

Tutorial - Parte 2

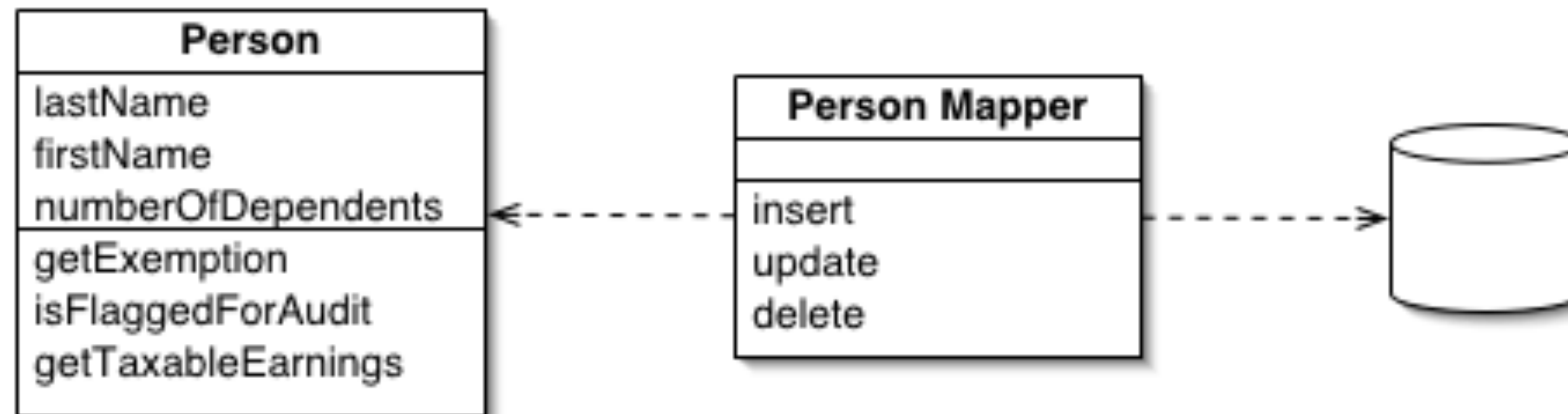
Spring

Resumen

- Completada la capa persistencia para nuestro microservicio
 - Mediante JPA hemos definido el mapeo que deseamos entre nuestras clases y las tablas de la base de datos
 - Usando SpringData hemos configurado las clases que deseamos que Spring fabrique e inyecte a nuestro proyecto para poder persistir objetos.

Data Mapper

- Patrón de diseño que crea una capa de abstracción entre los objetos del dominio y otra capa (generalmente la de persistencia). Su objetivo principal es proveer una separación de responsabilidades, permitiendo que la lógica de negocio (objetos del dominio) se mantengan completamente indiferentes a la forma en que es almacenada o recuperada la información.



Data Mapper

Problemas que resuelve

- Alto acoplamiento: Si no se usa el patrón Data Mapper, los objetos del dominio podrían contener sentencias SQL, lógica de conexión a la base de datos, o conocimiento respecto a la estructura de las tablas. Y todas estas cosas acoplan fuertemente la lógica de negocio al mecanismo de persistencia.
- Dificultad de pruebas: Cuando los objetos de dominio están atados a la base de datos, probar se hace más difícil ya que dependerá de la conexión a la base de datos.
- Flexibilidad limitada: Cuando existe el acoplamiento cambiar la base de datos (de MySQL a PostgreSQL o a una base NoSQL) requiere cambios en la capa de dominio.
- Lógica duplicada: El mapear datos de forma manual en varias partes de la aplicación puede generar código redundante.

Data Mapper

Elementos

1. **Objetos del dominio:**

- Representan conceptos del mundo real (Usuario, Producto, Orden, etc).
- Contienen datos (campos de propiedades) y lógica de negocio (métodos).
- Importante que NO contengan ninguna referencia al medio de persistencia. Deben ser POJOs (Plain old Java objects).

Data Mapper

Elementos

2. Data Mapper

- Objecto responsable de mapear datos entre el dominio y el almacén de datos.
- Contiene la lógica para las operaciones CRUD (Create, Read, Update, Delete).
- Lo que conoce es:
 - La estructura del objeto del dominio.
 - El esquema de la base de datos (nombres de tablas, columnas).
 - Como ejecutar queries.
 - Como convertir los resultados de la base de datos a objetos del dominio y viceversa.
- Típicamente tiene métodos como: `findById()`, `save()`, `update()`, `delete()`.

Data Mapper

Elementos

3. **Data Store:**

- La base de datos (MySQL, PostgreSQL, MongoDB, archivos planos, etc).

Data Mapper

Flujo - Recuperar información

1. La aplicación (por ejemplo la capa de servicio) solicita un objeto.
2. Para obtener su objeto ejecuta un método del UserMapper (por ejemplo `userMapper.findById(123)`).
3. El UserMapper:
 - A. Establece una conexión a la base de datos.
 - B. Ejecuta una sentencia SQL.
 - C. Toma los datos crudos del result set.
 - D. Construye un objeto del dominio y lo llena con los datos recuperados.
 - E. Retorna el objeto a la aplicación.

Data Mapper

Flujo - Persistir objetos

1. La aplicación tiene un objeto del dominio que necesita ser persistido.
2. Llama a un método del UserMapper (`userMapper.save(obj)`).
3. El UserMapper:
 - A. Extrae los datos del objeto del dominio.
 - B. Construye la sentencia SQL.
 - C. Ejecuta la sentencia SQL.
 - D. Maneja los detalles de la respuesta proporcionada por la base de datos

Data Mapper

Implementación

Herramientas



Java bean mappings, the easy way!

Data Mapper

Implementación

Como usamos Gradle sólo necesitamos añadir la dependencia:



The screenshot shows the MapStruct documentation website. The top navigation bar includes links for MapStruct, News, Documentation (which is highlighted), Community, Development, Sponsor, FAQ, and Code & Issues. On the left sidebar, under the DOCUMENTATION section, the 'Installation' link is highlighted. The main content area is titled 'Installation' and includes a link to 'Edit on GitHub'. Below this, there is a section for 'Distribution Bundle' and a 'Gradle' section. The Gradle section contains the text: 'When using a modern version of Gradle (>= 4.6), you add something along the following lines to your *build.gradle*:' followed by a code block. The code block shows a Gradle dependencies list with the following lines: 1. dependencies {, 2. ..., 3. implementation 'org.mapstruct:mapstruct:1.6.3', 4. annotationProcessor 'org.mapstruct:mapstruct-processor:1.6.3', 5. }, 6. }. A red oval is drawn around the implementation and annotationProcessor lines.

```
1. dependencies {
2.   ...
3.   implementation 'org.mapstruct:mapstruct:1.6.3'
4.   annotationProcessor 'org.mapstruct:mapstruct-processor:1.6.3'
5. }
6. }
```

