Tutorial - Parte 2

Spring

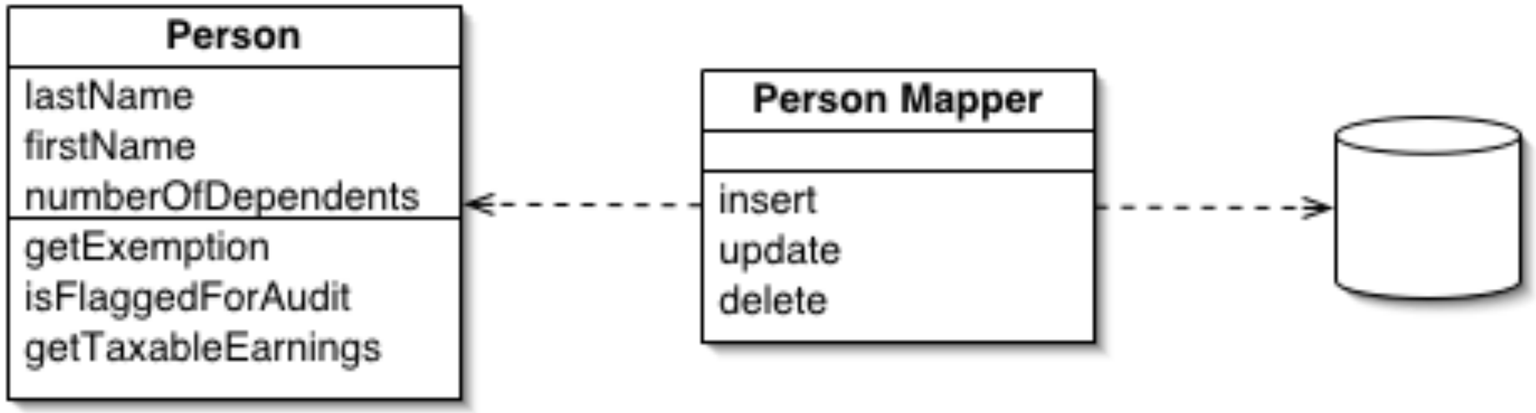
FCHAVEZ

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 domino podrían contener sentencias SQL, lógica de conexión a la base de datos, o conocimiento respecto a las estructura de las tablas. Y todas estas coas acoplan fuertemente la lógica de negocio al mecanismo de persistencia.
- Dificultad de pruebas: Cuando los objetos de dominio están atados a la base de datis, 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 aproporcionada por la base de datos

