ZEPPELINUM

ANTONIO MARTÍNEZ FERNÁDEZ | ADAM MIERZWA

GRUPO 1.2

49973734R | L1TC899Y1

Tabla de contenido

1. Introducción 2
2. Asociaciones JPA 2
   - 2.1 Plato 2
   - 2.2 Restaurante 3
   - 2.3 Usuario 3
   - 2.4 Incidencia 3
   - 2.5 Categoria Restaurante 4
   - 2.6 Categoria 4
3. MongoDB 5
   - 3.1 Pedido 5
     - 3.1.1 ItemPedido 6
     - 3.1.2 Estado pedido 6
     - 3.1.3 Tipo de estado 6
   - 3.2 Dirección 7
   - 3.3 Opinión 8
4. Interfaces 8
5. Ejercicio 1 17
   - 5.1 Añadir Categoria a Restaurante 17
   - 5.2 Editar disponibilidad de plato 18
   - 5.3 Buscar incidencias por usuario e incidencias cerradas 19
6. Estadísticas 20
7. Extras 22

1. Introducción

El proyecto realizado es ZeppelinUm , una aplicación desarrollada en Java , mediante el uso de una base de datos en SQL y otra en MongoDB. Las interfaces se han llevado a cabo mediante el uso de xtml y Java Primefaces , los servidores con Tomcat y WildFly y se ha implementado otro proyecto EJB para realizar el conteo de las estadísticas de dicha aplicación.

2. Asociaciones JPA

2.1 Plato

Hemos considerado que sería interesante tener un control de los platos libres de un restaurante y de todos los platos libres , por este motivo hemos decidido utilizar una relación oneToMany en restaurante bidireccional.

En la base de datos estos será plasmado como un campo restaurante donde tendremos el id del restaurante.

	id	descripcion	disponibilidad	precio	titulo	restaurante
▶	1	a	1	1	plato1	1
	2	a	1	1	a	2
*	NULL	NULL	NULL	NULL	NULL	NULL

```
@ManyToOne
@JoinColumn(name="restaurante")
private Restaurante restaurante;
```

```
@OneToMany(mappedBy = "restaurante", cascade = CascadeType.ALL)
private List<Plato> platos;
```

2.2 Restaurante

En restaurante tenemos que conocer el id del responsable(usuario) de éste pero a su vez el usuario o responsable no deberia tener acceso al restaurante.

Lo hemos modelado como una relación manyToOne unidireccional en restaurante.

En la base de datos obtendremos el campo id en el restaurante.

id	fecha_alta	nombre	num_penalizaciones	num_valoraciones	valoracion_global	responsable
1	2022-12-28	conplato	0	1	7.5	2
2	2022-12-28	RE	0	1	10	1
3	2022-12-28	sinplato	0	2	6.5	3

```
@JoinColumn(name = "responsable")
@ManyToOne
private Usuario responsable;
```

2.3 Usuario

En cuanto a usuario guardamos sus respectivos datos , los campos más relevantes son los de tipo , el cual especifica que funciones podrá realizar o acceder determinado usuario y válidado lo que le permitirá mostrar sus platos en caso de administrar un restaurante.

id	apellidos	clave	email	fecha_nacimiento	nombre	tipo	validado
1	user	user	user	2001-07-01	user	CLIENTE	1
2	conplato	conplato	conplato	2000-01-01	conplato	RESTAURANTE	0
3	sinplato	sinplato	sinplato	2000-01-01	sinplato	RESTAURANTE	0
4	repartidor	12345	veratti@palotes.es	1990-01-08	repartidor	RIDER	0
5	cliente	12345	veratti@palotes.es	1990-01-08	cliente	CLIENTE	1

2.4 Incidencia

Incidencia tiene que estar conectado con un usuario y un restaurante por lo que con dos relaciones ManyToOne desde usuario y restaurante a la vez que una OneToMany desde Incidencia obtenemos una tabla donde restaurante y usuario forman parte con su id de la tabla principal de Incidencias.

id	comentario_cierre	descripcion	fecha_cierre	fecha_creacion	restaurante	usuario
1		description	1990-01-08	1990-01-08	4	6
NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
@OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL)
private List<Incidencia> incidencias;
```

```java
@ManyToOne
@JoinColumn(name="restaurante")
private Restaurante restaurante;

@ManyToOne
@JoinColumn(name="usuario")
private Usuario usuario;
```

```java
@JoinColumn(name="incidencia")
@OneToMany(mappedBy="restaurante",cascade = CascadeType.ALL)
private List<Incidencia> incidencias;
```

2.5 Categoria Restaurante

El modelado ha resultado ser un ManyToMany bidireccional ya que nos interesa obtener la categoría de los restaurantes y a la vez los restaurantes por categoría.

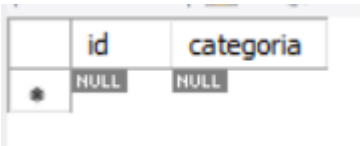
```java
@ManyToMany
@JoinTable(name="restaurante_categoria",

                    joinColumns= {@JoinColumn(name="categoria_restaurante_id")},
                    inverseJoinColumns= {@JoinColumn(name="restaurante_id")}
            )
```

```java
@ManyToMany(mappedBy = "restaurantes")
private List<CategoriaRestaurante> categorias;
```

Estas configuraciones de manyToMany bidireccional nos forma una tabla separada en la base de datos en la cual una columna es el id de la categoría en en la otra aparecería el id del restaurante.

2.6 Categoria

En esta tabla simplemente se almacena el String que contiene el nombre de la categoría y su id correspondiente.



3. MongoDB

3.1 Pedido

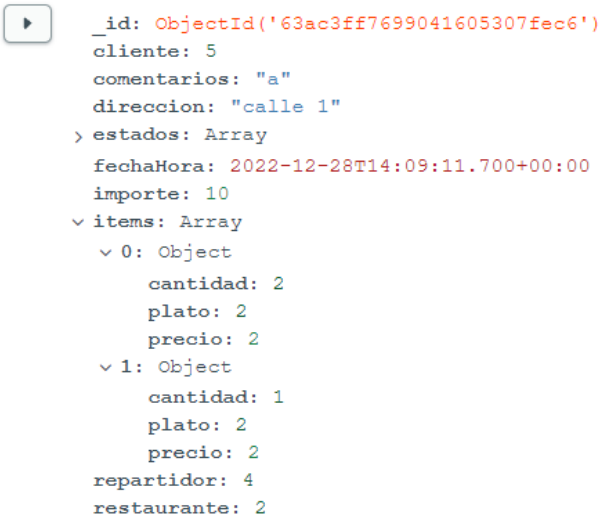
El pedido tiene una estructura algo compleja , ya que esta compuesto de una lista de estadoPedido y otra de items.

Esto se representa simplemente con el atributo lista de las diferentes clases.

```
    @BsonId
private ObjectId id;
    private LocalDateTime fechaHora ;
    private String comentarios;
    private double importe;
    private String direccion;
    private Integer restaurante;
    private Integer repartidor;
    private Integer cliente;
    private List<EstadoPedido> estado;
    private List<ItemPedido> items;
```

Finalmente en la base de datos se representa con valores primitivos en su gran mayoría salvo para la fecha y un array en el cual especificamos los items del pedido , estos a su vez tienen una serie de atributos que mencionaremos posteriormente.

Además en otra lista vemos como se guardan los diferentes estados del pedido.



3.1.1 ItemPedido

ItemPedido a su vez solo guarda la cantidad del item y el precio de éste.

```
private Integer cantidad;
private Double precio;
private Integer plato;
```

3.1.2 Estado pedido

Estado pedido almacena la fecha en la cual se ha cambiado el estado y el tipo de estado al que pertenece.

```
private LocalDateTime fechaEstado;
private TipoEstado estado;
```

3.1.3 Tipo de estado

Lo hemos representado como un enumerado con los diferentes estados que puede tomar un pedido.

```java
public enum TipoEstado {
    INCIADO("INCIADO"),
    ACEPTADO("ACEPTADO"),
    PREPARADO("PREPARADO"),
    RECOGIDO("RECOGIDO"),
    ENTREGADO("ENTREGADO"),
    CANCELADO("CANCELADO"),
    ERROR("ERROR");
```

3.2 Dirección

En dirección mediante el uso de las coordenadas que nos ofrece MongoDb simplemente guardamos también el restaurante al cual esta asociada la dirección.

```json
{
    "_id" : ObjectId("632a2321b1f14659cb35ef97"),
    "calle" : "Juan de Borbón",
    "ciudad" : "Murcia",
    "codigoPostal" : "30110",
    "coordenadas" : {
        "type" : "Point",
        "coordinates" : [
            -1.2083210681136292,
            38.055464804494235
        ]
    },
    "numero" : 5,
    "restaurante" : 9
}
```

En la clase java simplemente usamos objetos del tipo primitvo para su representación.

```
    @BsonId
private ObjectId id;
private Integer restaurante;
private Point coordenadas;
private String calle;
private String codigoPostal;
private String ciudad;
private Integer numero;
```

3.3 Opinión

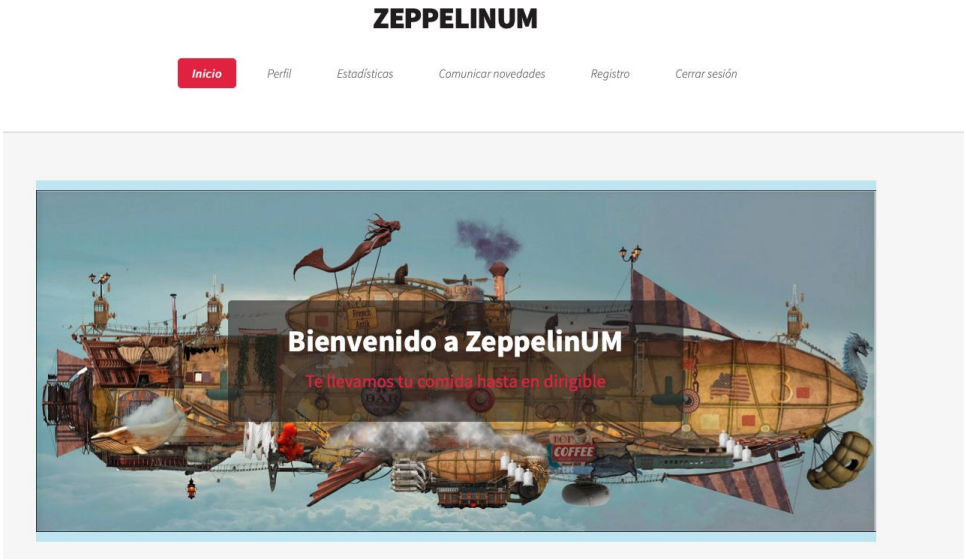
Para la opinión guardamos el id del restaurante y pedido y lo representamos de la siguiente manera:

```
_id: ObjectId('63ac40a7160b4c7a9d35cb6b')
opinion: "Todo estupendo y muy rico"
restaurante: 2
usuario: 1
valor: 10
```

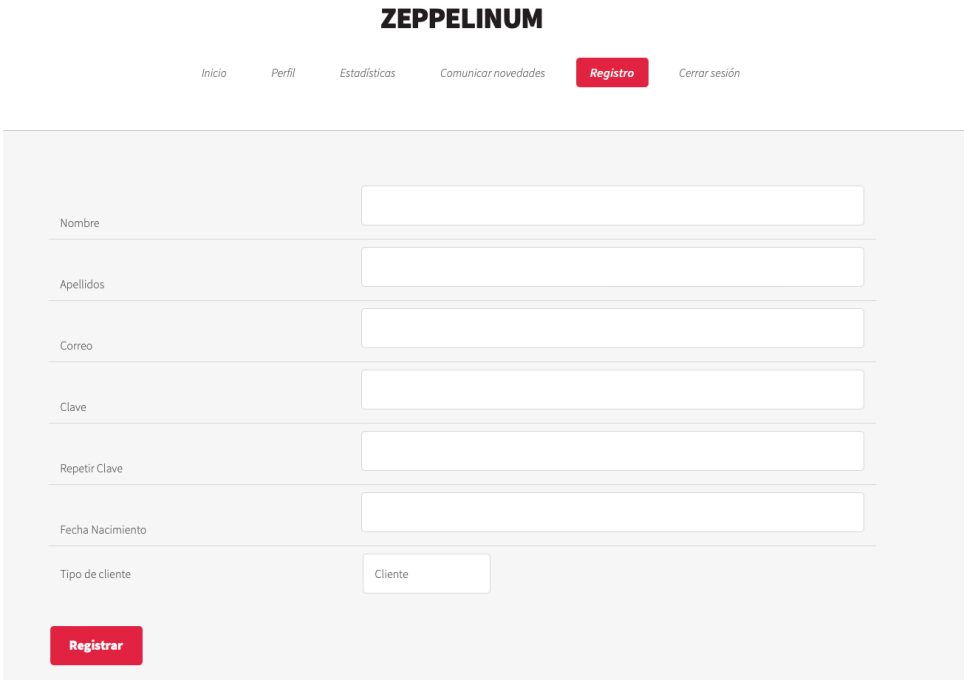
```
    @BsonId
private ObjectId id;
private Integer usuario;
private Integer restaurante;
private String opinion;
private Double valor;
```

4. Interfaces

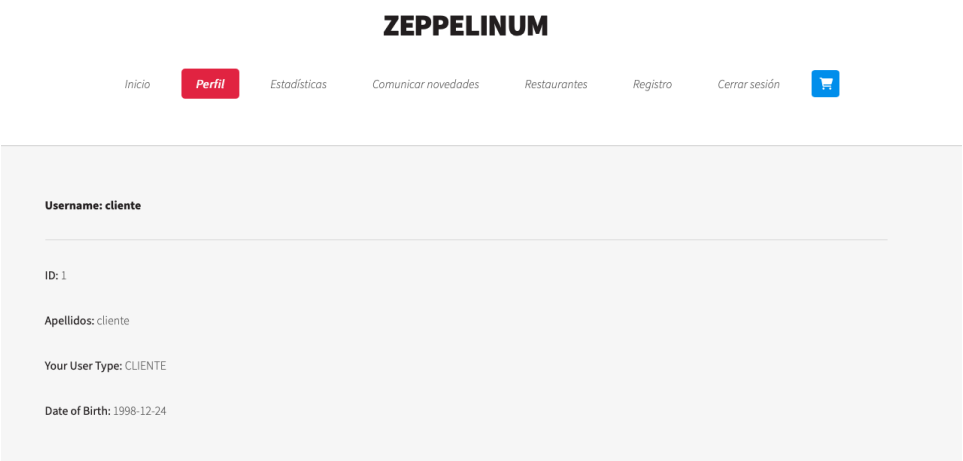
El menú principal se encuentra en el "header" y su uso es básicamente muy intuitivo



En este caso, el usuario sólo tiene la posibilidad de iniciar sesión o crear una cuenta.



El usuario crea aquí una cuenta.

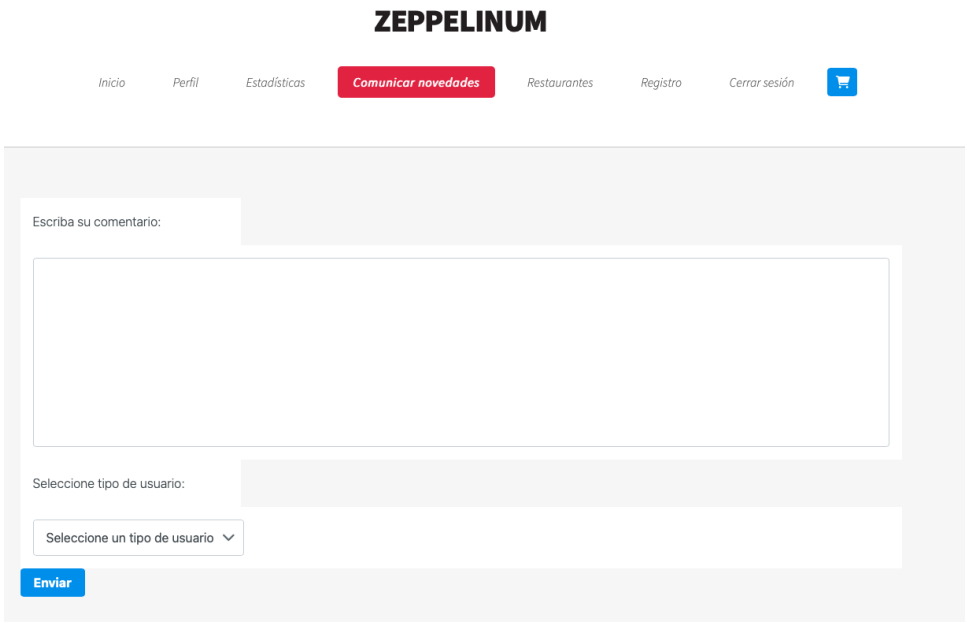


Tras iniciar sesión, el usuario dispone de muchas opciones nuevas

Por ejemplo:

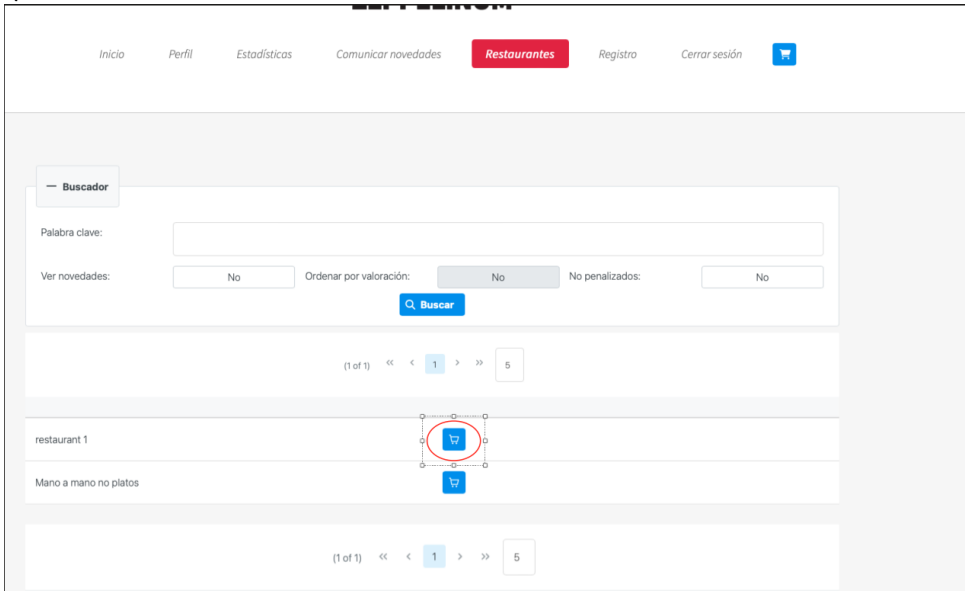
El usuario puede consultar las estadísticas

o enviar un mensaje a los grupos de personas deseados (imagina que tu comida llega tarde y quieres preguntar a los proveedores dónde está).

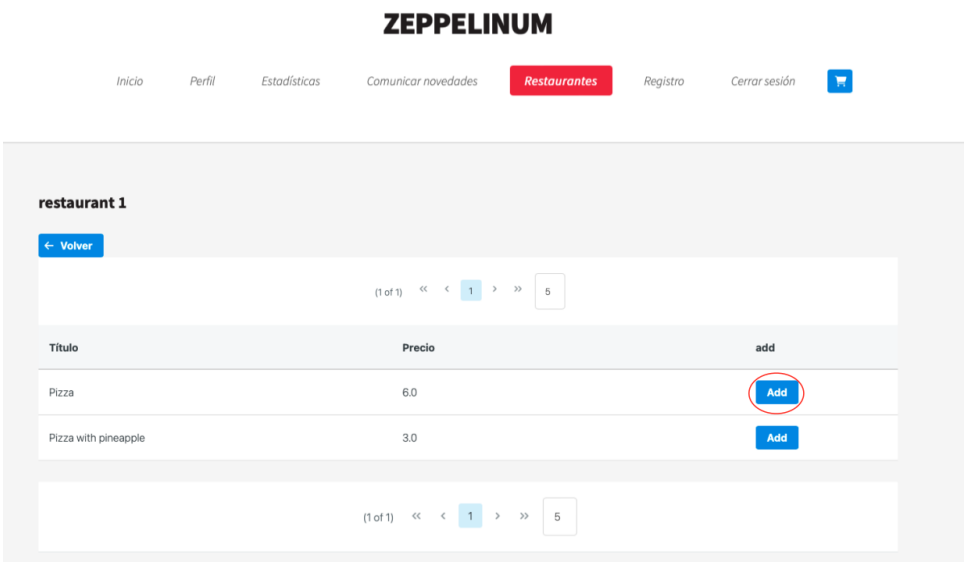


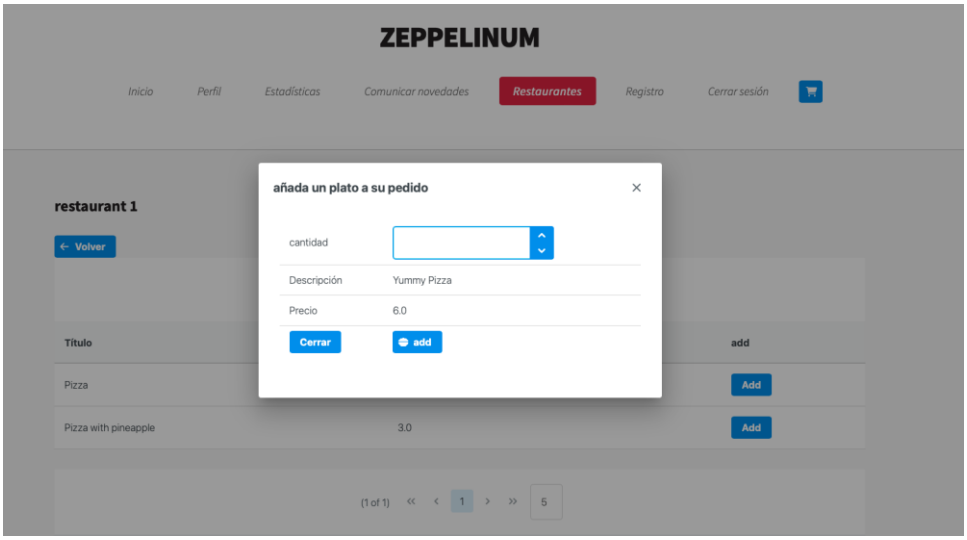
El usuario puede ver los restaurantes y tambien los platos del restaurantes.

(marcado en rojo)



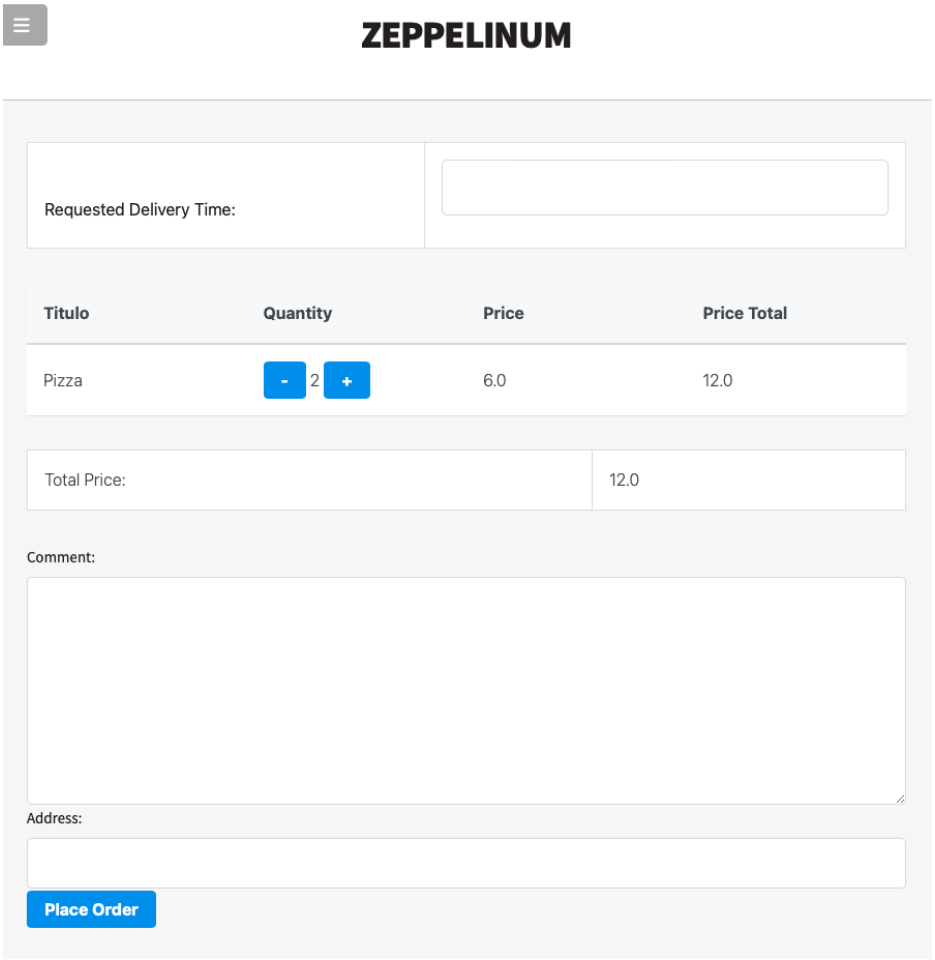
Aquí el usuario puede anadir platos a su pedido





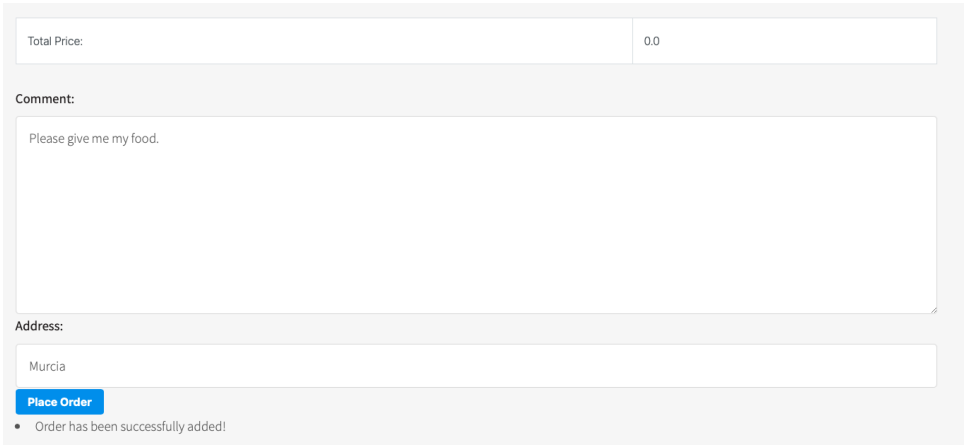
Aquí el usuario puede ver y, si es necesario, realizar su pedido

El campo con la hora de entrega y la dirección no debe estar vacío



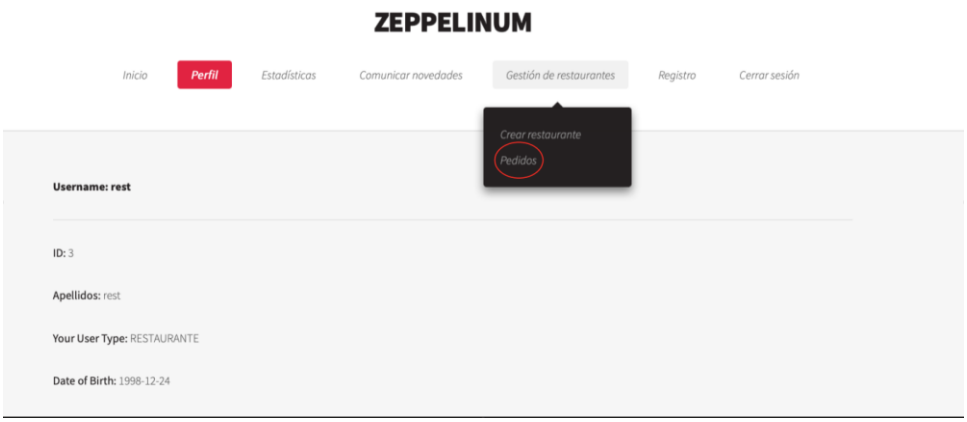
Un texto indica si el pedido se ha realizado correctamente.

Ahora un restaurante puede ver el pedido y aceptarlo o rechazarlo si es necesario.



Conectémonos como restaurante y veamos las novedades.

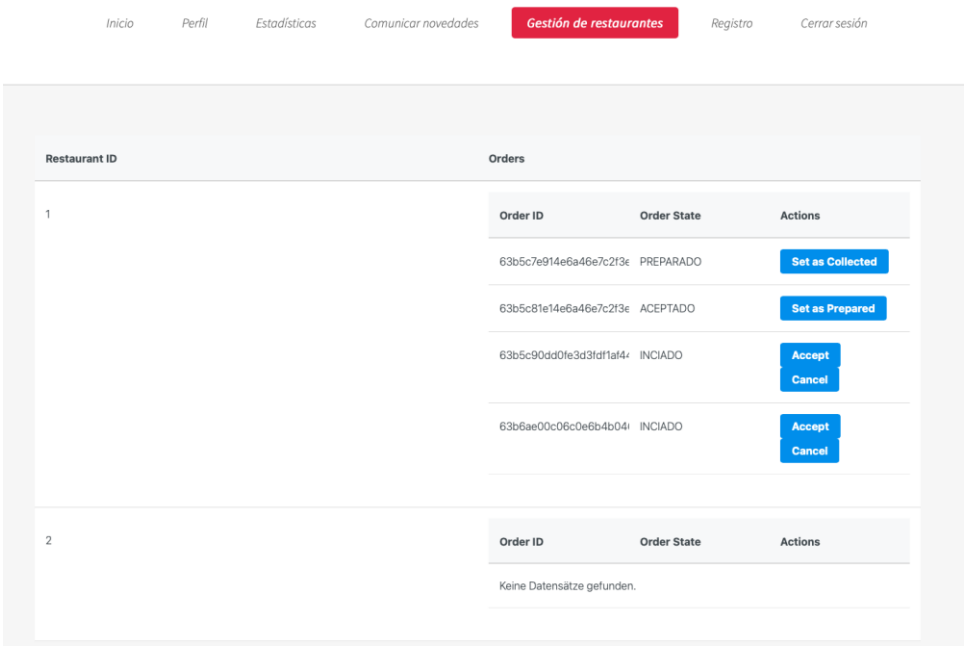