Data Mapper

Implementación

Como usamos Gradle sólo necesitamos añadir la dependencia:



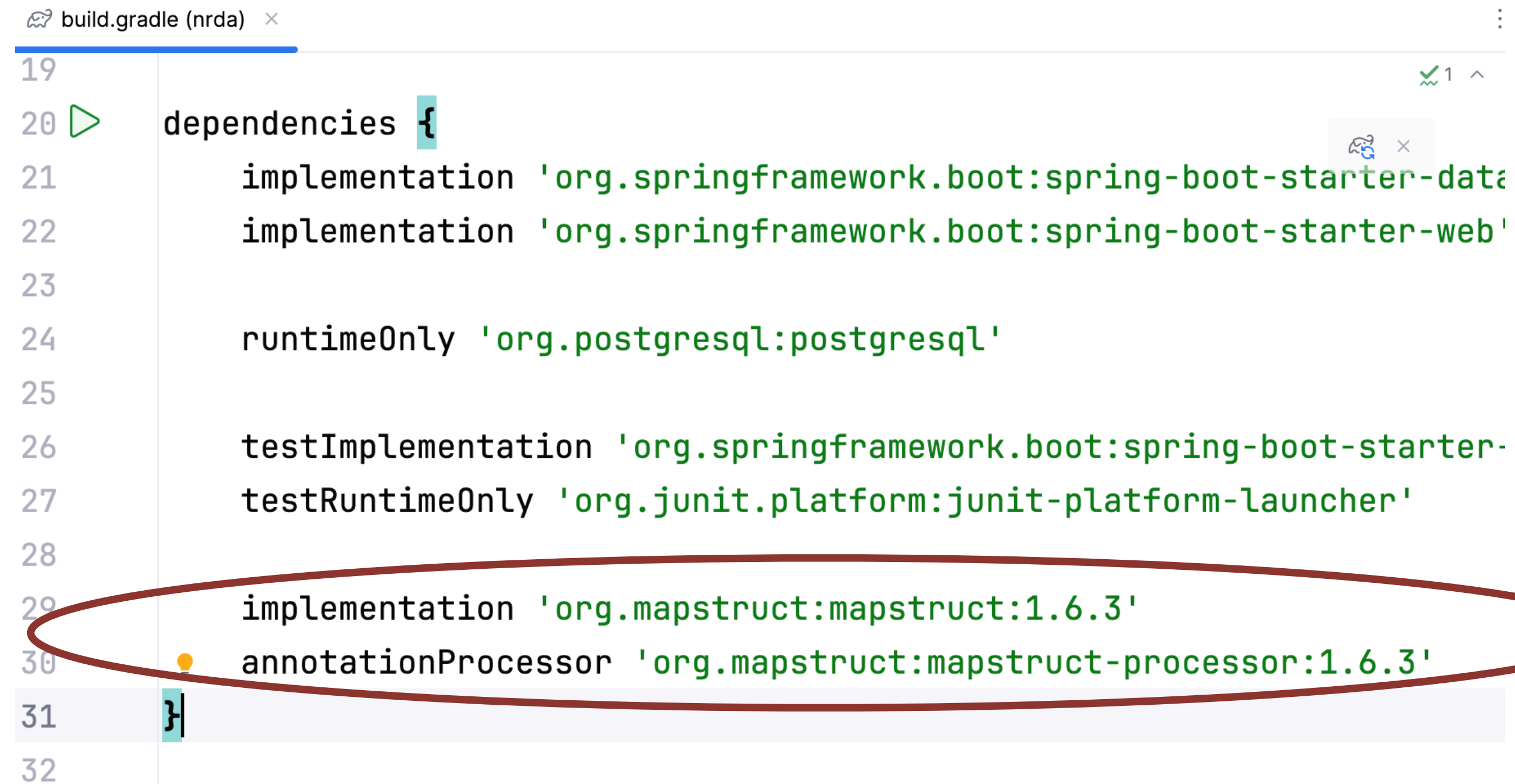
```
build.gradle (nrda) x
19
20 dependencies {
21     implementation 'org.springframework.boot:spring-boot-starter-data
22     implementation 'org.springframework.boot:spring-boot-starter-web'
23
24     runtimeOnly 'org.postgresql:postgresql'
25
26     testImplementation 'org.springframework.boot:spring-boot-starter-
27     testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
28
29
30 }
```

The screenshot shows a code editor window titled 'build.gradle (nrda)'. The code defines the 'dependencies' block for a Gradle project. It includes 'spring-boot-starter-data' and 'spring-boot-starter-web' as implementation dependencies, 'postgresql' as a runtimeOnly dependency, and 'spring-boot-starter-test' and 'junit-platform-launcher' as test dependencies. The code is formatted with green text on a light background. A light blue bar highlights the end of the dependencies block on line 29.

Data Mapper

Implementación

Como usamos Gradle sólo necesitamos añadir la dependencia:

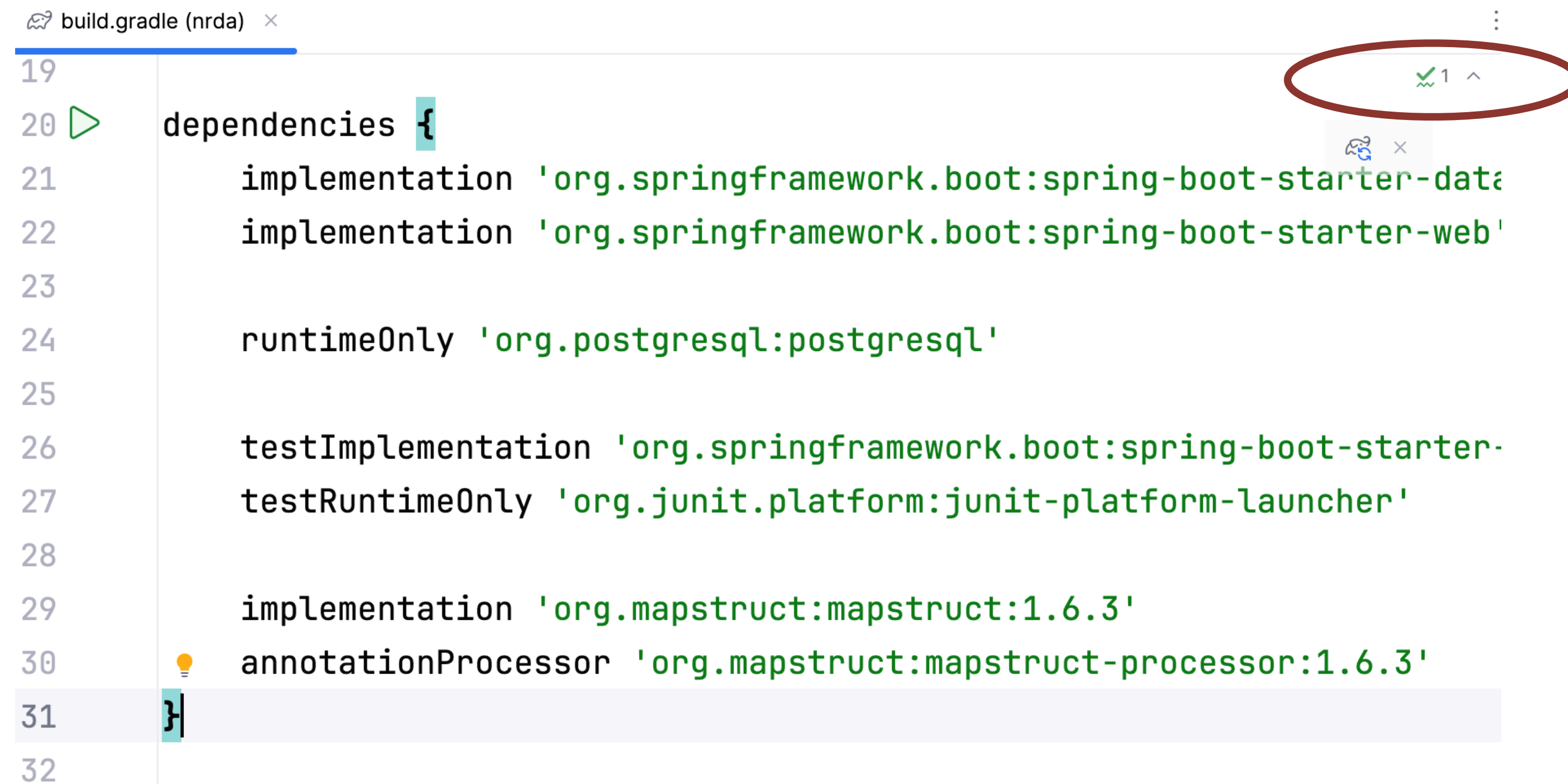


```
build.gradle (nrda) x
19
20 dependencies {
21     implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
22     implementation 'org.springframework.boot:spring-boot-starter-web'
23
24     runtimeOnly 'org.postgresql:postgresql'
25
26     testImplementation 'org.springframework.boot:spring-boot-starter-test'
27     testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
28
29     implementation 'org.mapstruct:mapstruct:1.6.3'
30     annotationProcessor 'org.mapstruct:mapstruct-processor:1.6.3'
31 }
32
```

Data Mapper

Implementación

Como usamos Gradle sólo necesitamos añadir la dependencia:



```
build.gradle (nrda) x
19
20 dependencies {
21     implementation 'org.springframework.boot:spring-boot-starter-data
22     implementation 'org.springframework.boot:spring-boot-starter-web'
23
24     runtimeOnly 'org.postgresql:postgresql'
25
26     testImplementation 'org.springframework.boot:spring-boot-starter-
27     testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
28
29     implementation 'org.mapstruct:mapstruct:1.6.3'
30     annotationProcessor 'org.mapstruct:mapstruct-processor:1.6.3'
31 }
32
```

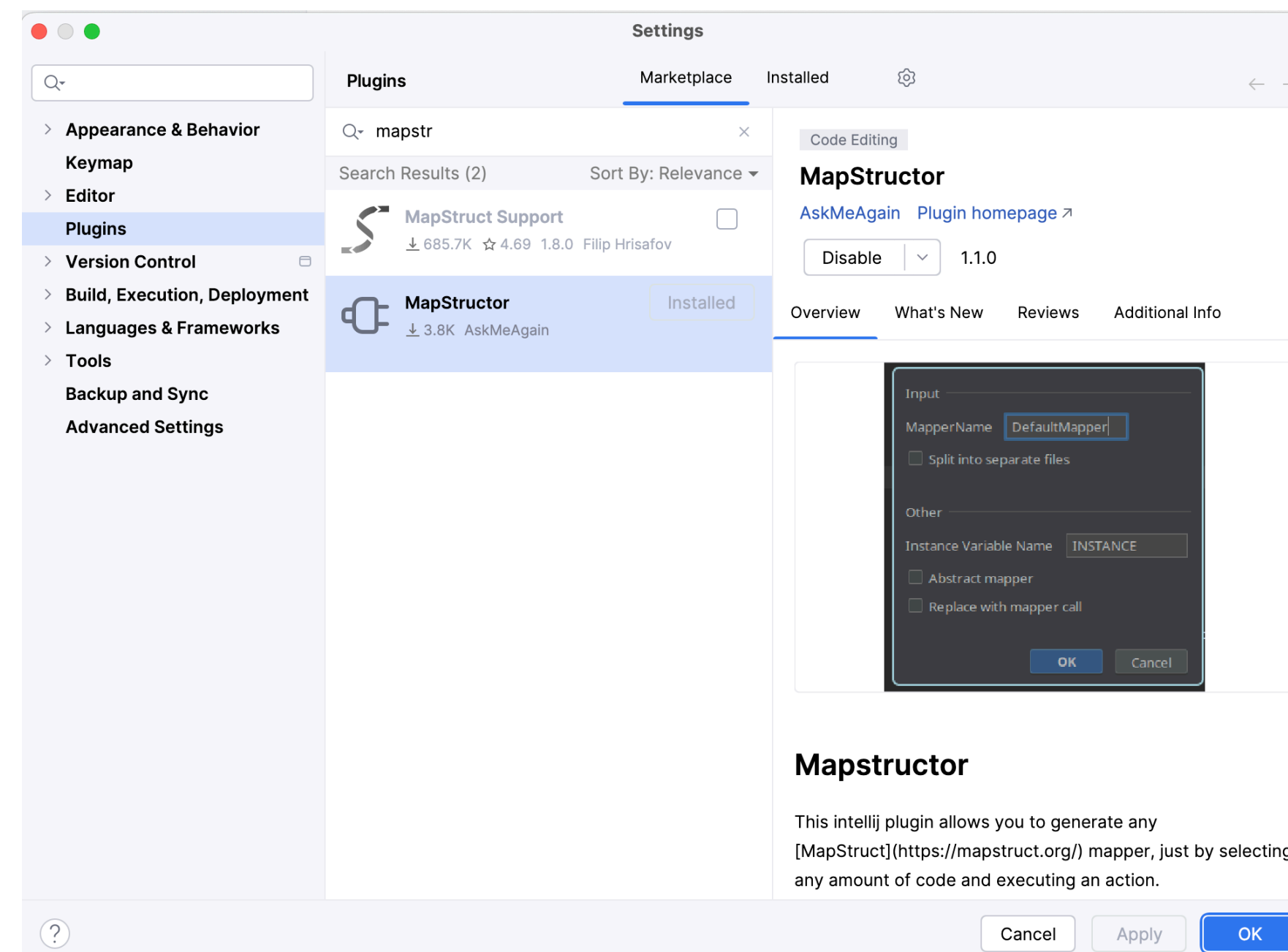
Actualizar cambios
en Gradle

Data Mapper

Implementación

Para que el IDE nos ayude conviene instalar el soporte de MapStruct para IntelliJ

