Práctica 4.01	Acceso a Datos
JakartaEE, Hibernate y relaciones	23-24

Table of Contents

Introducción

JPA es el estándar de persistencia de Java. JPA nos permite mapear nuestras entidades de dominio (nuestro modelo) directamente a una estructura de la base de datos y además permite manipular la información en BBDD como objetos en lugar de usar los componentes JDBC como `Connection`, `ResultSet`, etc.

API de persistencia de Java

JPA es una API que tiene como objetivo estandarizar la forma en la que accedemos a una base de datos relacional desde el software Java utilizando `Object Relational Mapping (ORM)`.

JPA no es más que una API y, por lo tanto, no proporciona ninguna implementación, sino que define y estandariza únicamente los conceptos de ORM en Java. Existen por tanto distintos proveedores que implementan dicha API.

- Hibernate
- EclipseLink
- DataNucleus

En nuestro caso usaremos la implementación que hace Hibernate de la API de JPA, ya que la mayor parte del estándar provino originariamente de Hibernate.

Mapeo relacional de objetos

El mapeo relacional de objetos es una técnica que se utiliza para crear un mapeo entre una base de datos relacional y los objetos de un software, en nuestro caso, objetos Java. La idea detrás de esto es dejar de trabajar con cursores o matrices de datos obtenidos de la base de datos, sino obtener directamente objetos que representen nuestro dominio.

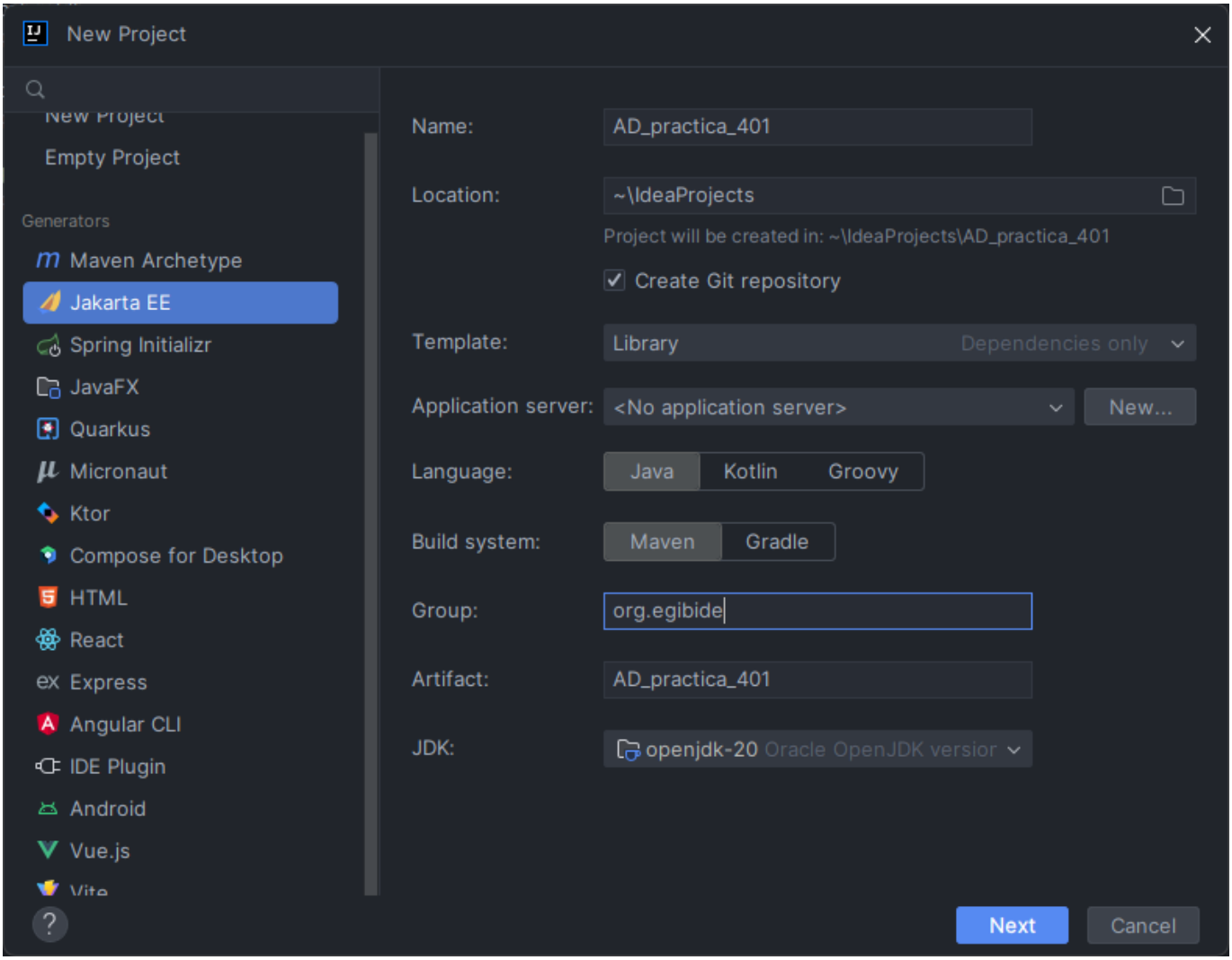
Para lograrlo, utilizamos técnicas para mapear nuestros objetos de dominio a las tablas de la base de datos para que se llenen automáticamente con los datos de las tablas. Entonces, podemos realizar la manipulación de objetos estándar en ellos.

A continuación vamos a crear un proyecto nuevo que nos servirá de ejemplo donde crearemos las entidades representadas en un modelo y las relaciones entre ellas.

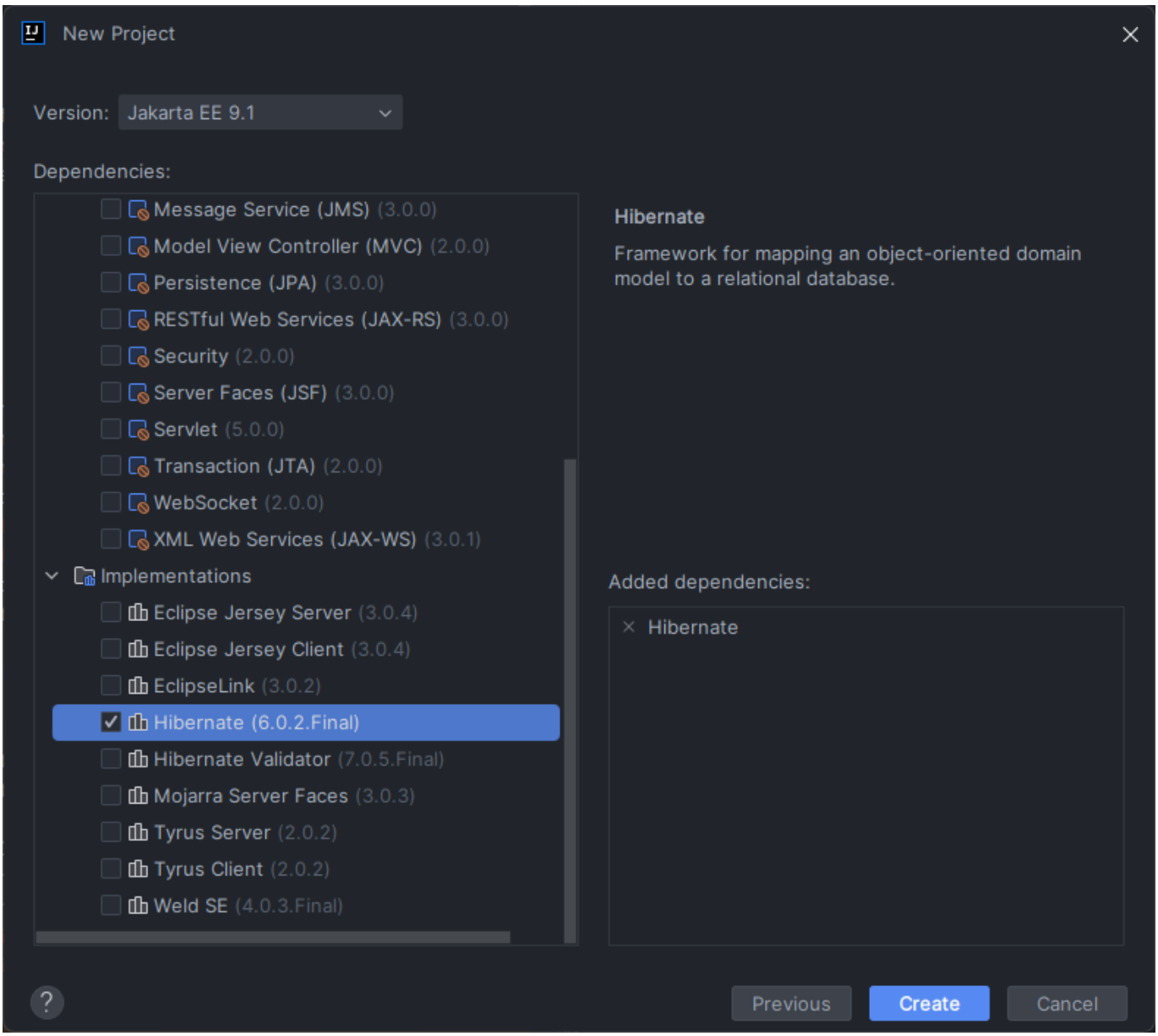
Ejemplo proyecto Jakarta EE

Aquellos desarrollos de aplicaciones que utilicen clases avanzadas de Java, utilizan además de las librerías pertenecientes al core de Java (JDK), lo que habitualmente se denomina(ba) Java EE (Enterprise Edition), actualmente renombrado a Jakarta EE tras la compra de Sun Microsystems por ORACLE. Incluyen librerías para el manejo de JavaBeans, Servlets, JSPs y distintos tipos de servicios web.

Es por ello, que vamos a crear un nuevo proyecto JakartaEE de la siguiente forma con el nombre `AD_practica_401`:

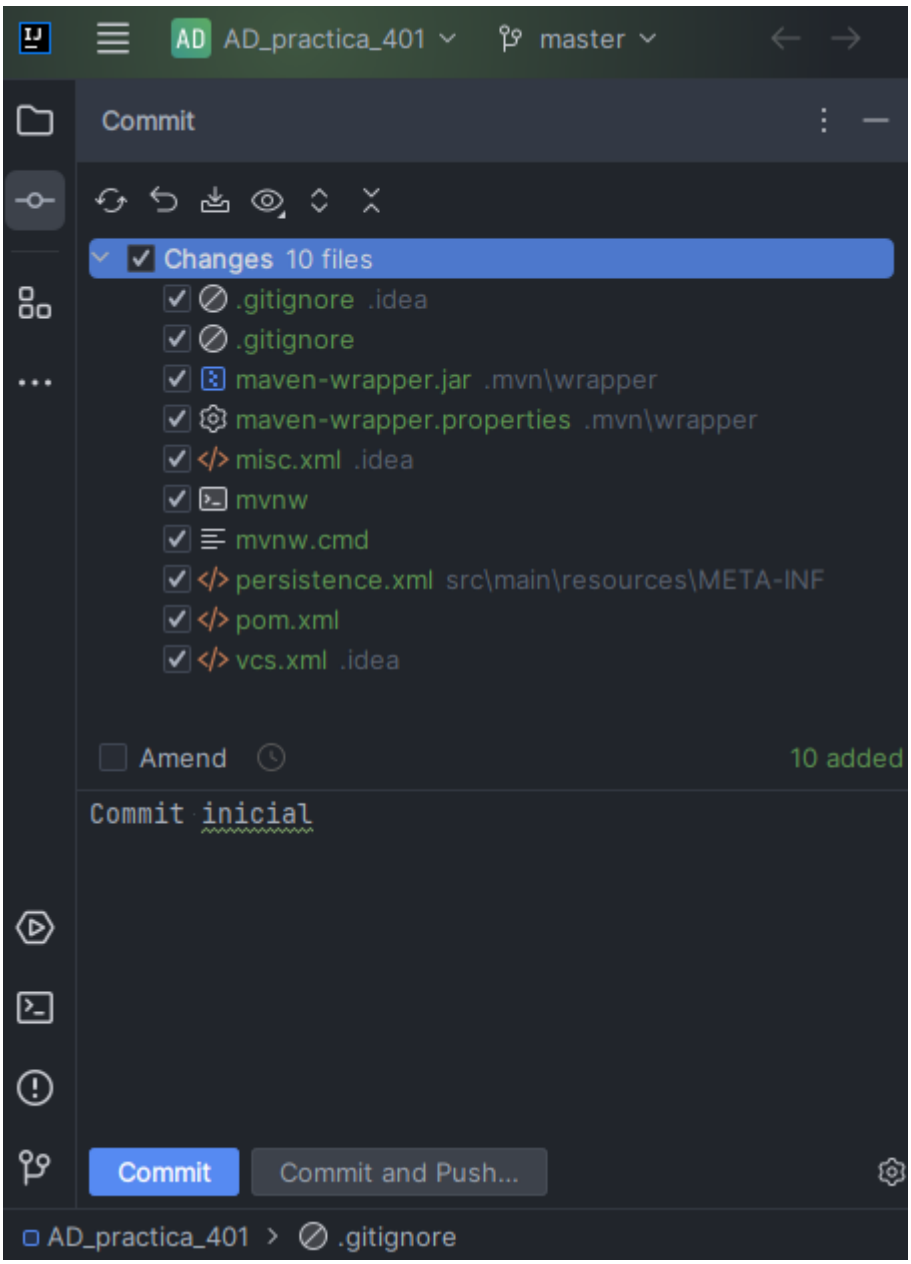


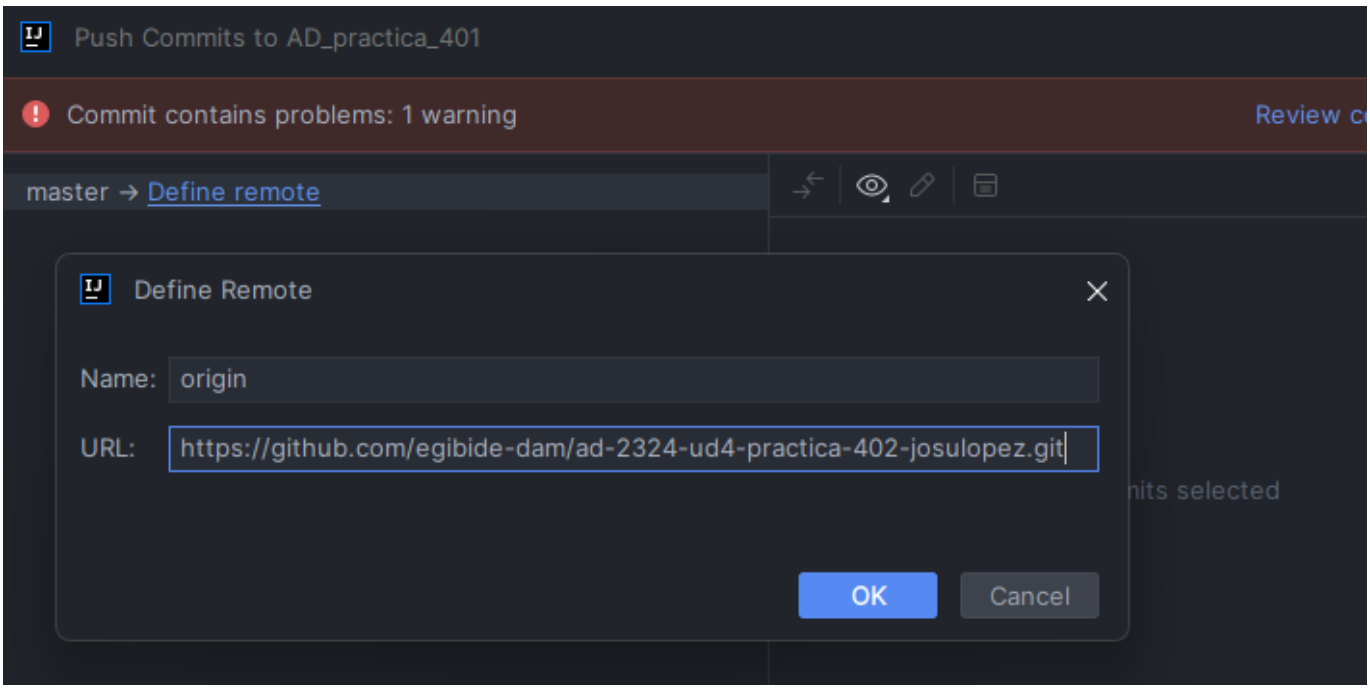
Y en la siguiente pestaña seleccionamos la dependencia con Hibernate:

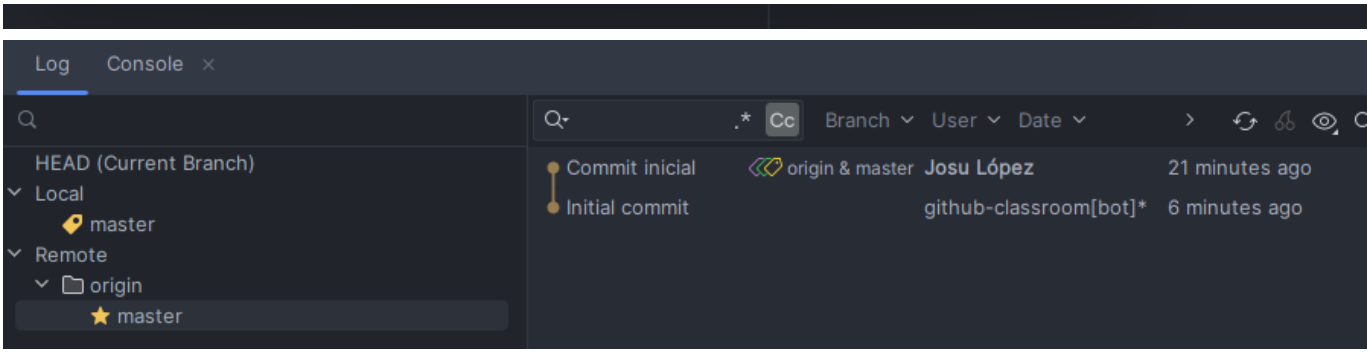


Esto nos creará un proyecto preparado para usar Hibernate como ORM.

Este es un buen momento para crear el commit inicial de Git y hacer push al repositorio de Github que se ha creado al aceptar la tarea, teniendo que definir para ello un remote a la hora de hacer push. Antes de hacer push, tendremos que hacer un fetch de la rama master del repositorio y a continuación un pull into 'master' using Rebase.


AD_practica_401
master
Commit
Changes 10 files
.gitignore .idea
.gitignore
maven-wrapper.jar .mvn\wrapper
maven-wrapper.properties .mvn\wrapper
misc.xml .idea
mvnw
mvnw.cmd
persistence.xml src\main\resources\META-INF
pom.xml
vcs.xml .idea
Amend
10 added
Commit inicial
Commit
Commit and Push...
AD_practica_401 › .gitignore

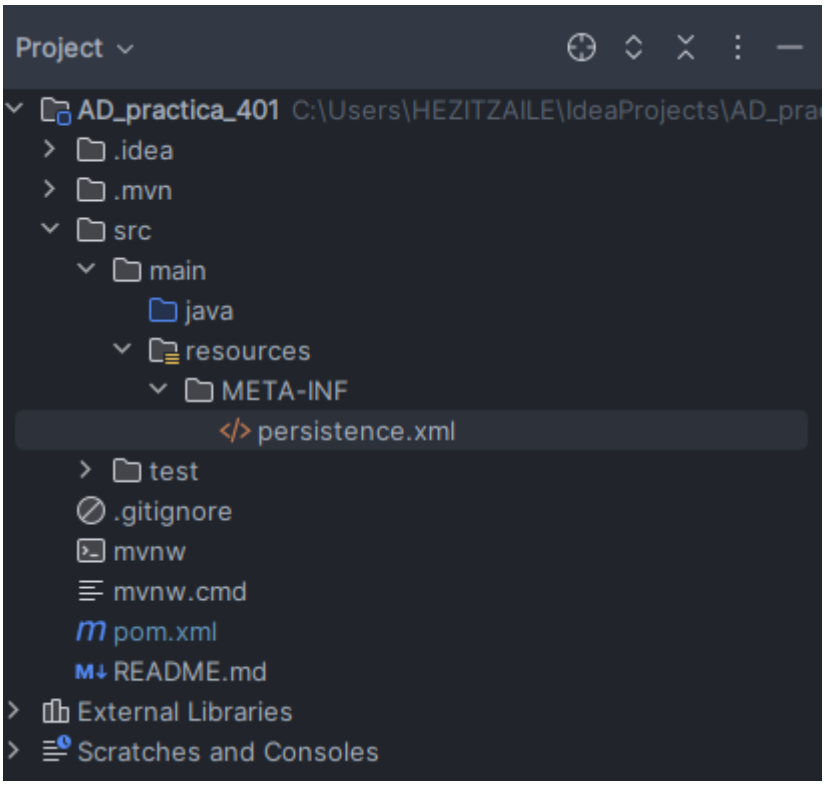

Push Commits to AD_practica_401
Commit contains problems: 1 warning
Review c
master → Define remote
Define Remote
Name: origin
URL: https://github.com/egibide-dam/ad-2324-ud4-practica-402-josulopez.git
OK
Cancel



Antes de empezar, recordad abrir el fichero `pom.xml` y añadir la dependencia con el conector para el SGDB que vayamos a utilizar tal y como hicimos en la unidad didáctica anterior.

Además de eso, dentro del gestor de BBDD que tengamos creamos un schema nuevo denominado `AD_aulavirtual`.

Dentro de la estructura de nuestro proyecto, vemos que existe un fichero `persistence.xml`:



Este archivo se utiliza para configurar JPA, indicando cuál será nuestro proveedor de BD, la BD de datos vamos a usar y cómo conectarnos, cuáles son las clases a mapear, etc...

Dentro de dicho fichero, añade las siguientes propiedades:

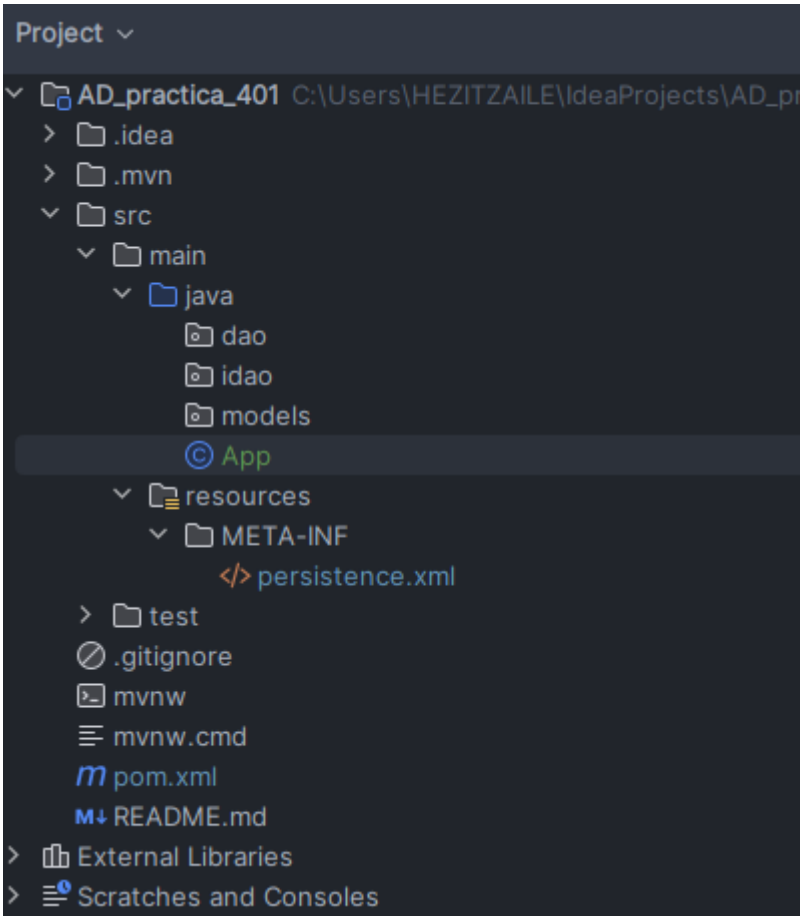
```
<properties>
      <property name="hibernate.connection.url"
value="jdbc:mariadb://localhost:33060/AD_aulavirtual"/>
      <property name="hibernate.connection.driver_class" value="org.mariadb.jdbc"/>
      <property name="hibernate.connection.password" value="12345Abcde"/>
      <property name="hibernate.connection.username" value="root"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.MariaDBDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
</properties>
```

Sobre las propiedades de Hibernate que hemos definido comentar alguno que puede resultar novedoso:

- `hibernate.hbm2ddl.auto` : permite configurar una herrienta hbm2dll, con el fin de generar y mantener el schema de BD según las entidades definidas en el proyecto a través de las anotaciones. Los valores admtidos por esta propiedad son:
  - `update` : consultará la API del controlador JDBC para obtener los metadatos de la base de datos y luego Hibernate compara el modelo de objetos que crea basándose en la lectura de sus clases anotadas o asignaciones XML de HBM e intentará ajustar el esquema sobre la marcha.
  - `create` : Las tablas se generan de acuerdo con los modelos cada vez que se carga la hibernación y elimina lo que hubiera anteriormente. Utilizado sobre todo en entornos de pruebas.
  - `create-drop` : Las tablas se generan de acuerdo con los modelos cada vez que se carga la hibernación, pero la tabla se elimina automáticamente tan pronto como se cierra sesión. Utilizado sobre todo en entornos de pruebas.
  - `none` : sobre todo en entornos de producción donde el esquema debería mantenerse intacto.

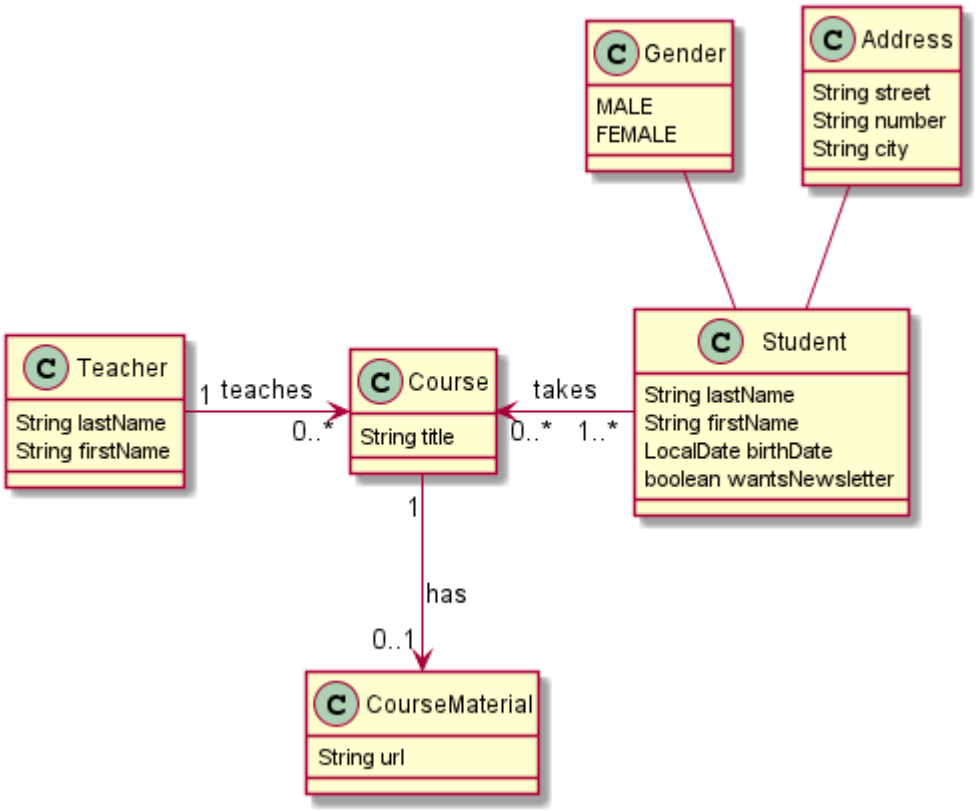
  Otra estrategia sería mediante librerías de migración, como [Flyway](Migrations - Migrations - Flyway by Redgate • Database Migrations Made Easy.), que permiten definir y mantener el schema en BD mediante ficheros independientes y que se ejecutan normalmente al arranque de la aplicación.
- `hibernate.show_sql` : podemos indicar esta propiedad a true en caso de que queramos que en los logs de ejecución nos vaya mostrando las queries que se van ejecutando.

Por otro lado, crea la siguiente estructura de carpetas semejante a nuestros proyectos dao del pasado:



Esquema del modelo

A continuación, os presento el esquema de las entidades y las relaciones que vamos a representar en este ejemplo:



Si nos paramos a analizar un poco el esquema vemos las siguientes relaciones:

- one-to-one: existen tres relaciones de este tipo en el esquema. La primera entre `Student` y `Address`, la segunda entre `Student` y `Gender` y la tercera sería entre `Course` y `CourseMaterial`. Es esta última la que implementaremos en el proyecto.
- one-to-many: la relación entre `Teacher` y `Course` sería una relación de este tipo. Un profesor tiene de cero a muchos cursos, pero un curso pertenece a un único profesor.
- many-to-many: entre `Student` y `Course` tendremos una relación N-M, ya que un estudiante puede formar parte de muchos cursos y en un curso hay muchos estudiantes.

One-to-Many/Many-to-One

Las relaciones one-to-many y many-to-one, son distintas caras de la misma moneda. Una vez establezcamos una de ellas, la inversa nos vendrá dada, aunque habrá que indicarla explícitamente.

Para ello, lo primero que tenemos que hacer es definir las entidades que forman parte de esta relación. En este caso son `Course` y `Teacher`. Ya que como hemos visto anteriormente, un `Teacher` puede tener muchos cursos `Course`, pero un curso pertenece a un único profesor.

Tenemos que definir correctamente en el proyecto nuestras entidades y sus relaciones, ya que será el ORM quien se encargue de actualizar y definir las tablas en la BD que le hayamos indicado en el `persistence.xml`.

Para ello, comenzamos por definir las clases `Course` y `Teacher` dentro de nuestro package `models`:

```
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;
    @NotNull
    private String title;
    //include getters, setters...
}
```

```
@Entity
public class Teacher {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;
    @NotNull
    private String firstName;
    @NotNull
    private String lastName;
    //include getters, setters...
}
```

Con respecto a los atributos comentar que siguiendo las anotaciones proporcionadas por los paquetes de persistencia somos capaces de representar nuestro modelos, con las restricciones que quisieramos después en su implementación en BD:

- `Entity`: indica que esa clase representa una entidad.
- `Id`: Aquí le indicamos qué campo es primario en nuestra entidad, y como estrategia de generación de claves primarias, hemos escogido `IDENTITY`, que viene a ser un autoincremental.
- `NotNull`: Podemos indicar que un determinado campo es obligatorio y no permitimos nulo.

- También podríamos restringir los campos usando otras JPA Annotations - Hibernate Annotations | DigitalOcean.

Lo siguiente que debemos hacer es marcar la relación en las entidades.

Dentro de la entidad `Course`, vamos a establecer la relación usando la anotación `ManyToOne`, definiendo un nuevo atributo de tipo Teacher, donde se recoja el profesor asociado a un objeto curso, el cual se obtendrá gracias a la relación definida:

```
@ManyToOne (optional = false, fetch = FetchType.EAGER)
@JoinColumn(name = "teacher_id", referencedColumnName = "id")
private Teacher teacher;
//establecer getter y setter
```

Dentro de la entidad `Teacher` vamos a establecer la relación inversa usando la anotación `OneToMany`, y estableceremos un nuevo atributo de tipo lista donde asociar los cursos asociados a un objeto profesor:

```
@OneToMany(mappedBy = "teacher",  fetch = FetchType.LAZY)
private List<Course> courses;
//establecer getter y setter
```

Nota: *se puede establecer la anotación `JoinColumn` también junto con la anotación `OneToMany` y funcionaría igual, lo único que los que han implementado el ORM indican que desde el punto de vista de performance es mejor establecer la `JoinColumn` en la entidad que tenga la FK, es decir en la entidad a la que pertenece la relación.*

En el código anterior, observamos que para establecer la relación `OneToMany` no nos hace falta volver a definirla, sencillamente hacemos referencia al campo donde está definida la inversa de la relación.

Eager vs. Lazy Loading

Otra cosa que cabe destacar de cómo hemos definido las relaciones son el tipo de fetch. Esto tiene mucha importancia en el consumo de memoria ya que si nos traemos muchos objetos a la memoria del programa podríamos tener problemas de rendimiento u overflow.

- ManyToOne: hemos indicado EAGER aunque si no ponemos nada es así por defecto. Esto indica que cuando consultemos por un objeto de tipo Course (1), el objeto Teacher asociado (1) también nos lo traeremos a la memoria del programa.

- OneToMany: hemos indicado LAZY, aunque lo es por defecto. Esto indica que no nos traeremos a la memoria del programa todos los objetos de tipo `Course` para un `Teacher` concreto. Tendremos que específicamente hacer un `getCourses` para que así sea. OJO: con añadir la lista de cursos al toString!

Clase abstracta DAO

Dentro de la carpeta dao creamos una clase abstracta genérica llamada `Dao` de la siguiente forma:

```
package dao;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;
abstract class Dao<T, K> {
    EntityManager em = Persistence
            .createEntityManagerFactory("default")
            .createEntityManager();

    public EntityManager getEntityManager() {
        return this.em;
    }
    public abstract T find(K id);
    public T create(T t) {
        EntityTransaction entityTransaction= em.getTransaction();
        entityTransaction.begin();
        em.persist(t);
        entityTransaction.commit();
        return t;
    }
    public T update(T t) {
        EntityTransaction entityTransaction= em.getTransaction();
        entityTransaction.begin();
        em.merge(t);
        entityTransaction.commit();
        return t;
    }
    public void delete(T t) {
        EntityTransaction entityTransaction= em.getTransaction();
        entityTransaction.begin();
        em.remove(t);
        entityTransaction.commit();
    }
}
```

Si nos fijamos en ella esta clase nos permite realizar las operaciones de CRUD para cualquier implementación del DAO que extienda de esta clase.

Implementación de los DAO

Para el caso de la implementación de `TeacherDAOImpl` esta tendrá que extender de la clase abstracta `Dao` y además implementar su correspondiente interfaz `ITeacherDao`. En este momento no vamos a meter ningún método nuevo a nivel de interfaz, pero dejamos que la implementación la implemente igualmente para el futuro.

```
public class TeacherDaoImpl extends Dao<Teacher,Integer> implements ITeacherDao {
    @Override
    public Teacher find(Integer id) {
        Teacher teacher = (Teacher) em.find(Teacher.class, id);
        return teacher;
    }
}
```

Para probar el método find, tendremos que tener algún dato en BD. Para ello crea un profesor, y algún curso asociado al profesor que has creado. Finalmente llama desde `App` al método find de `TeacherDaoImpl`:

```
TeacherDaoImpl teacherDao = new TeacherDaoImpl();
System.out.println(teacherDao.find(1));
```

Observa como dado un objeto Teacher, nos devolverá una lista con los cursos asociados.

> **Tarea 1**: haz lo mismo que acabamos de hacer pero para el método find de `CourseDaoImpl`. De tal forma que dado un curso, el objeto `Course` devuelto devuelva también el objeto `Teacher` asociado.
>
> **Tarea 2**: Prueba para alguna de las dos entidades anteriores el resto de métodos del CRUD: update, create, delete.

Opcionalidad

La relación `OneToMany` es por defecto siempre opcional, es decir que puede haber un objeto `Teacher` sin lista de cursos asociados. Sin embargo, por defecto la relación `ManyToOne` es opcional pero si quisieramos cambiar este compotamiento lo tendríamos que indicar así:

```
@ManyToOne(optional = false)
@JoinColumn(name = "teacher_id", referencedColumnName = "id")
private Teacher teacher;
```

De esta manera, siempre tendríamos que asociar un objeto `Teacher` a un objeto `Course`.

> **Tarea 3**: probar a crea un objeto Course y a persistirlo en BD. Os debería de dar un error.
>
> **Tarea 4**: Crear un Teacher, crear un Course. Asociar el profesor al curso que habeis creado y probar a persistirlo. Os debería de saltar un error tipo:

```
...detached entity passed to persist: ...`
```

Eso ocurre porque hemos intentado persistir el objeto `Course` antes de que el objeto `Teacher` lo estuviera. Así que habría que hacer algo así:

```
Teacher teacher = new Teacher();
teacher.setLastName("Zengotitabengoa");
teacher.setFirstName("Jokin");
teacherDao.create(teacher);

Course course2 = new Course("");
course2.setTeacher(teacher);
courseDao.create(course2);
```

Operaciones tipo Cascading

El anterior problema se puede solucionar, y es indicándole al atributo donde está la relación `ManyToOne` que el tipo de `cascade=CascadeType.PERSIST`. De esa manera hibernate sabe que esa relación y por tanto el objeto asociado lo tiene que persistir igualmente en caso de una operación de tipo PERSIST, es decir al añadir un registro nuevo de tipo course, antes tendrá que salvar el objeto teacher.

```
@ManyToOne(optional = false, cascade = CascadeType.PERSIST)
@JoinColumn(name = "teacher_id", referencedColumnName = "id")
private Teacher teacher;
```

Los tipos de cascada permitidos son:

- **CascadeType.ALL**: se aplican todos los tipos de cascada.

- **CascadeType.PERSIST**: las operaciones de guardado en la base de datos de las entidades padre se propagarán a las entidades relacionadas.
- **CascadeType.MERGE**: las entidades relacionadas se unirán al contexto de persistencia cuando la entidad propietaria se una.
- **CascadeType.REMOVE**: las entidades relacionadas se eliminan de la base de datos cuando la entidad propietaria se elimine.
- **CascadeType.REFRESH**: las entidades relacionadas actualizan sus datos desde la base de datos cuando la entidad propietaria se actualiza.
- **CascadeType.DETACH**: se separan del contexto de persistencia todas las entidades relacionadas cuando ocurre una operación de separación manual.

Por defecto, no se aplica ninguna operación de cascada.

Además, se podrían combinar varias tal que: `cascade = { CascadeType.PERSIST, CascadeType.MERGE })`.

Las operación en cascada también se pueden indicar en el `OneToMany`. Pero como regla general, hay que saber lo que se está haciendo o lo que va a ocurrir en caso de animarnos a definir una operación en cascada.

OnetoOne

Si echamos un ojo al esquema inicial, la relación entre `Course` y `CourseMaterial` está definida como 1-1. Lo primero que hay que hacer es definir la entidad `CourseMaterial` con sus atributos:

```
@Entity
public class CourseMaterial {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;
    @NotNull
    private String url;
    //getters, setters...
}
```

Ahora definiremos la relación one-to-one, que como hemos dicho antes, será mejor si la definimos allí donde esté la FK, que en el caso concreto sería `CourseMaterial`:

```
    @OneToOne(optional = false)
    @JoinColumn(name = "course_id", referencedColumnName = "id")
    private Course course;
```

Añadimor el `optional = false` porque no tiene sentido crear un material si su curso asociado.

Para hacer la relación bidireccional, vamos a la clase `Course` y añadimos la relación inversa:

```
    @OneToOne(mappedBy = "course")
     private CourseMaterial material;
```

En este caso, dejamos el `optional` por defecto a true, ya que puede haber un curso sin material asociado.

> **Tarea 5**: Crea un objeto de tipo `CourseMaterial` y persístelo en BBDD. Después has un find() sobre el objeto creado, y también sobre el curso a que está asociado para ver el tipo de fetch que tiene el one-to-one por defecto. Debería ser EAGER.

Many-to-many

Esta relación la dejamos para el final, porque implica algo más de trabajo. A la hora de implementar esta relación sabemos que se deberá crear una tabla intermedia entre las dos entidades que participan en la relación. En concreto, en nuestro ejemplo, hay una relación `ManyToMany` entre la entidad `Student` y `Course`:

Por ejemplo, podemos definir dicha relación en la entidad `Course` de la siguiente manera:

```
    @ManyToMany
    @JoinTable(
            name = "students_courses",
            joinColumns = @JoinColumn(name = "course_id", referencedColumnName =
"id"),
            inverseJoinColumns = @JoinColumn(name = "student_id", referencedColumnName
= "id")
    )
    private List<Student> students;
```

Si nos fijamos en el detalle de la relación, en este caso le indicamos el nombre de la tabla a través de la cual se relacionan `students_courses`. Es habitual encontrar esta convención de nombres, donde el nombre de la tabla, es el nombre de las dos entidades separadas por '__' .

- `joinColumns` define cómo configurar la columna de join a la que pertenece la relación (en este caso será de Course que es donde la estamos definiendo), y su correspondiente campo 'id' en la tabla de Courses. Los nombres de las tablas no hace falta especificarlas, serán el nombre de la entidad en plural. Aunque se podría especificar.

- `inverseJoinColumns` define la columna a través de la cual obtendremos los datos de la otra entidad que forma parte de esta relación, en este caso `Student`. Los nombres de las tablas no hace falta especificarlas, serán el nombre de la entidad en plural. Aunque se podría especificar

Puesto que hemos definido un atributo lista de estudiantes dentro de la clase de dominio `Courses`, necesitaremos añadir un método que nos permita añadir estudiantes a la lista:

```
    public void addStudent(Student student) {
        this.students.add(student);
    }
```

> **Tarea 6**: Crea varios objetos estudiantes, y sobre un objeto curso nuevo persiste los datos. Haz lo mismo pero modificando un objeto cursos existente. En caso de que te de error piensa en cómo solucionarlo. Una vez funcione, haz un find del curso nuevo introducido. ¿Qué tipo de fetch se está realizando?
>
> **Tarea 7**: Crea la inversa de la relación definida. Y pruébalo convenientemente, asocia varios cursos a un objeto nuevo estudiante que crees y persiste los datos. Después consulta los datos del estudiante introducido.
>
> **Tarea 8**: Crea un nuevo método en la interfaz DAO de Course, que se llame listCoursesByLastName, el cual pasándole una cadena de texto, obtendrá el listado de cursos de los profesores que tengan el apellido como el parámetro que se les pasa.
>
> **Tarea 9**: Crea un método en la interfaz de Student, el cual obtenga el listado de estudiantes para un id de curso concreto.