Data Mapper

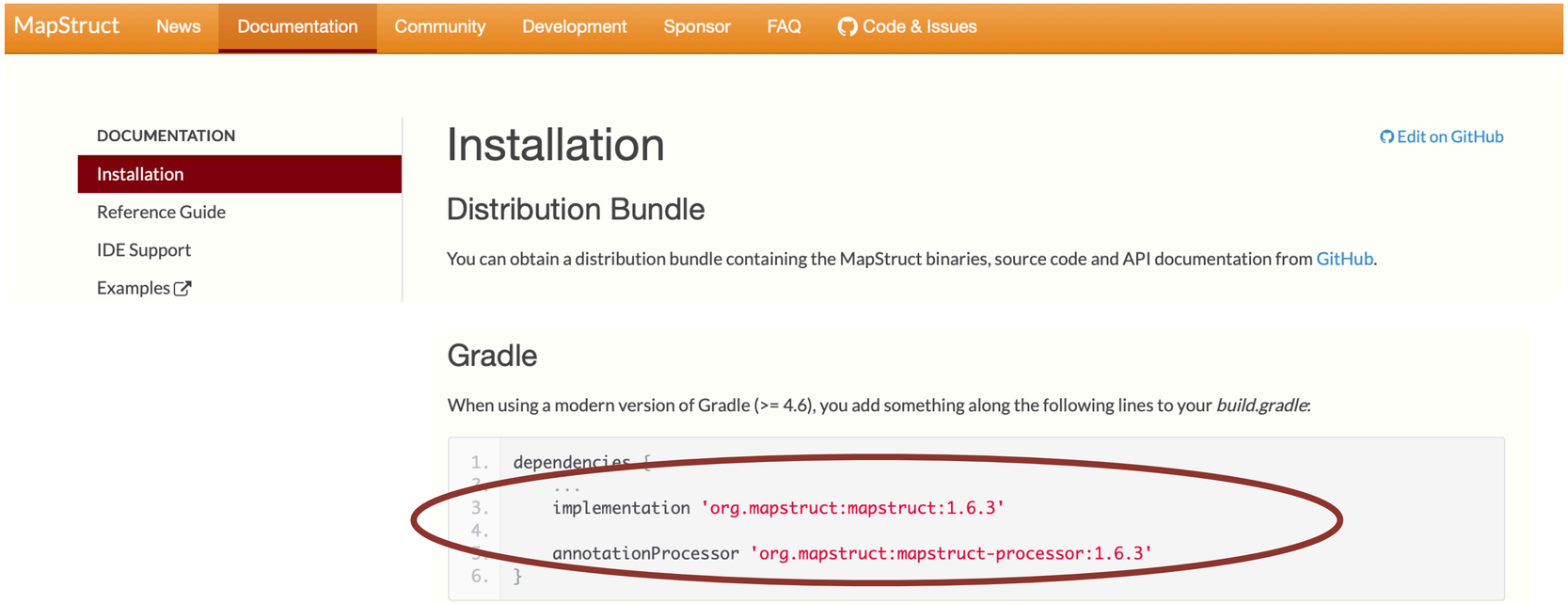
Implementación

Herramientas

Data Mapper

Implementación

Como usamos Gradle sólo necesitamos añadir la dependencia:



Data Mapper

Implementación

Como usamos Gradle sólo necesitamos añadir la dependencia:

build.gradle (nrda)

```
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  }
```

Data Mapper

Implementación

Como usamos Gradle sólo necesitamos añadir la dependencia:

build.gradle (nrda)

```
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
```

Data Mapper

Implementación

Como usamos Gradle sólo necesitamos añadir la dependencia:

build.gradle (nrda)

```
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
```