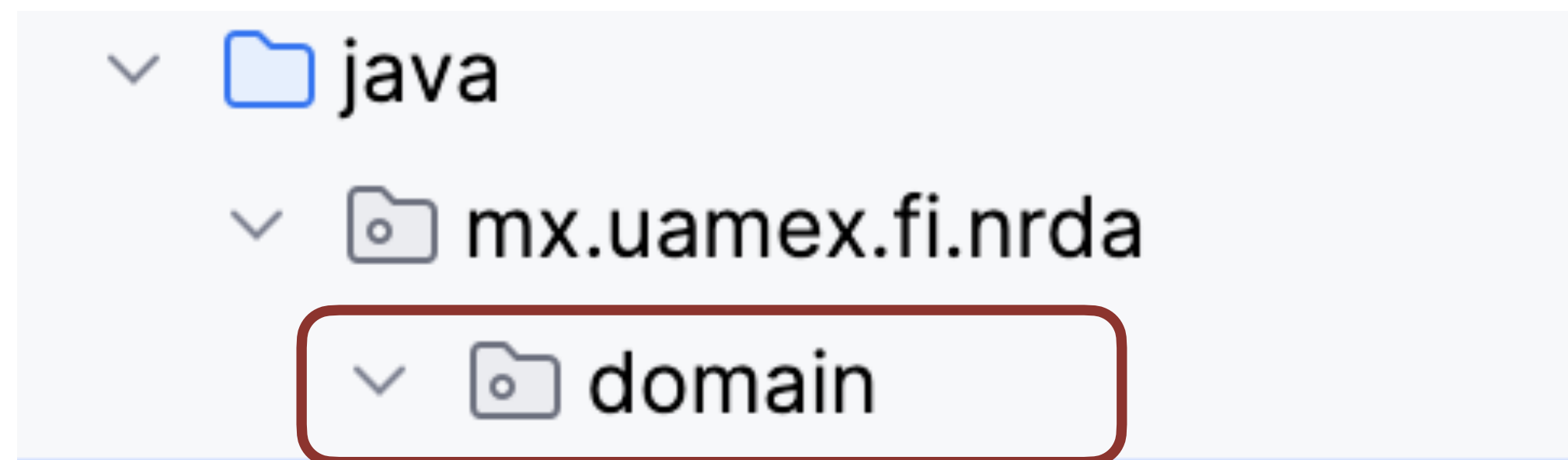
- Abrir IDE Settings



Data Mapper

Implementación

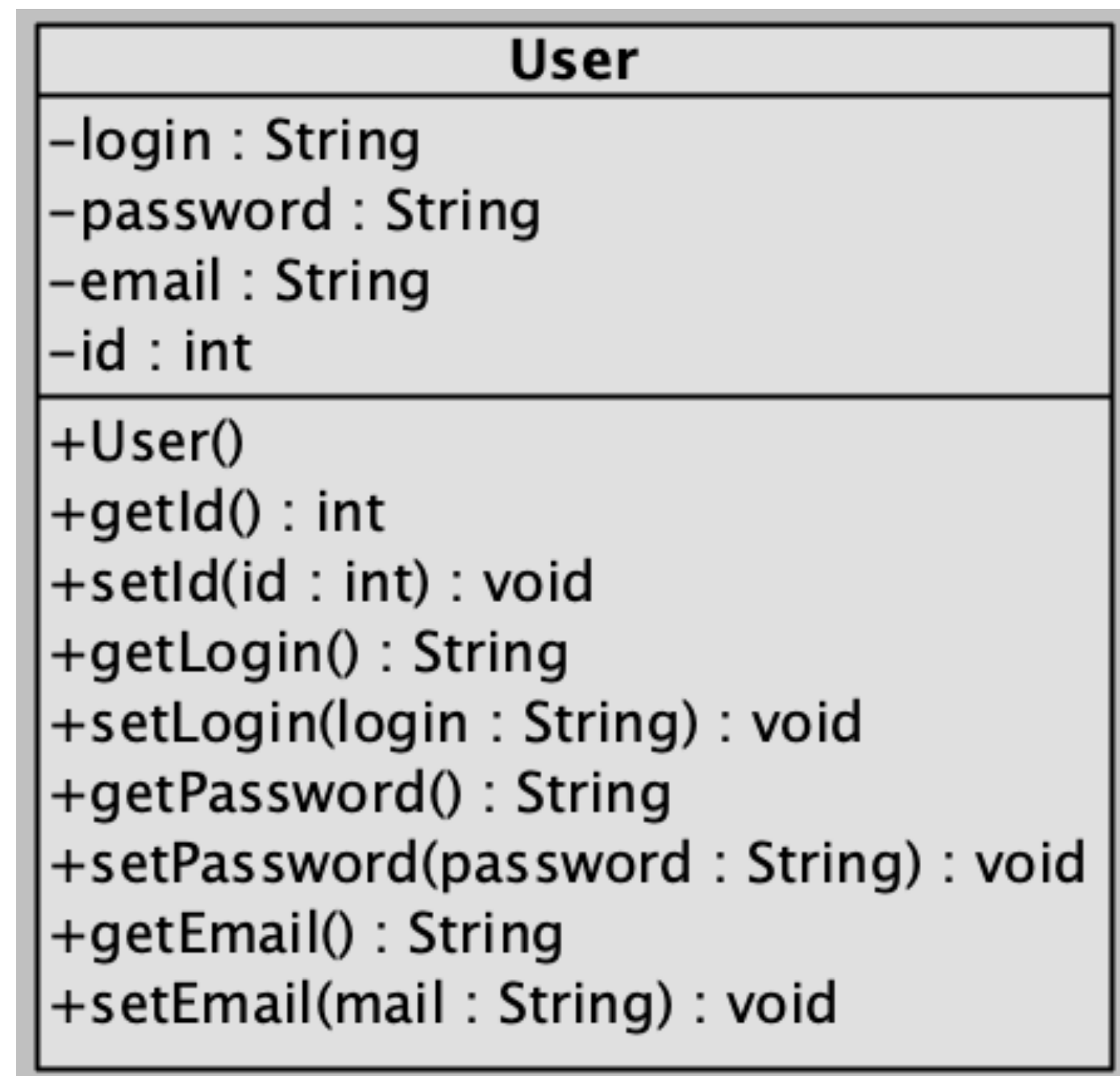
Definimos la(s) clase(s) del dominio



Data Mapper

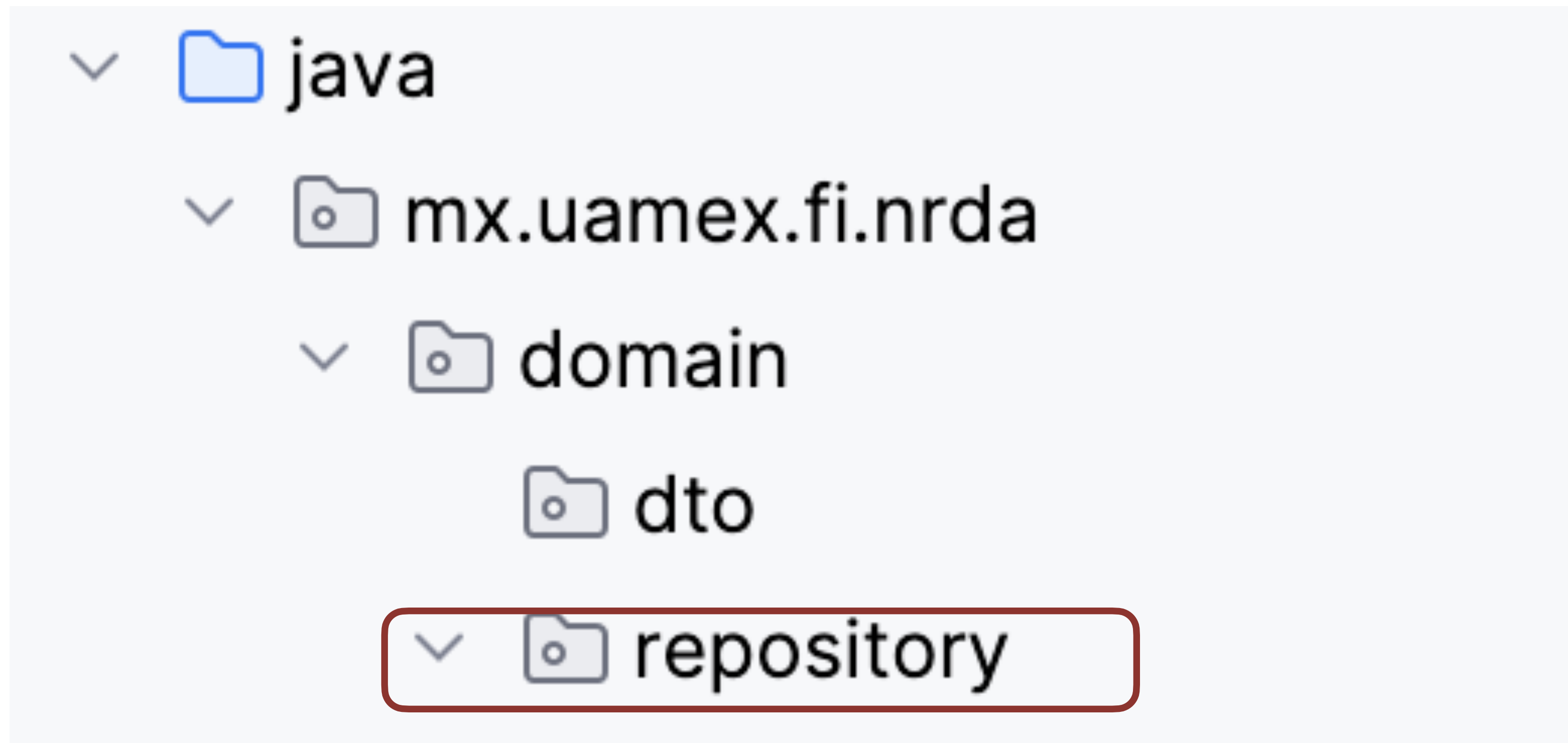
Implementación

Definimos la(s) clase(s) del dominio



```
1 package mx.uamex.fi.nrda.domain;
2
3 public class User { 2 usages
4     private int id; 2 usages
5     private String login; 2 usages
6     private String password; 2 usages
7     private String email; 2 usages
8
9     public int getId() { no usages
10         return id;
11     }
12
13     public void setId(int id) { no usages
