MapStruct**.

Para este ejemplo, supongamos que tenemos **AccountDao** y **Account**. La primera sirve para consultas en el ORM, y la segunda vamos a usarla para enviar por la Api Rest. Nótese que en este caso ambas clases son 100% iguales.

A modo de ejemplo, así es **Account**:

```
public class Account extends ResourceAbstract implements Serializable {  
  
    private static final long serialVersionUID = -8691149171142425942L;  
  
    private Long id = null;  
    private String login = null;  
    private String pass = null;  
  
    private UserType userType;  
  
    enum UserType {  
        TEACHER, STUDENT  
    }  
  
    private User entityUser = null;
```

Para poder convertir de forma automática un objeto en otro con **MapStruct**, hacemos una interface en el Controlador llamada **AccountMapper**.

```
@Mapper(componentModel = "spring")  
public interface AccountMapper {  
  
    // Map EntityAccount → Account  
    public Account toApi(EntityAccount entity);  
  
    // Map Account → EntityAccount  
    public EntityAccount toEntity(Account api);  
}
```

Definimos en ella dos métodos, que podemos definir como queramos realmente.

- **toApi ()**: convierte un EntityAccount en un Account
- **toEntity ()**: convierte un Account en un EntityAccount

Y ya podríamos usar tranquilamente los métodos. Por ejemplo, este código se encuentra en el Controlador:

```
if (doCheck(entityAccount, login, decryptedPass)) {  
    ret = accountMapper.toApi(entityAccount);  
    log.debug("Login - Ok!");  
} else {
```

Qué ocurre realmente

Si te limitas a hacer esto, en cuanto el Spring ejecute la línea `toApi()` ya tendrá todo el código necesario para hacer la conversión de `EntityAccount` en `Account`. No tienes que escribirla tú.

Pero ten cuidado, porque, por defecto **MapStruct**:

- Mapea automáticamente campos por nombre y tipo (tienen que coincidir).
- Para campos **primitivos o tipos simples** (Long, String, enums), asigna los valores sin problema.
- Para campos **objetos complejos**, es decir, todo lo demás, intenta:
 - Si hay un **mapper** definido para ese tipo, lo usa.
 - Si **no hay mapper, no hace nada**. Lo pone a null.

Por tanto, en el ejemplo:

```
public class Account extends ResourceAbstract implements Serializable {  
  
    private static final long serialVersionUID = -8691149171142425942L;  
  
    private Long id = null;  
    private String login = null;  
    private String pass = null;  
  
    private UserType userType;  
  
    enum UserType {  
        TEACHER, STUDENT  
    }  
  
    private User entityUser = null;
```

Los campos `id`, `login`, `pass`, `userType` los mapea sin problemas.

El campo `entityUser` lo mapea a null.

¿Y si quiero hacer cosas raras?

Cuando hablamos de mapear un objeto en otro, o un JSON a un objeto, etc. siempre se da por sentado que vas a mapearlo todo. Por defecto. Pero ten en cuenta que no siempre es así.

En realidad, tú puedes mapear lo que quieras como quieras. Por ejemplo, el valor de una clave externa no se lleva nunca a un objeto en JDBC o ORM. Puede ser que cuando lees la tabla Alumno no te interese nunca la foto. En ese caso, no la vas a usar, tu EntityAlumno no tendrá ese campo. Y puede ser que tengas varios objetos Alumno distintos para mapear esa tabla.

No es tan infrecuente como parece esta cosa.

Por otro lado, **MapStruct** te permite 'jugar' con los mapeos añadiendo anotaciones. Veamos algún ejemplo:

@Mapping nos permite mapear un campo en otro campo. Útil cuando tienes dos objetos diferentes

```
@Mapper(componentModel = "spring")
public interface AccountMapper2 {

    @Mapping(source = "pass", target = "password")
    @Mapping(source = "userType", target = "role")
    public Account toApi(EntityAccount entity);

    @Mapping(source = "password", target = "pass")
    @Mapping(source = "role", target = "userType")
    public EntityAccount toEntity(Account api);
}
```

También podemos añadir valores por defecto si añadimos la variable **constant**.

@Mapper nos permite indicar otro mapper para hacer el mapeo de los objetos complejos, como sucede con el campo entityUser.

```
@Mapper(componentModel = "spring", uses = UserMapper.class)
public interface AccountMapper2 {

    public Account toApi(EntityAccount entity);

    public EntityAccount toEntity(Account api);
}
```

También puedes mapear listas, ignorar campos, calcular y asignar valores, convertir tipos, etc.