un gerente de restaurante no puede ver los restaurantes y no puede hacer pedidos, pero puede crear restaurantes y ver los pedidos de sus restaurantes existentes.



Aquí vemos que el gerente regenta dos restaurantes.

Sin embargo, sólo el primer restaurante tiene pedidos.

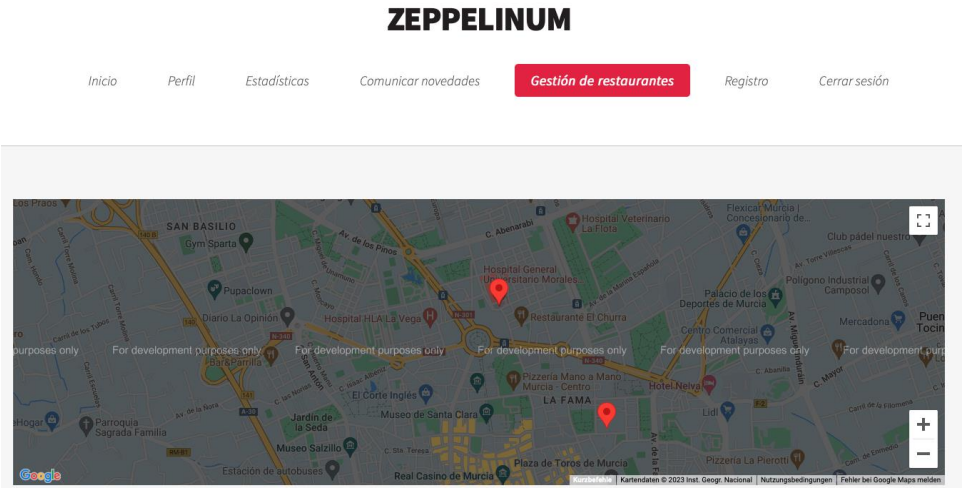
Podemos ver el estado del pedido y realizar diversas acciones.



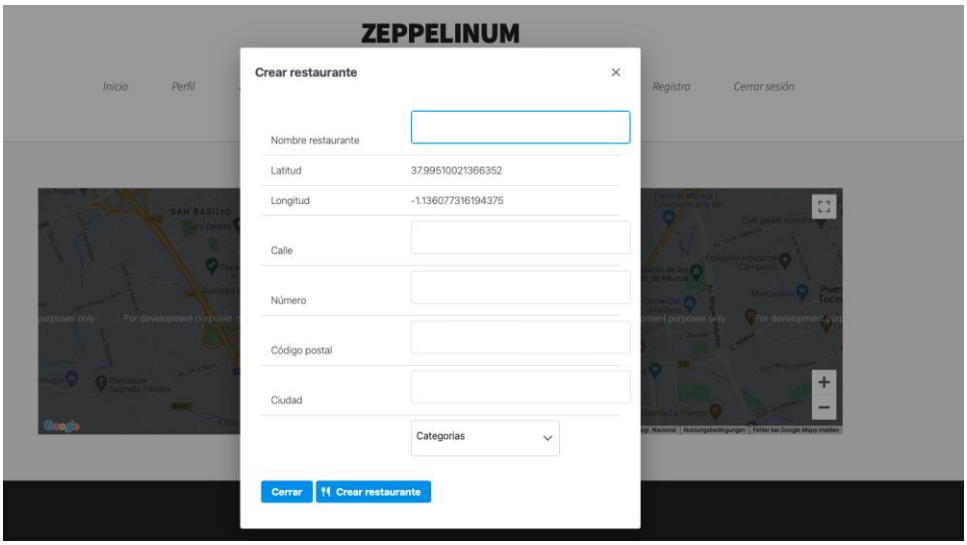
Si decidimos abrir un restaurante, debemos navegar aquí en el menú:

Aquí podemos ver que nuestros dos restaurantes están en el mapa.

Al hacer clic en un marcador, se puede ver el restaurante y añadir platos.



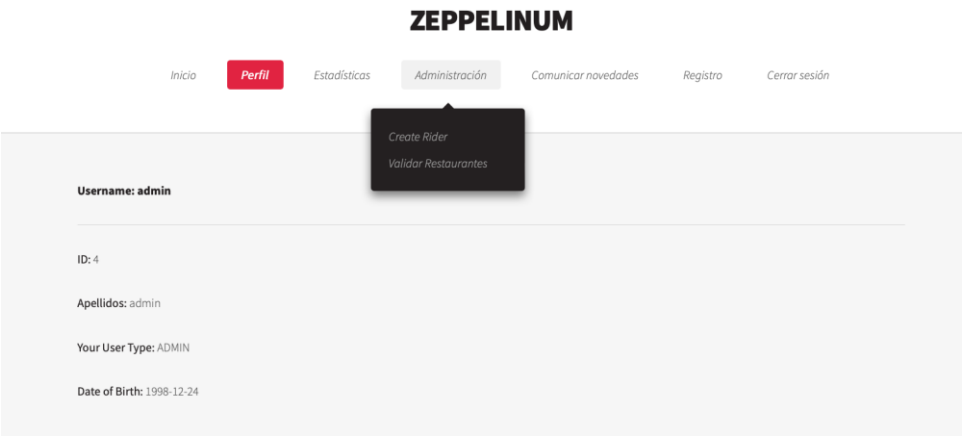
Si queremos añadir un restaurante, podemos hacer clic en cualquier lugar del mapa:



Ahora nos gustaría ver qué puede hacer un administrador que no puedan hacer los demás.

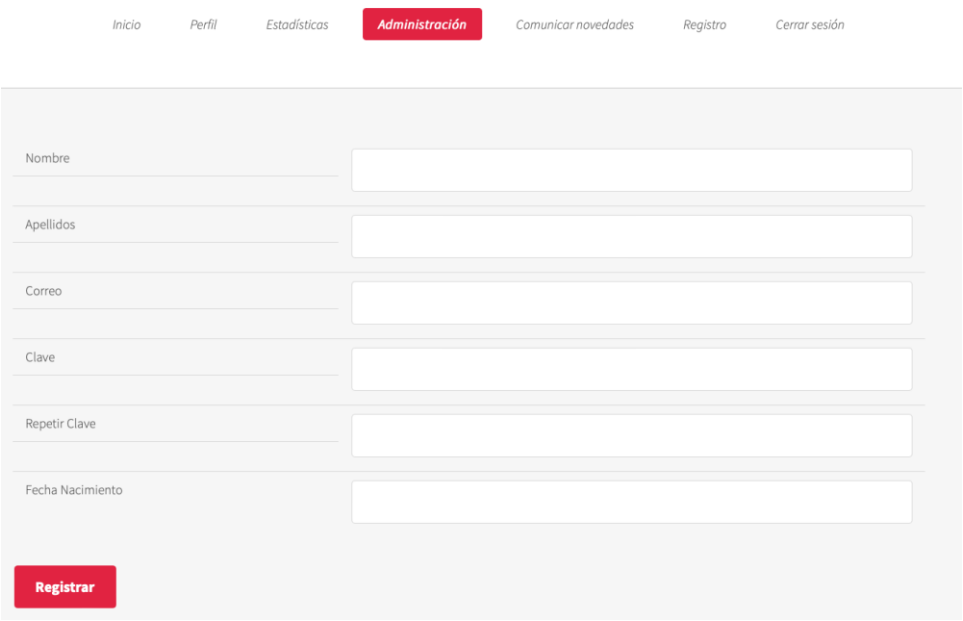
Un administrador puede añadir proveedores y confirmar propietarios de restaurantes.

Si el propietario de un restaurante no está confirmado, no podrá añadir restaurantes a la plataforma.



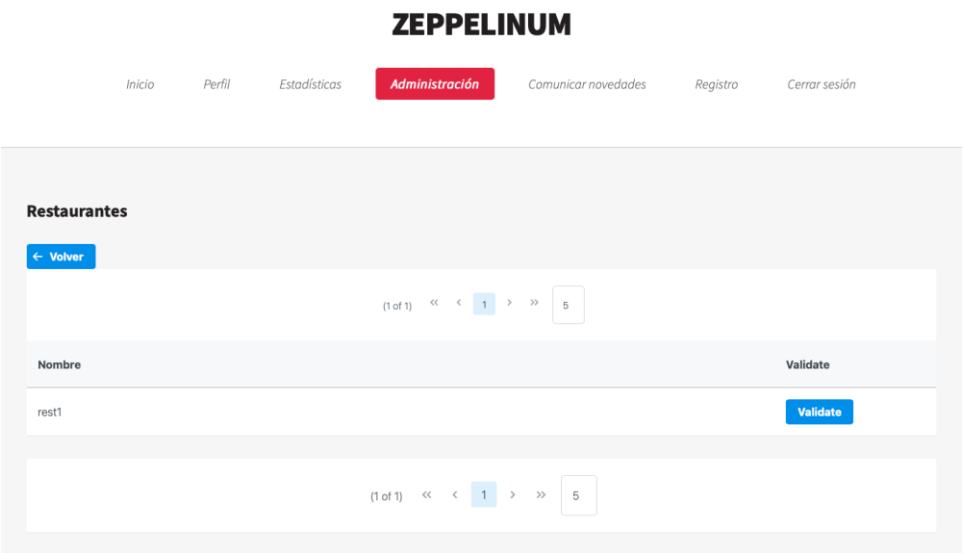
Veamos ahora cómo se creó un proveedor

La interfaz es casi idéntica a la interfaz de registro normal.



Este es el aspecto del menú para validar a los restauradores.

Un clic en el botón azul valida a un propietario de restaurante.



el proveedor no puede hacer nada especial, no llegamos lejos con las tareas de bonificación

5. Ejercicio 1

Los métodos de creación de cada una de las entidades han sido creados en ServicioGestionPlataforma y ServicioGestionPedido y básicamente se basan en crear las entidades en la base de datos correspondiente y devolver el id asignado a dicho objeto.

- Implementa los métodos de creación de cada una de estas entidades.
- Implementa métodos de edición para al menos dos entidades de MySQL. Sugerencias:
  - Editar un restaurante para añadirle una nueva categoría
  - Editar un plato para cambiar su disponibilidad
- Implementa métodos de consulta para las incidencias:
  - Buscar incidencias por usuario
  - Buscar incidencias sin cerrar

5.1 Añadir Categoria a Restaurante

Creamos una transación englobada en un bloque try-catch donde buscamos el id objeto restaurante asociado al id y el de categoría para posteriormente añadir al objeto restaurante en su atributo categorias. Tras esto procedemos a guardar el objeto en la persistencia con el método save() y posteriormente si no ha habido un error durante la transacción se confirma en la base de datos.