14         this.id = id;
15     }
16 }
```

Creamos un repositorio



Creamos un repositorio

El repositorio será una interfaz que marcará las operaciones que deseamos realizar con los objetos del dominio

```
package mx.uamex.fi.nrda.domain.repository;

import mx.uamex.fi.nrda.domain.User;

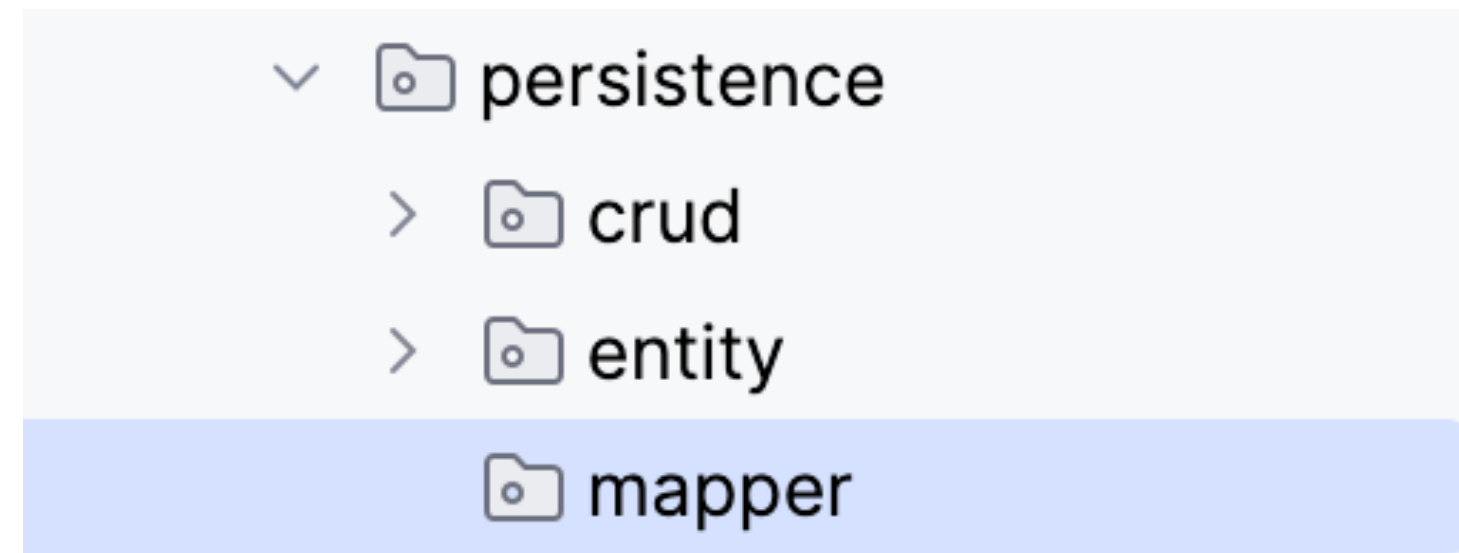
import java.util.List;
import java.util.Optional;

public interface UserRepository {
    List<User> getAll();
    Optional<User> getUser(String login);
    Optional<User> getUser(int id);
    void save(User user);
    boolean delete(int id);
}
```

Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (1): Creamos el paquete para los mapeadores



Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

Señalamos que es un mapeador



```
1 package mx.uamex.fi.nrda.persistence.mapper;
2
3 import mx.uamex.fi.nrda.domain.User;
4 import mx.uamex.fi.nrda.persistence.entity.Usuario;
5 import org.mapstruct.*;
6
7
8 @Mapper(componentModel = "spring")
9 public interface UserMapper {
10     @Mappings({
11         @Mapping(source = "id",target = "id"),
12         @Mapping(source = "login",target = "login"),
13         @Mapping(source = "contrasena",target = "password"),
14         @Mapping(source = "correo",target = "email"),
15     })
16     User toUser(Usuario u);
17     @InheritInverseConfiguration
18     Usuario toUsuario(User u);
19 }
20
```

⚠ 3 ✓ 2

Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

```
1 package mx.uamex.fi.nrda.persistence.mapper;
2
3 import mx.uamex.fi.nrda.domain.User;
4 import mx.uamex.fi.nrda.persistence.entity.Usuario;
5 import org.mapstruct.*;
6
7
8 @Mapper(componentModel = "spring")
9 public interface UserMapper {
10     @Mappings({
11         @Mapping(source = "id",target = "id"),
12         @Mapping(source = "login",target = "login"),
13         @Mapping(source = "contrasena",target = "password"),
14         @Mapping(source = "correo",target = "email"),
15     })
16     User toUser(Usuario u);
17     @InheritInverseConfiguration
18     Usuario toUsuario(User u);
19 }
20
```

⚠ 3 ✓ 2

→ Solicitamos la integración
MapStruct con Spring

Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

Nota: el nombre es importante, debe llamarse “to” seguido del nombre de la clase destino.

```
1 package mx.uamex.fi.nrda.persistence.mapper;
2
3 import mx.uamex.fi.nrda.domain.User;
4 import mx.uamex.fi.nrda.persistence.entity.Usuario;
5 import org.mapstruct.*;
6
7
8 @Mapper(componentModel = "spring")
9 public interface UserMapper {
10     @Mappings({
11         @Mapping(source = "id",target = "id"),
12         @Mapping(source = "login",target = "login"),
13         @Mapping(source = "contrasena",target = "password"),
14         @Mapping(source = "correo",target = "email"),
15     })
16
17     User toUser(Usuario u);
18     @InheritInverseConfiguration
19     Usuario toUsuario(User u);
20 }
```

→ Método mapeador

Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

Anotación que especifica
la forma de realizar el mapeo

```
1 package mx.uamex.fi.nrda.persistence.mapper;
2
3 import mx.uamex.fi.nrda.domain.User;
4 import mx.uamex.fi.nrda.persistence.entity.Usuario;
5 import org.mapstruct.*;
6
7
8 @Mapper(componentModel = "spring")
9 public interface UserMapper {
10     @Mappings({
11         @Mapping(source = "id",target = "id"),
12         @Mapping(source = "login",target = "login"),
13         @Mapping(source = "contrasena",target = "password"),
14         @Mapping(source = "correo",target = "email"),
15     })
16     User toUser(Usuario u);
17     @InheritInverseConfiguration
18     Usuario toUsuario(User u);
19 }
20
```

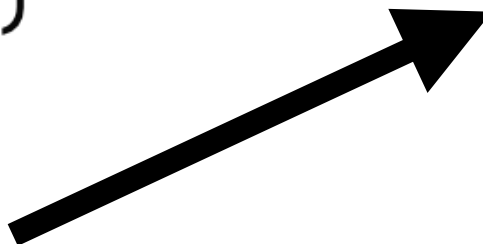
⚠ 3 ✓ 2

Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

```
@Mapper(componentModel = "spring")
public interface UserMapper {
    @Mappings({
        @Mapping(source = "id", target = "id"),
        @Mapping(source = "login", target = "login"),
        @Mapping(source = "contrasena", target = "password"),
        @Mapping(source = "correo", target = "email"),
    })
    User toUser(Usuario u);
    @InheritInverseConfiguration
    Usuario toUsuario(User u);
    List<User> toUsers(List<Usuario> usuarios);
}
```



Atributo en la clase origen

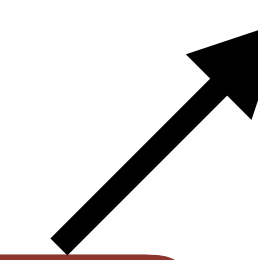
Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

```
@Mapper(componentModel = "spring")
public interface UserMapper {
    @Mappings({
        @Mapping(source = "id", target = "id"),
        @Mapping(source = "login", target = "login"),
        @Mapping(source = "contrasena", target = "password"),
        @Mapping(source = "correo", target = "email"),
    })
    User toUser(Usuario u);
    @InheritInverseConfiguration
    Usuario toUsuario(User u);
    List<User> toUsers(List<Usuario> usuarios);
}
```

Atributo en la clase destino



Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

```
@Mapper(componentModel = "spring")
public interface UserMapper {
    @Mappings({
        @Mapping(source = "id",target = "id"),
        @Mapping(source = "login",target = "login"),
        @Mapping(source = "contrasena",target = "password"),
        @Mapping(source = "correo",target = "email"),
    })
    User toUser(Usuario u);
    @InheritInverseConfiguration
    Usuario toUsuario(User u);
    List<User> toUsers(List<Usuario> usuarios);
}
```