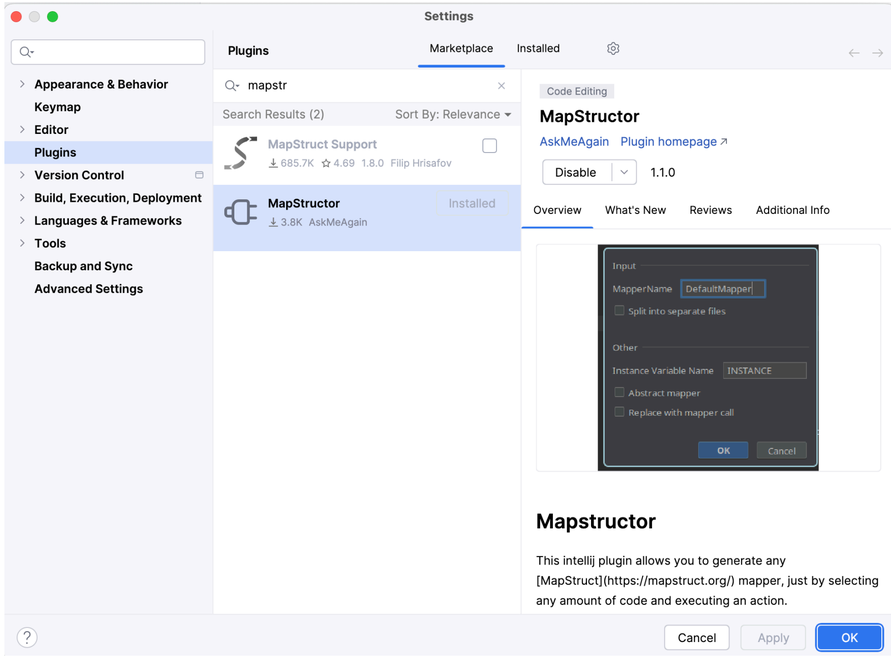
Actualizar cambios en Gradle

Data Mapper

Implementación

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

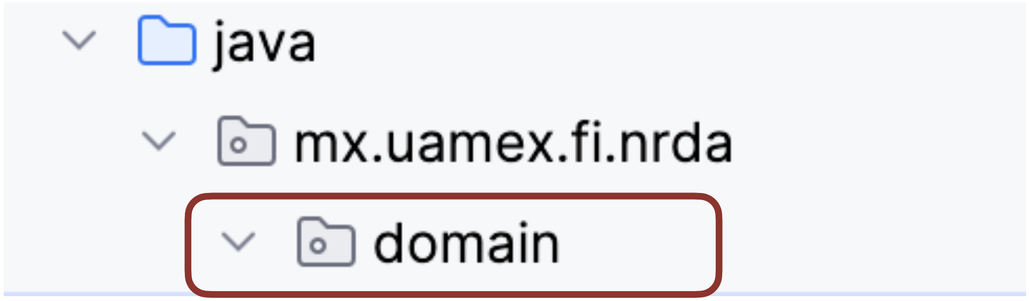
- Abrir IDE Settings



Data Mapper

Implementación

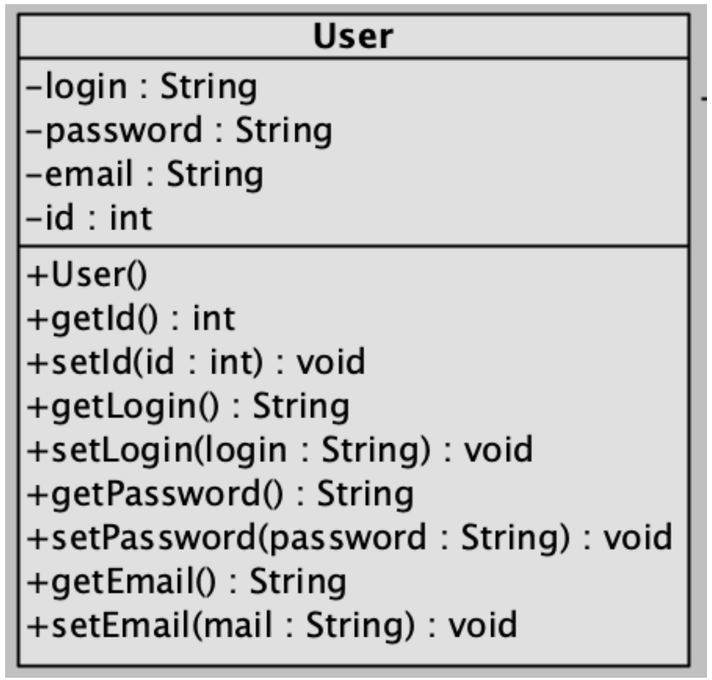
Definimos la(s) clase(s) del dominio



Data Mapper

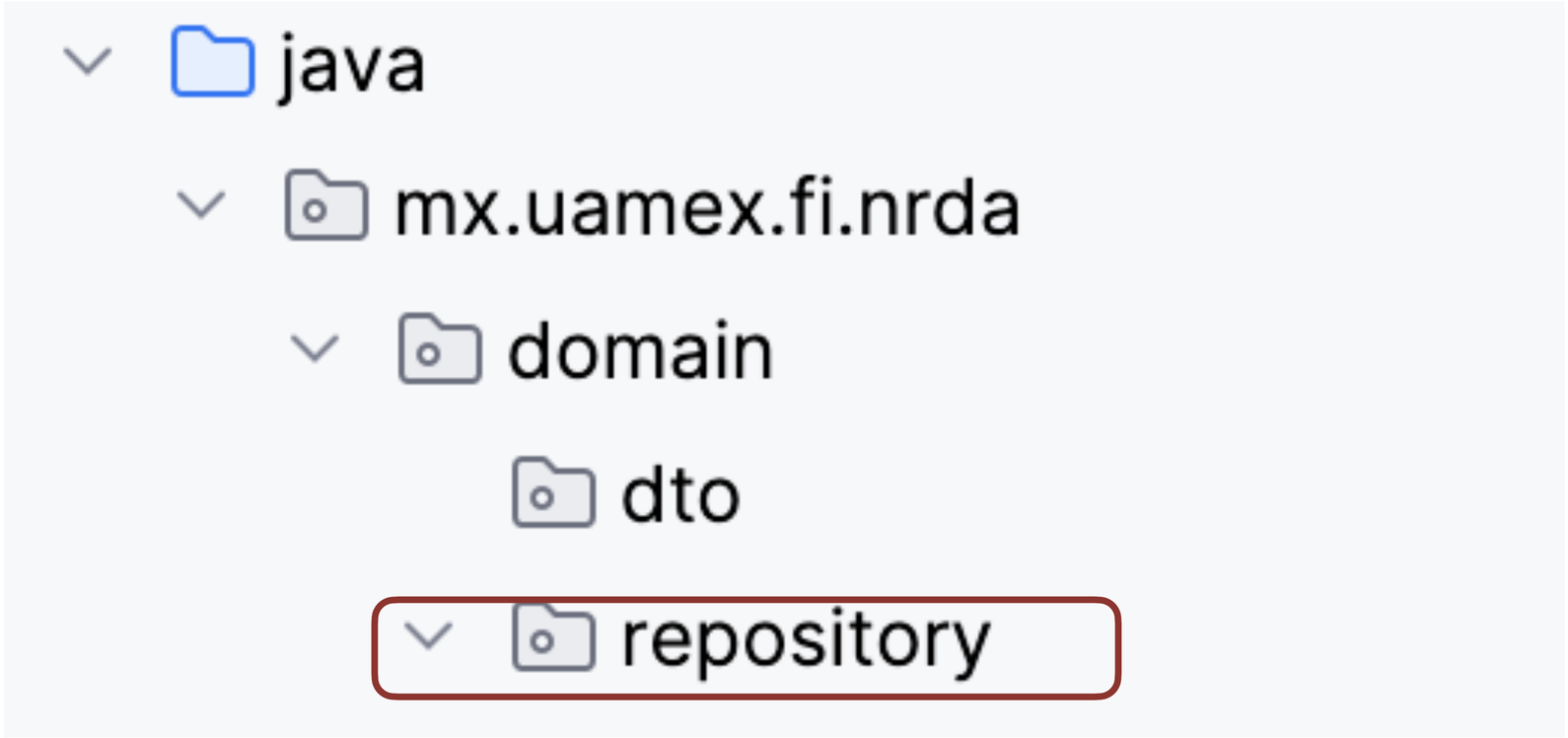
Implementación

Definimos la(s) clase(s) del dominio




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

public class User {  2 usages
    private int id;  2 usages
    private String login;  2 usages
    private String password;  2 usages
    private String email;  2 usages

    public int getId() {  no usages
        return id;
    }

    public void setId(int id) {  no usages
        this.id = id;
    }

```

Creamos un repositorio



Creamos un repositorio

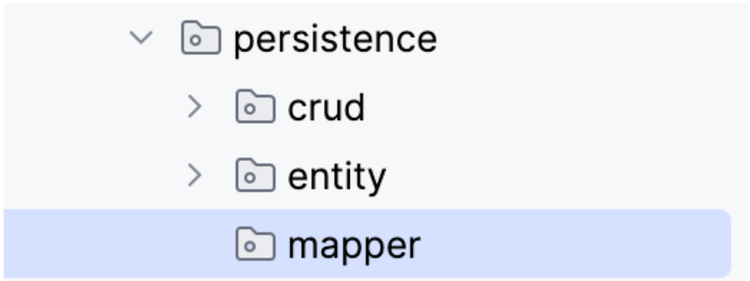
El repositorio será una interfaz que marcará las operaciones que deseamos realizar con los objetos del dominio

```java
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

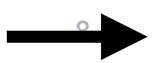


Creamos los mapeadores

- 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



```java
package mx.uamex.fi.nrda.persistence.mapper;

import mx.uamex.fi.nrda.domain.User;
import mx.uamex.fi.nrda.persistence.entity.Usuario;
import org.mapstruct.*;


@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);
}
```

Creamos los mapeadores

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

Paso (2): Creamos las interfaces para los mapeadores

```java
package mx.uamex.fi.nrda.persistence.mapper;

import mx.uamex.fi.nrda.domain.User;
import mx.uamex.fi.nrda.persistence.entity.Usuario;
import org.mapstruct.*;


@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);
}
```

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.

```java
package mx.uamex.fi.nrda.persistence.mapper;

import mx.uamex.fi.nrda.domain.User;
import mx.uamex.fi.nrda.persistence.entity.Usuario;
import org.mapstruct.*;


@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);
}
```

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

```java
package mx.uamex.fi.nrda.persistence.mapper;

import mx.uamex.fi.nrda.domain.User;
import mx.uamex.fi.nrda.persistence.entity.Usuario;
import org.mapstruct.*;


@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);
}
```

Creamos los mapeadores

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

Paso (2): Creamos las interfaces para los mapeadores

Atributo en la clase origen

```
@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);
}
```

Creamos los mapeadores

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

Paso (2): Creamos las interfaces para los mapeadores

Atributo en la clase destino

```java
@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);
}
```

Creamos los mapeadores

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

Paso (2): Creamos las interfaces para los mapeadores

```java
@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

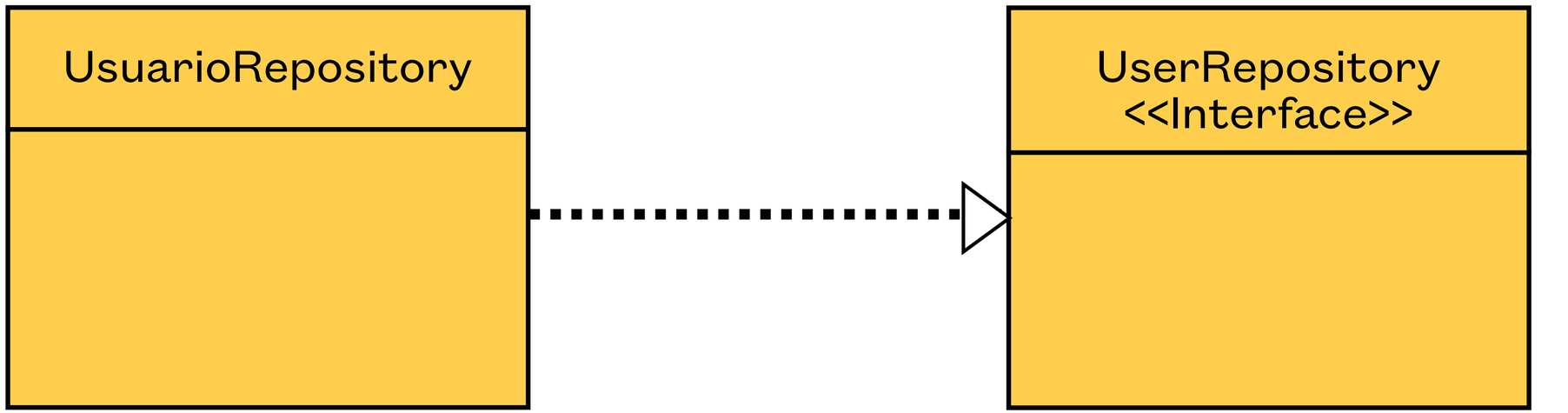
```
@Mapper(componentModel = "spring")
public interface UserMapper {
    @Mappings({
            @Mapping(source = "id",target = "id"),
            @Mapping(source = "login",target = "login"),
            @Mapping(source = "contrasena",target = "password"),
            @Mapping(source = "correo",target = "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



Clase del paquete `persistence` orientada hacia la base

Interfaz del paquete `domain.repository` orientada 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

- 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 sobreescribe 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
```

Postman

Instalar