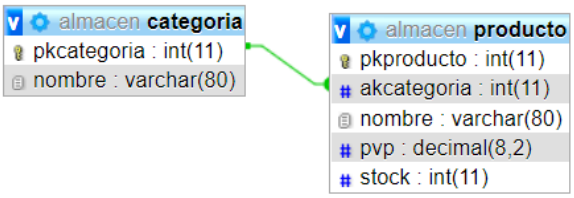
Ejemplo – Spring Data JPA.

Vamos a realizar el acceso a datos a una base de datos relacional, empleando las herramientas de Spring para JPA.

Base de datos: Almacén

La base de datos muestra un almacén de productos, donde cada producto pertenece a una categoría y la misma categoría puede contener varios productos.



Dependencias – Spring Initializr

Añadimos dependencias para

- Spring Data JPA
- MySQL Driver

Una vez creado el proyecto con las dependencias Maven que necesitamos, lo abrimos en Netbeans y especificamos los parámetros para la conexión a la BD.

Parámetros para la conexión: application.properties

Además de la URL, el usuario y la contraseña, vamos a incluir parámetros propios de JPA para indicar que queremos hacer con la base de datos cuando nos conectemos.

Las opciones que tenemos para la configuración del acceso spring.jpa.hibernate.ddl-auto son las siguientes:

- none: Para indicar que no queremos que genere la base de datos
- update: Si queremos que la genere de nuevo en cada arranque
- create: Si queremos que la cree pero que no la genere de nuevo si ya existe.

En el ejemplo, escogemos la opción “none” puesto que tenemos la bd creada y precargada, como si estuviese en producción.

Una vez guardado el archivo, ejecutamos el proyecto para comprobar que se conecta correctamente a la BD.

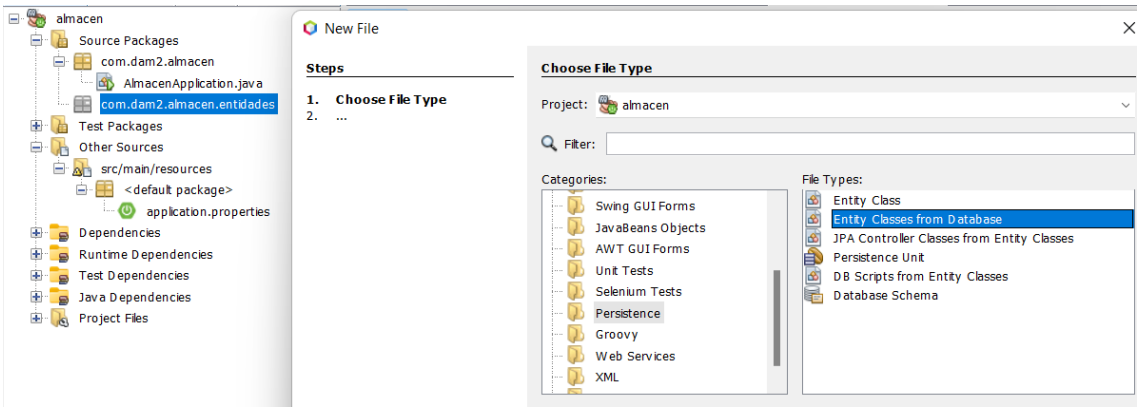
```
#parámetros de conexión a la BD
spring.datasource.url = jdbc:mysql://localhost:3306/almacen?zeroDateTimeBehavior=CONVERT_TO_NULL
spring.datasource.username = admin
spring.datasource.password = Admin-123
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver

# Otras propiedades JPA interesantes para nuestra comexión

# Configuracion para el acceso a la Base de Datos
spring.jpa.hibernate.ddl-auto=none
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

Definir las entidades y las interfaces para el modelo

Para mapear las tablas y las relaciones podemos hacer uso del asistente JPA para persistencia que hemos utilizado otras veces:



Observamos las dos clases generadas, prestando especial atención a la forma en la que ha mapeado la relación 1:N entre categoría y producto. La notación que se emplea indica que la relación se mapea en las dos direcciones, de categoría a producto y desde producto hacia la tabla categoría:

```
@Entity
@Table(name = "categoria")

public class Categoria implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "pkcategoria")
    private Integer pkcategoria;
    @Basic(optional = false)
    @Column(name = "nombre")
    private String nombre;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "akcategoria
    private Collection<Producto> productoCollection;

    public Categoria() {
```

@OneToMany
Una categoría varios productos

```
@Entity
@Table(name = "producto")

public class Producto implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "pkproducto")
    private Integer pkproducto;
    @Basic(optional = false)
    @Column(name = "nombre")
    @Basic(optional = false)
    @Column(name = "pvp")
    private BigDecimal pvp;
    @Column(name = "stock")
    private Integer stock;
    @JoinColumn(name = "akcategoria", referencedColumnName = "pkcategoria")
    @ManyToOne(optional = false)
    private Categoria akcategoria;
```

@JoinColumn
Referencia a columna relacionada
@ManyToOne
Muchos productos en la misma categoría

Ahora definimos las interfaces, una para cada Entidad, extendiendo el CrudRepository de Spring Data:

```java
@Repository
public interface ICategoriaDAO extends CrudRepository<Categoria, Integer> {
    //Se pueden añadri métodos más allá del CrudRepository
}
```

```java
@Repository
public interface IProductoDAO extends CrudRepository<Producto, Integer>{

}
```

También podemos añadir, en la interfaz de cada clase, los métodos específicos o renombrar los que implementa el CrudRepository para practicar.

Por ejemplo:

```java
@Repository
public interface IProductoDAO extends CrudRepository<Producto, Integer>{

    //voy a escribir los métodos que vamos a manejar en el ejercicio

    public Producto findByNombre (String nombre);

    public Producto findByNombreAndPvp (String nombre, double pvp);

//Todos los productos de una categoria
    @Query("select p from Producto p inner join Categoria c on p.akcategoria=c.pkcategoria where c.pkcategoria=:akcategoria")
    public List<Producto> findByCategoria(@Param("akcategoria")int pk);
```

Realizar las pruebas:

Ahora estamos en disposición de probar la funcionalidad.

```java
@SpringBootApplication
public class AlmacenApplication {

    @Autowired
    private ICategoriaDAO categoriadao;

    @Autowired
    private IProductoDAO productodao;

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

    @EventListener({ApplicationReadyEvent.class})
    public void pruebaConsultas() {
        System.out.println("********* LISTA CATEGORIAS ****************");
        List<Categoria> cats = (List<Categoria>) categoriadao.findAll();
        for (Categoria c: cats){
            System.out.println(c.getNombre());
        }
        System.out.println("********* Producto por nombre y precio ****************");
        Producto producto = productodao.findByNombre("lentejas");
        System.out.println("Producto encontrado =  " + producto.toString());

        System.out.println("********* LISTA Productos de una Categoria ****************");

        List<Producto> prods  = productodao.findByCategoria(2001);
        for (Producto p: prods){
            System.out.println(p.getNombre() + " - " + p.getAkcategoria().getNombre());
        }
    }
}
```