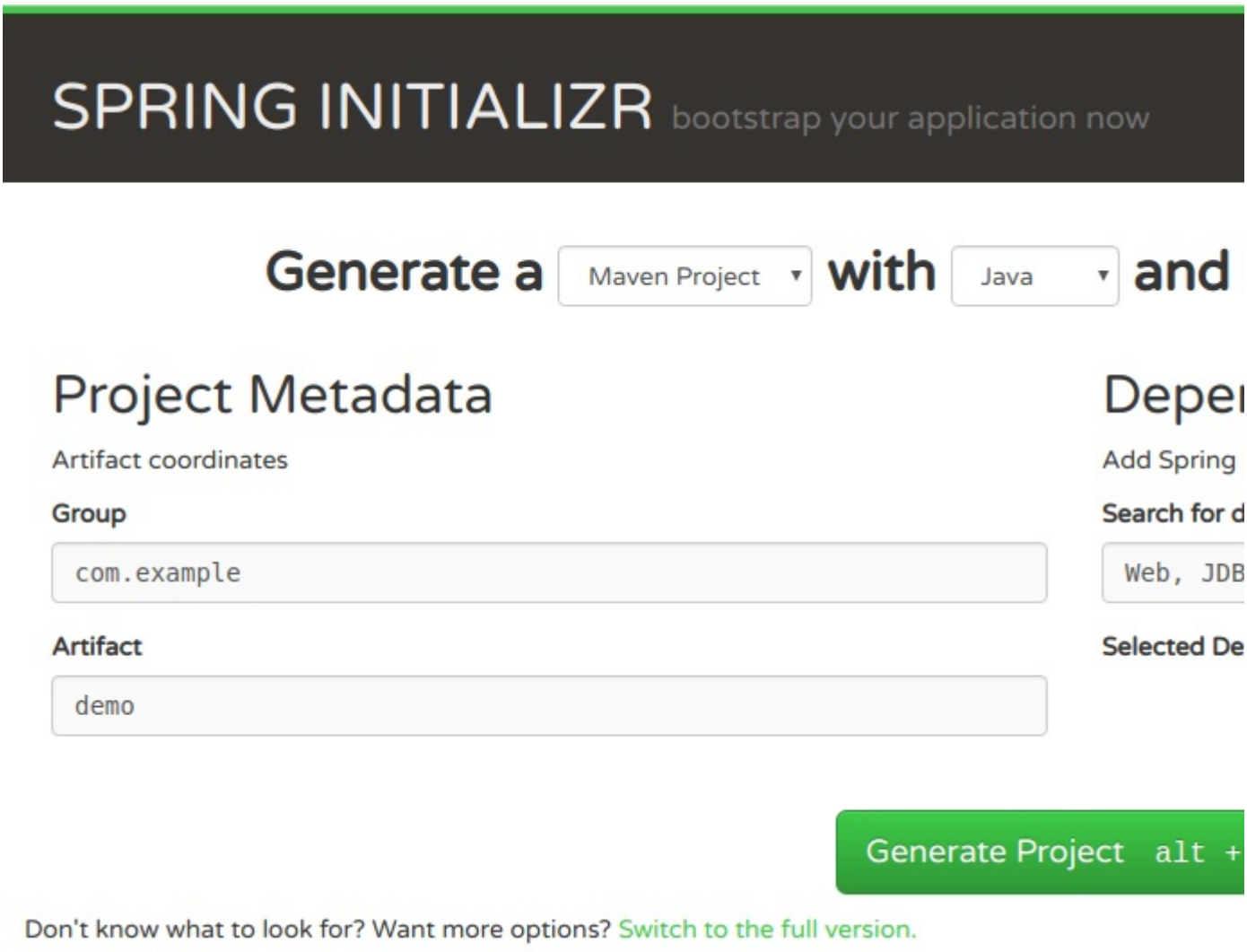
Desarrollo del API que consumira la Oracle DB

Introducción

Vamos a desarrollar un **API en Java** muy simple que nos permita interactuar con la Oracle DB a través del Weblogic que hemos estado trabajando. Será la propia app la que se encargue de **añadir datos** a la tabla que se define en el propio modelo de datos que veremos a continuación. Hay aplicaciones que se encargan de generar todo el código como por ejemplo https://start.spring.io/, pero en nuestro caso vamos a crearla nosotros desde cero. Si quisieras emplear la web, comentar que necesitarás las siguientes opciones:

- Dependencies:
  - Web
  - JDBC
  - DevTools



Nuestra app estará dividida en varias partes:

- **Ejecutable de la aplicación**: será la clase principal, la entrada a la app.