```java
public boolean addCategoria(Integer restaurante, Integer categoria) {
        EntityManager em = EntityManagerHelper.getEntityManager();
        try {
                em.getTransaction().begin();

                Restaurante r = RestauranteDAO.getRestauranteDAO().findById(restaurante);
                CategoriaRestaurante cat = CategoriaRestauranteDAO.getCategoriaRestauranteDAO().findById(categoria);
//              System.out.println(r.getNombre());
//              System.out.println(cat);
                LinkedList<CategoriaRestaurante> lista = new LinkedList<>();
                lista.add(cat);

                r.addCategorias(lista);

                CategoriaRestauranteDAO.getCategoriaRestauranteDAO().save(cat, em);
                RestauranteDAO.getRestauranteDAO().save(r, em);

                em.getTransaction().commit();
                return true;

        } catch (Exception e) {
                e.printStackTrace();
                return false;
        } finally {
                if (em.getTransaction().isActive()) {
                        em.getTransaction().rollback();
                }
                em.close();
        }
}
```

5.2 Editar disponibilidad de plato

Para editar la disponibilidad de un plato seguimos el mismo método utilizado anteriormente , obtenemos el objeto plato a partir del id , cambiamos su atributo de disponibilidad y lo guardamos en la base de datos.

```java
public boolean cambiarDisponibilidadPlato(Integer plato, boolean disponibilidad) {

        EntityManager em = EntityManagerHelper.getEntityManager();
        try {
                em.getTransaction().begin();

                Plato p = PlatoDAO.getPlatoDAO().findById(plato);
                p.setDisponibilidad(disponibilidad);

                PlatoDAO.getPlatoDAO().save(p, em);

                em.getTransaction().commit();
                return true;
        } catch (Exception e) {
                e.printStackTrace();
                return false;
        } finally {
                if (em.getTransaction().isActive()) {
                        em.getTransaction().rollback();
                }
                em.close();
        }

}
```

5.3 Buscar incidencias por usuario e incidencias cerradas

Para poder realizar ambos métodos será necesario realizar varias request SQL para obtener tanto las incidencias de un dado usuario y las incidencias cuya fecha de cierre sea NULL, es decir, que aún no se ha cerrado.

```java
@NamedQueries({
        @NamedQuery(name = "Incidencia.findIncidenciaByUser", query = " SELECT i FROM Incidencia i INNER JOIN i.usuario u WHERE u.id = :usuario"),
        @NamedQuery(name = "Incidencia.findIncidenciaNotClosed", query = " SELECT i FROM Incidencia i WHERE i.fechaCierre=null")

})
```

En ambos métodos añadimos los parámetros necesarios a las consultas SQL anteriores y luego devolvemos los objetos en formato DTO.

```java
public List<UsuarioDTO> findIncidenciaByUser(Integer usuario) {
        try {
                Query query = EntityManagerHelper.getEntityManager().createNamedQuery("Incidencia.findIncidenciaByUser");
                query.setParameter("usuario",usuario);
                return transformarToDTO(query.getResultList());
        } catch (RuntimeException re) {
                throw re;
        }
}
```

```
public List<UsuarioDTO> findIncidenciaNotClosed() {
        try {
                Query query = EntityManagerHelper.getEntityManager().createNamedQuery("Incidencia.findIncidenciaNotClosed");
                return transformarToDTO(query.getResultList());
        } catch (RuntimeException re) {
                throw re;
        }
}
```

6. Estadísticas

El primer paso realizado para poder comunicar ambos proyectos (ZeppelinUM y ZeppelinUMEJB) es declarar dos interfaces con los diferentes métodos que contendrá el proyecto EJB. En el proyecto principal se declaran y luego estos son implementados en el proyecto EJB.

```
@Remote
public interface ZeppelinUMRemoto {
    public Integer getNumVisitas(Integer idUsuario);
    public void pedidoIniciado(String pedido);
    public List<EstadisticaOpinionDTO> getEstadisticasOpinion(Integer idUsuario);
        //List<EstadisticaPedidoDTO> getEstadisticasPedido(Integer idRestaurante);
        public int findNumPedidoByUser(int id);
        public int getNumAllPedidos();
        public int findNumPedidoByUserDifferentRestaurant(Integer id);
        public int findPedidosRestaurants(List<Integer> l);
        public int findNumUsersRestaurants(List<Integer> findRestaurantIdByResponsable);
        public int findPedidoByUserDifferentRestaurant(Integer usuario);




}
```

En el ServioGestionPedido o ServicioGestionPlataforma creamos la instancia del proyecto EJB para poder realizar las llamadas desde el proyecto original y en ambas clases mencionadas hacemos uso del zeppelinremoto creado para llamar al proyecto EJB.

```
public static ServicioGestionPedido getServicioGestionPedido() {
    if (servicio == null) {
        try {
            zeppelinumRemoto = (ZeppelinUMRemoto)
InitialContextUtil.getInstance().lookup("ejb:AADD2022/ZeppelinUMMartinezMierzwaEJB/ZeppelinUMRemoto!zeppelinum.ZeppelinUMRemoto");
        } catch (NamingException e) {
            e.printStackTrace();
        }
        servicio = new ServicioGestionPedido();
    }
    return servicio;
}
```

```
public int findPedidoByUserDifferentRestaurant(Integer usuario) {

        return zeppelinumRemoto.findPedidoByUserDifferentRestaurant(usuario);
        //return PedidoDAO.getPedidoDAO().numPedidosDifferentRestaurant(usuario);


}
```

En en proyecto EJB llamamos al método implementado en pedidoDAO dentro del EJB y dentro de este simplemente hacemos la consulta a la base de datos MongoDB.

```
@Override
public int findNumPedidoByUserDifferentRestaurant(Integer id) {
        return pedidoDAO.findNumPedidoByUserDifferentRestaurant(id);


}
```

```
@Lock(LockType.READ)
public int findNumPedidoByUserDifferentRestaurant(Integer id) {

        Bson query_restaurant=Filters.eq("usuario",id);

        return (int) coleccion.countDocuments(query_restaurant);

        int count=0;
        Bson query_restaurant=Filters.eq("usuario",id);
        MongoCursor<String> it=coleccion.distinct("restaurante", query_restaurant,String.class).iterator();

        while (it.hasNext()) {
                count++;
        }
        return count;
}
```

7. Extras

Para la comprobación del correcto uso de todas las funcionalidades contenidas en ServicioGestionPlataforma y ServicioGestionPedido hemos decidido realizar una serie de test para asegurarnos que aún cambiando el proyecto todo seguía funcionando correctamente.

Estos test se pueden encontrar en la carpeta /src/test/java/pruebas en el archivo OwnTest.java.

Los test realizados son unitarios , es decir , no tienen dependencias unos con otros , el orden a ejecutarlos no es importante y son independientes. También hemos rediseñado los test suministrados por la profesora para seguir el esquema anterior y comprobar que toda la funcionalidad sigue siendo correcta en todas las etapas del proyecto.