→ Otro método apeador

Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

```
@Mapper(componentModel = "spring")
public interface UserMapper {
    @Mappings({
        @Mapping(source = "id",target = "id"),
        @Mapping(source = "login",target = "login"),
        @Mapping(source = "contrasena",target = "password"),
        @Mapping(source = "correo",target = "email"),
    })
    User toUser(Usuario u);
    @InheritInverseConfiguration
    Usuario toUsuario(User u);
    List<User> toUsers(List<Usuario> usuarios);
}
```

→ Aprovechamos la definición anterior especificando que simplemente esta es su inversa

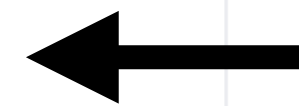
Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (2): Creamos las interfaces para los mapeadores

```
@Mapper(componentModel = "spring")
public interface UserMapper {
    @Mappings({
        @Mapping(source = "id",target = "id"),
        @Mapping(source = "login",target = "login"),
        @Mapping(source = "contrasena",target = "password"),
        @Mapping(source = "correo",target = "email"),
    })
    User toUser(Usuario u);
    @InheritInverseConfiguration
    Usuario toUsuario(User u);
    List<User> toUsers(List<Usuario> usuarios);
}
```

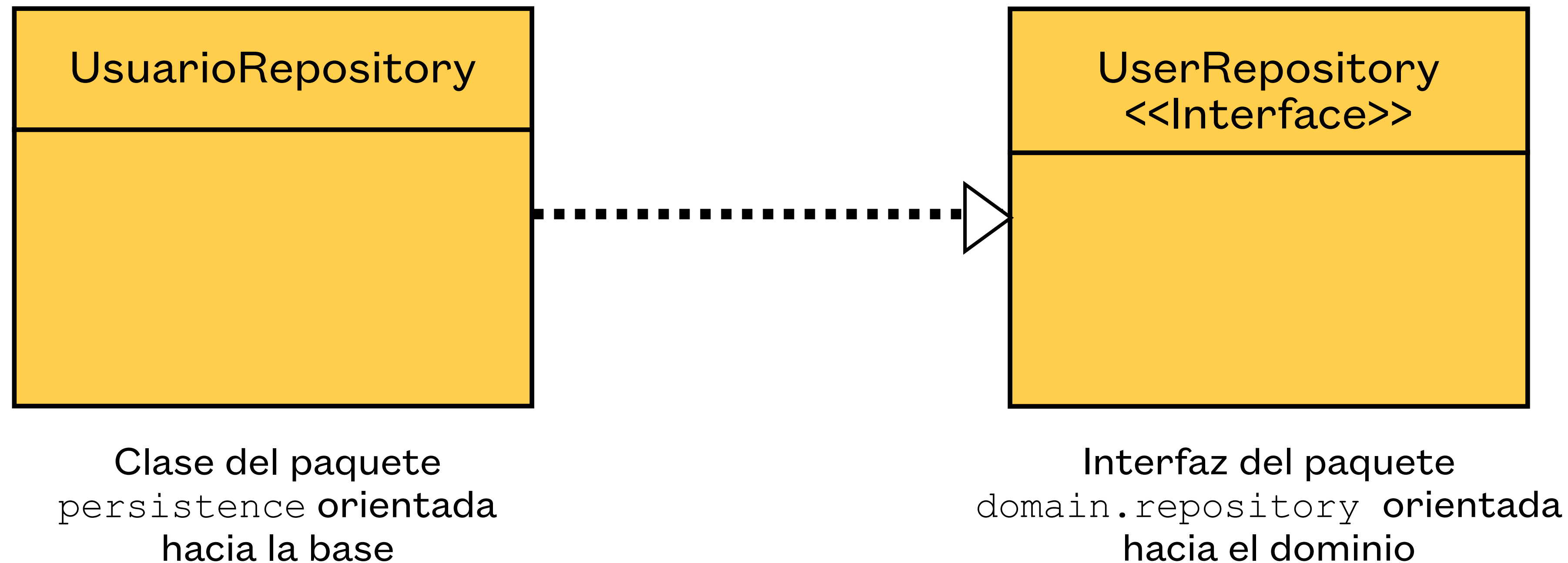
Mapeamos colecciones
aprovechando la definición
anterior



Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

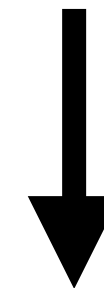


Creamos los mapeadores

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

```
public class UsuarioRepository implements UserRepository {
```



Implementamos el repositorio enfocado al dominio

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Para implementar la interfaz debemos incluir sus métodos a la clase, pero al hacerlo nos encontramos con un error en el método `getAll`, porque tanto la clase como la interfaz lo tienen pero difieren en valor de retorno

```
public List<Usuario> getAll(){ no usages
```



'getAll()' in 'mx.uamex.fi.nrda.persistence.UsuarioRepository' clashes with 'getAll()' in 'mx.uamex.fi.nrda.domain.repository.UserRepository'; incompatible return type

Make 'getAll()' return 'java.util.List<mx.uamex.fi.nrda.domain.User>'   More actions...  

 mx.uamex.fi.nrda.persistence.entity

@Entity

public class Usuario

 nrda.main 

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Para corregir el error, adaptaremos el método `getAll` para coincidir con el de la interfaz

Valor de retorno
de la interfaz




```
public class UsuarioRepository implements UserRepository {  
    private UsuarioCrudRepository repository; 6 usages  
    private UserMapper mapper; 1 usage  
  
    public List<User> getAll(){ no usages  
        List<Usuario> consultados;  
        consultados = (List<Usuario>) repository.findAll();  
        return mapper.toUsers(consultados);  
    }
```

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Para corregir el error, adaptaremos el método `getAll` para coincidir con el de la interfaz

Marcamos que el método sobrescribe el método abstracto de la interfaz

```
public class UsuarioRepository implements UserRepository {  
    private UsuarioCrudRepository repository; 6 usages  
    private UserMapper mapper; 1 usage  
     @Override  
    public List<User> getAll(){ no usages  
        List<Usuario> consultados;  
        consultados = (List<Usuario>) repository.findAll();  
        return mapper.toUsers(consultados);  
    }
```

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Para corregir el error, adaptaremos el método `getAll` para coincidir con el de la interfaz

Necesitamos
almacenar lo
consultado por
el crudRepository

```
public class UsuarioRepository implements UserRepository {  
    private UsuarioCrudRepository repository; 6 usages  
    private UserMapper mapper; 1 usage  
  
    public List<User> getAll(){ no usages  
        List<Usuario> consultados;  
        consultados = (List<Usuario>) repository.findAll();  
        return mapper.toUsers(consultados);  
    }  
}
```

- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Para corregir el error, adaptaremos el método `getAll` para coincidir con el de la interfaz

El mapper hará la conversión

```
public class UsuarioRepository implements UserRepository {  
    private UsuarioCrudRepository repository; 6 usages  
    private UserMapper mapper; 1 usage  
  
    public List<User> getAll() { no usages  
        List<Usuario> consultados;  
        consultados = (List<Usuario>) repository.findAll();  
        return mapper.toUsers(consultados);  
    }  
}
```

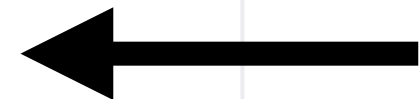
- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Para corregir el error, adaptaremos el método `getAll` para coincidir con el de la interfaz

```
public class UsuarioRepository implements UserRepository {  
    private UsuarioCrudRepository repository; 6 usages  
    private UserMapper mapper; 1 usage  
  
    public List<User> getAll(){ no usages  
        List<Usuario> consultados;  
        consultados = (List<Usuario>) repository.findAll();  
        return mapper.toUsers(consultados);  
    }
```

El mapper hará
la conversión



-
- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Adaptamos el método de la clase

```
public Usuario getById(Integer id){ no usages  
    return this.repository.findById(id).get();  
}
```

Al de la interfaz

```
@Override  
public Optional<User> getUser(int id) {  
    return Optional.empty();  
}
```

-
- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Utilizamos el cuerpo dentro de la interfaz pero eliminamos el método

```
public Usuario getById(Integer id){ no usages  
    return this.repository.findById(id).get();  
}
```

```
@Override  
public Optional<User> getUser(int id) {  
    Usuario u = this.repository.findById(id).get();  
    return Optional.of(this.mapper.toUser(u));  
}
```

-
- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Utilizamos el cuerpo dentro de la interfaz pero eliminamos el método

```
Optional<User> getUser(String login);
```

```
@Override
public Optional<User> getUser(String login){
    return Optional.of(mapper.toUser(this.repository.findByLogin(login)));
}
```


-
- Usaremos MapStruct para crear el mecanismo de transformación entre Usuarios y Users

Paso (3): Orientar los repositorios hacia el dominio

Utilizamos el cuerpo dentro de la interfaz pero eliminamos el método

```
User save(User user);
```

```
public User save(User u){  
    return mapper.toUser(this.repository.save(mapper.toUsuario(u)));  
}
```

Sorry

```
public class Usuario {  
    private String contrasena;  
    private String apellidoPaterno; 2 usages  
    private String apellidoMaterno; 2 usages  
    private String correo; 2 usages
```



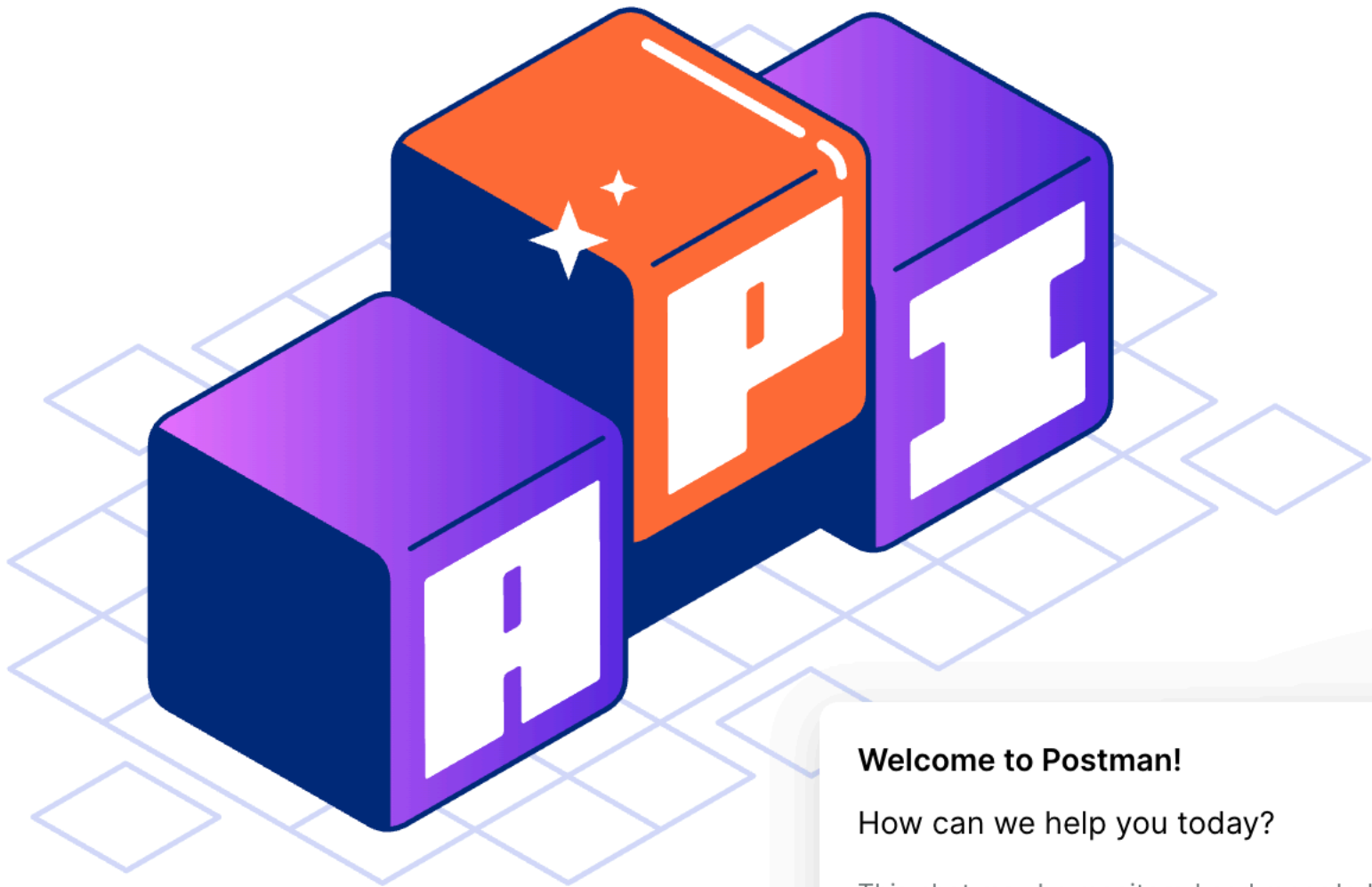


Your Complete API Platform, From Design to Delivery

Postman is the single platform for designing, building, and scaling APIs—together. Join over 40 million users who have consolidated their workflows and leveled up their API game—all in one powerful platform.

[Sign Up for Free](#)

Download the desktop app for



Welcome to Postman!

How can we help you today?

This chat may be monitored and recorded in accordance with our [Privacy Policy](#).