- **Modelo de datos**: Será el modelo de datos que tiene la tabla. Deberemos tener dos, el de la BBDD y el que emplearemos en la aplicación Java (así aislamos la capa de BBDD y la de aplicación).

- **Repositorio**: Es la interfaz con los métodos que van a atacar a la BBDD.

- **Servicio**: Es la capa encargada de la comunicación entre el repositorio y el recurso.

- **Recurso**: Serán los endpoints contra los cuales podamos acceder desde una aplicación externa como *CURL* o *Postman* entre otras.

A su vez la estuctura del proyecto estará formada por una **carpeta padre** que es la que contiene todo el proyecto en si y dos **carpetas hijo**:

- **app-web**: Será la que contenga la parte web. En nuestro caso lo vamos a dejar vacío.

- **app-rest-api**: Será la que tenga la lógica Java.

Añadir que es un proyecto basado en **Maven** por lo que constará de tres *pom.xml*, uno para el padre y otro para cada hijo.

Estructura Maven

El proyecto consta de tres *pom.xml*

Padre pom.xml

Será el que tenga las dependencias y la estructura del proyecto:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.a
    <modelVersion>4.0.0</modelVersion>

    <groupId>openwebinar.marvel.app</groupId>
    <artifactId>app</artifactId>
    <version>0.1.0-SNAPSHOT</version>
    <packaging>pom</packaging>
    <name>rest-api</name>
    <description>Demo for Openwebinar</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.0.M7</version>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncodin
        <project.reporting.outputEncoding>UTF-8</project.reporting.outpu
        <java.version>1.8</java.version>
    </properties>

    <repositories>
        <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
        <repository>
            <id>repository.springframework.maven.release</id>
            <name>Spring Framework Maven Release Repository</name>
            <url>http://maven.springframework.org/milestone/</url>
```

```
        </repository>
        <repository>
            <id>org.springframework</id>
            <url> http://maven.springframework.org/snapshot</url>
        </repository>
        <repository>
            <id>spring-milestone</id>
            <name>Spring Maven MILESTONE Repository</name>
            <url>http://repo.spring.io/libs-milestone</url>
        </repository>
        <repository>
            <id>spring-release</id>
            <name>Spring Maven RELEASE Repository</name>
            <url>http://repo.spring.io/libs-release</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>

    <modules>
        <module>app-rest-api</module>
        <module>app-web</module>
    </modules>

</project>
```

Hijo frontend pom.xml

En este caso lo dejaremos por defecto, ya que no vamos a añadir nada a nivel de frontal.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www
    <modelVersion>4.0.0</modelVersion>

    <artifactId>app-web</artifactId>
    <version>0.1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <parent>
        <groupId>openwebinar.marvel.app</groupId>
        <artifactId>app</artifactId>
```

```
        <version>0.1.0-SNAPSHOT</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                    <archive>
                        <manifest>
                            <addDefaultImplementationEntries>false</addD
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Hijo backend pom.xml

Tendrá la siguiente estructura:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://ma
    <modelVersion>4.0.0</modelVersion>

    <artifactId>app-rest-api</artifactId>
    <version>0.1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <parent>
        <groupId>openwebinar.marvel.app</groupId>
        <artifactId>app</artifactId>
        <version>0.1.0-SNAPSHOT</version>
    </parent>

    <properties>
```

```
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncodin
        <project.reporting.outputEncoding>UTF-8</project.reporting.outpu
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.apache.tomcat</groupId>
                    <artifactId>tomcat-jdbc</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc7</artifactId>
            <version>12.1.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Ejecutable de la aplicación

Como hemos visto en los *pom.xml* es una aplicación **Spring Boot** por lo que deberemos anotar esta clase con la anotación `SpringBootApplication` ya que va a ser el punto de entrada. Su código sera:

```
@SpringBootApplication
public class MyApiRestApplication extends SpringBootServletInitializer i
```

```
    public static void main(String[] args) {
        SpringApplication.run(MyApiRestApplication.class, args);
    }

}
```

Modelo de datos

Como ya hemos comentado el modelo de datos se debe corresponder con la tabla en la BBDD. En este caso **Spring Boot** se encargará de validar si existe la tabla que indicamos con el **modelo** y si no existe la **creará.** Así nos ahorramos el paso del `create table`.

Deberemos diferenciar dos clases diferentesl la de Spring Boot, que será la clase con las **anotaciones** apropiadas para poder **interactuar con la BBDD** y el modelo **sin anotaciones** para trabajar con la aplicación **Java.**

Es importante hacer esta distinción por dos facotres, la primera es para **independicar** el modelo de base de datos con la clase empleada en Java y el segundo es para **evitar** que las anotaciones añadan **código indeseado** a nuestra lógica.

El *modelo* para *BBDD* será:

```
@Entity(name = "avenger")
public class Avenger {
    @Id
    @Column(nullable = false)
    private Long id;

    @Column(length = 2)
    private String lang;

    @Column(length = 75)
    private String name;

    @Column(length = 75)
    private String actor;

    @Column(length = 1024)
    private String description;

    @Column(length = 2048)
    private String urlimage;

    public Avenger() {}
```

```
public Avenger(Long id, String lang, String name, String actor, Stri
    this.id = id;
    this.lang = lang;
    this.name = name;
    this.actor = actor;
    this.description = description;
    this.urlimage = urlimage;
}

public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getLang() {
    return lang;
}

public void setLang(String lang) {
    this.lang = lang;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getActor() {
    return actor;
}

public void setActor(String actor) {
    this.actor = actor;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

public String getUrlimage() {
```

```
        return urlimage;
    }

    public void setUrlimage(String urlimage) {
        this.urlimage = urlimage;
    }
}
```

Donde destacaremos las siguientes anotaciones:

- `@Entity(name = "avenger")`: Estamos definiendo la **entidad** (en BBDD) *avenger*, es decir la tablar *avenger*.

- `@Id`: Define la PK de la BBDD.

- `@Column(nullable = false)`: Cualquier atributo de la BBDD anotado con *Column* implicará que es una **columna** en la correspondiente BBDD.

Respecto al *modelo* de datos empleado en *Java* podremos **copiar y pegar** la clase, pero deberemos **eliminar todas las anotaciones**. Según desarrolladores, podremos añadir al final del nombre de la clase el término *DTO* (Data Transfer Object) para diferenciarlo del modelo de BBDD.

Repositorio

Es la interfaz donde están los métodos que se van a emplear para comunicar con la BBDD.

```
public interface AvengerRepository extends Repository<Avenger, Long> {
    List<Avenger> findAll();
}
```

En este caso tendremos un método para recuperar todas las entradas en la tabla. Otros tres métodos para recuperar información según una propiedad (en la consulta SQL que genere se traducira con un `WHERE columna = parametro de entrada al método`. Finalmente genera un método para añadir un registro en la tabla (método *save*) y otro para eliminar un registro (*delete*).

Servicio

Como ya hemos comentado es la capa encarga de de comunicar el repositorio con el recurso. Vamos a **copiar la interfaz** del **repositorio** pero quitando la herencia. Posteriormente crearemos una clase que implemente la nueva interfaz que hemos copiado, cada método de la clase deberá llamar al respectivo método del

repositorio.

La interfaz será:

```
public interface AvengerService {
    List<Avenger> findAll();
}
```

Mientras que la clase será:

```
@Component
public class AvengerServiceImpl implements AvengerService {
    private AvengerRepository repository;

    public AvengerServiceImpl(AvengerRepository avengerRepository) {
        this.repository = avengerRepository;
    }

    @Override
    public List<Avenger> findAll() {
        return repository.findAll();
    }
}
```

La anotación *@Component* le dice a Spring que esta clase es un componente y permite que ésta sera **autodetectada** por Spring. Para más información https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html.

Recurso

Con esta clase vamos a poder definir los endpoints. Su código será:

```
@RestController
@RequestMapping("avengers")
public class AvengerResource {

    private final Log log = LogFactory.getLog(AvengerResource.class);

    @Autowired
    private AvengerService avengerService;


    @GetMapping("/getAll")
    public List<Avenger> getAll() {
        log.info("getAll");
        return avengerService.findAll();
    }
```

```
}
```

- `@RestController`: Le decimos a Spring Boot que es un controlador REST. Totalmente imprescindible para poder acceder al endpoint.

- `@RequestMapping("avengers")`: Añade un subnivel a la URL de los endpoints. En este caso sería http://[xxx]/avengers/[recurso]/[parametro]. Lo veremos más afondo en próximas clases.

- `@Autowired`: Nos permite instanciar un objeto de Spring.

- `@GetMapping`: Los métodos anotados con *GetMapping* serán endpoints.

Recursos de la aplicación

Añadiremos este archivo necesario para WL. Donde destacaremos la etiqueta `wls:context-root` que será el nombre de nuestra aplicación cuando la vayamos a invocar (lo veremos con más detalle en la siguiente clase):

src/main/webapp/WEB-INF/weblogic.xml:

```
<wls:weblogic-web-app
        xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-we
        http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-w
    <wls:context-root>/app-rest-api</wls:context-root>
    <wls:container-descriptor>
        <wls:prefer-application-packages>
            <wls:package-name>org.springframework.*</wls:package-name>
            <wls:package-name>com.google.*</wls:package-name>
            <wls:package-name>antlr.*</wls:package-name>
            <wls:package-name>org.dom4j.*</wls:package-name>
            <wls:package-name>org.hibernate.annotations.common.*</wls:pa
            <wls:package-name>org.hibernate.*</wls:package-name>
            <wls:package-name>com.fasterxml.classmate.*</wls:package-nam
            <wls:package-name>javax.validation.*</wls:package-name>
            <wls:package-name>javax.el.*</wls:package-name>
            <wls:package-name>javax.persistence.*</wls:package-name>
        </wls:prefer-application-packages>
    </wls:container-descriptor>
</wls:weblogic-web-app>
```

src/main/resources/application.properties. Nos permite definie la conexión a la BBDD. Es importante poner en "**172.17.0.3**" la dirección del contenedor Oracle DB:

```
# Oracle DB
```

```
spring.main.banner-mode=off

# create and drop tables and sequences, loads import.sql
spring.jpa.hibernate.ddl-auto=create-drop

# Oracle settings

spring.datasource.driver-class-oracle.jdbc.driver.OracleDriver

# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@//172.17.0.3:1521/ORCLCDB.locald
spring.datasource.username=dummy
spring.datasource.password=dummy
```

src/main/resources/import.sql. Estarán los datos que queramos que se importen cuando se lance la aplicación (*en caso que no estén ya importados*):

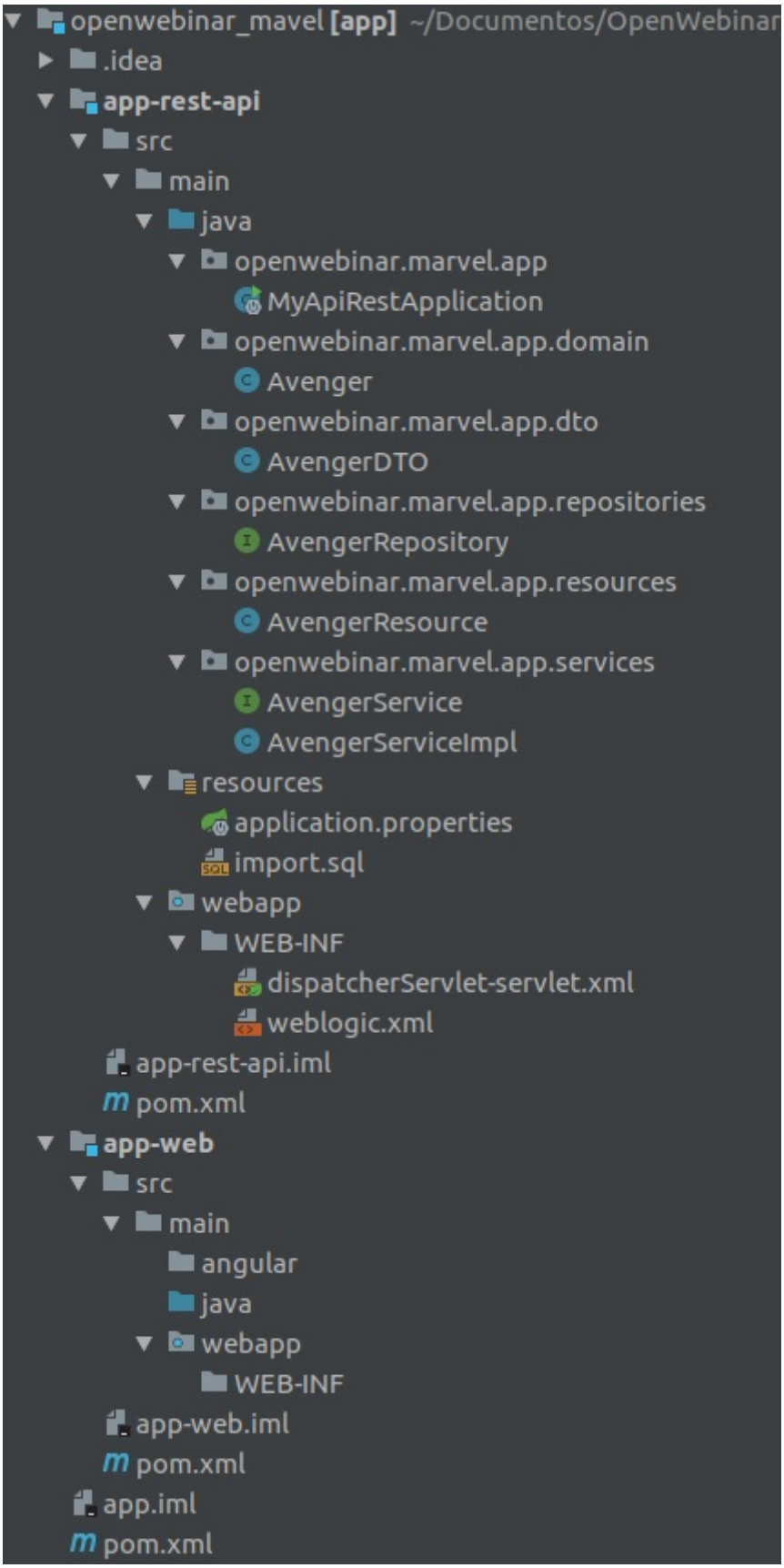
```
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE

INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE
INSERT INTO AVENGER (ID, LANG, NAME, ACTOR, DESCRIPTION, URLIMAGE) VALUE


COMMIT;
```

Compilar la aplicación

Como resultado veremos una estructura similar a la que se muestra en la imagen (las carpetas *.idea* y los archivos **.*iml* no son necesarios del proyecto):



Finalmente, para compilar el proyecto desde la carpeta padre deberemos lanzar el comando:

```
mvn clean install
```

El cual le está diciendo a Maven que debe limpiar todo el proyecto y generar una nueva aplicación. Si necesitas más información:

https://maven.apache.org/

https://maven.apache.org/download.cgi

El resultado de nuestra aplicación se encontrará en el path `openwebinar_mavel/app-rest-api/target`, donde *openwebinar_mavel* es la carpeta padre que contiene tanto al proyecto de backend como de frontend. El resultado será: