Aplicación Web usando REST + AngularJS

Autor:

- Francisco Javier Mesa Martín

ÍNDICE

1. Introducción

En el presente documento se abordarán los aspectos de mayor relevancia sobre el proceso de desarrollo e implementación de una SPA para la gestión de usuarios, restaurantes, platos y pedidos basada en Just Eat.

En base a lo anterior, a continuación, se proporciona un vistazo a la aplicación como producto final y a los detalles más importantes que deben ser considerados a la hora de evaluar la consecución de los requisitos mínimos impuestos, funcionales y no funcionales, y de las características adicionales introducidas.

A lo largo de este documento aparecerán aspectos clave de diseño marcados en **rojo**.

2. Estructura y organización de ficheros

Antes de entrar en detalle con la implementación de los distintos requisitos de la aplicación, se ha considerado oportuno explicar desde un inicio cómo se han estructurado y organizado los ficheros fuente dentro del proyecto desarrollado. Esto con el objetivo de tener ubicados los recursos cuando sean referenciados posteriormente en las siguientes secciones.

Así, en primer lugar se muestra la organización de los ficheros dentro del directorio src/main/java en Java Resources:

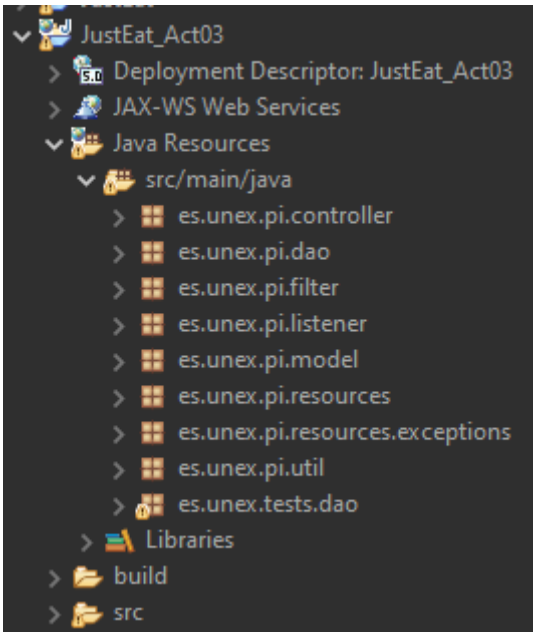


Figura 1. Carpeta Java Resources.

Dentro de esta carpeta se encuentran localizados los siguiente directorios:

- **es.unex.pi.controller**: contiene los servlets que se encargan del registro de usuario, del inicio y cierre de sesión.

- **es.unex.pi.dao**: contiene las clases Java que definen las interfaces para interactuar con la base de datos y sus implementaciones.
- **es.unex.pi.filter**: contiene los filtros que son disparados al realizar ciertas peticiones al servidor.
- **es.unex.pi.listener**: contiene el listener que se utiliza para inicializar la conexión a la base de datos cuando se despliega la aplicación.
- **es.unex.pi.model**: contiene las clases que implementan la lógica de negocio más básica de la aplicación.
- **es.unex.pi.resources**: contiene las clases que se encargan de implementar la API para acceder a los diferentes recursos de la aplicación.
- **es.unex.pi.resources.exceptions**: contiene dos clases auxiliares utilizadas para controlar excepciones a la hora de recuperar los datos de la API.
- **es.unex.pi.util**: contiene dos clases auxiliares empleadas a lo largo de la implementación.
- **es.unex.tests.dao**: contiene las pruebas que verifican el correcto funcionamiento de las implementaciones de las interfaces definidas en las clases Java del paquete *es.unex.pi.dao*.

  Por otro lado, dentro de la carpeta webapp se tiene lo siguiente:

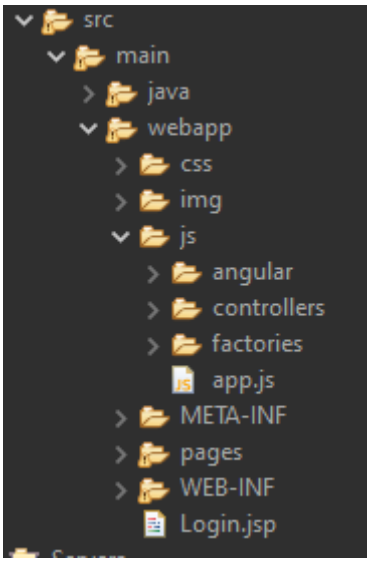


Figura 2. Carpeta WebApp.

- **css**: contiene todas las hojas de estilo que son aplicadas a los documentos HTML generados por los JSP. Como se puede observar, dichas hojas de estilo se han organizado en carpetas en función de la entidad cuya información estilan. En la carpeta denominada general se han incluido aquellos ficheros CSS que han sido reutilizados.
- **img**: contiene todas las imágenes decorativas que se han usado en la aplicación.
- **js**: contiene los ficheros encargados de gestionar la aplicación para AngularJS. Los ficheros son almacenados en varias carpetas, las cuales se distribuyen en diferentes tipos.
    - **angular**: contiene los ficheros angular.js y angular-route.js. El primer fichero contiene la implementación del framework de angular, el cual proporciona las

funcionalidades básicas y esenciales para definir una web en una sola página. El segundo fichero es una extensión adicional de js que proporciona un enrutador básico dentro de la aplicación el cual permite cargar diferentes vista y controladores en función de la URL actual.
    - **controllers**: contiene los diferentes controladores de la aplicación, los cuales son utilizados para definir y administrar el comportamiento de una parte específica de la interfaz de usuario. Sirven como intermediarios entre la vista y el modelo.
    - **factories**: contiene los ficheros encargados de crear o devolver objetos o funciones reutilizables en toda la aplicación.
    - **app.js**: es un fichero utilizado para definir y configurar el módulo principal de la aplicación. Este archivo contiene la configuración global, la definición de rutas y la inicialización del módulo de la aplicación.
- **pages**: contiene las diferentes páginas HTML, las cuales utilizarán los diferentes servicios de la carpeta de angular para llevar a cabo la muestra de información y las diferentes peticiones de los usuarios.
- **WEB-INF**: además del fichero web.xml, contiene todos los ficheros JSP a los que se despachan las peticiones. De forma paralela a los ficheros CSS, estos ficheros han sido organizados en carpetas en función de la entidad a la que representan. De forma adicional, en la carpeta general se ha incluido un fichero HTML que es reutilizado en múltiples JSP.
- **login.jsp**: fichero JSP que actúa como archivo de bienvenida, es decir, define la página que se muestra al usuario cuando este inicia la aplicación.

3. Requisitos mínimos

Vista la organización de ficheros, a continuación se explica cómo se han alcanzado los requisitos mínimos establecidos.

3. 1. Requisitos no funcionales

3.1.1. Validez de HTML y CSS

- **HTML**: todos los archivos HTML generados por los JSP han sido comprobados en el servicio de validación de lenguajes de marcado del W3C, resultando en todos ellos válidos. Por otro lado, el archivo footer.html, localizado en WEB-INF/general, no valida ya que se trata de un fragmento de código HTML que es reutilizado en diversos JSP.
- **CSS**: todas las hojas de estilo empleadas en la aplicación han sido comprobadas en el servicio de validación de CSS del W3C. Todas ellas han resultado válidas.

3. 1. 2. Seguridad

En cuanto a seguridad, se han tenido en cuenta diversos aspectos que deben ser comentados.

En primer lugar, se ha cifrado la comunicación entre el cliente y el servidor a través de la configuración de un **certificado SSL**. El objetivo de este es asegurar que la información compartida está cifrada para terceros.

```
<!-- SSL config -->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Entire Application</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Figura 3. Definición certificado SSL dentro del web.xml.

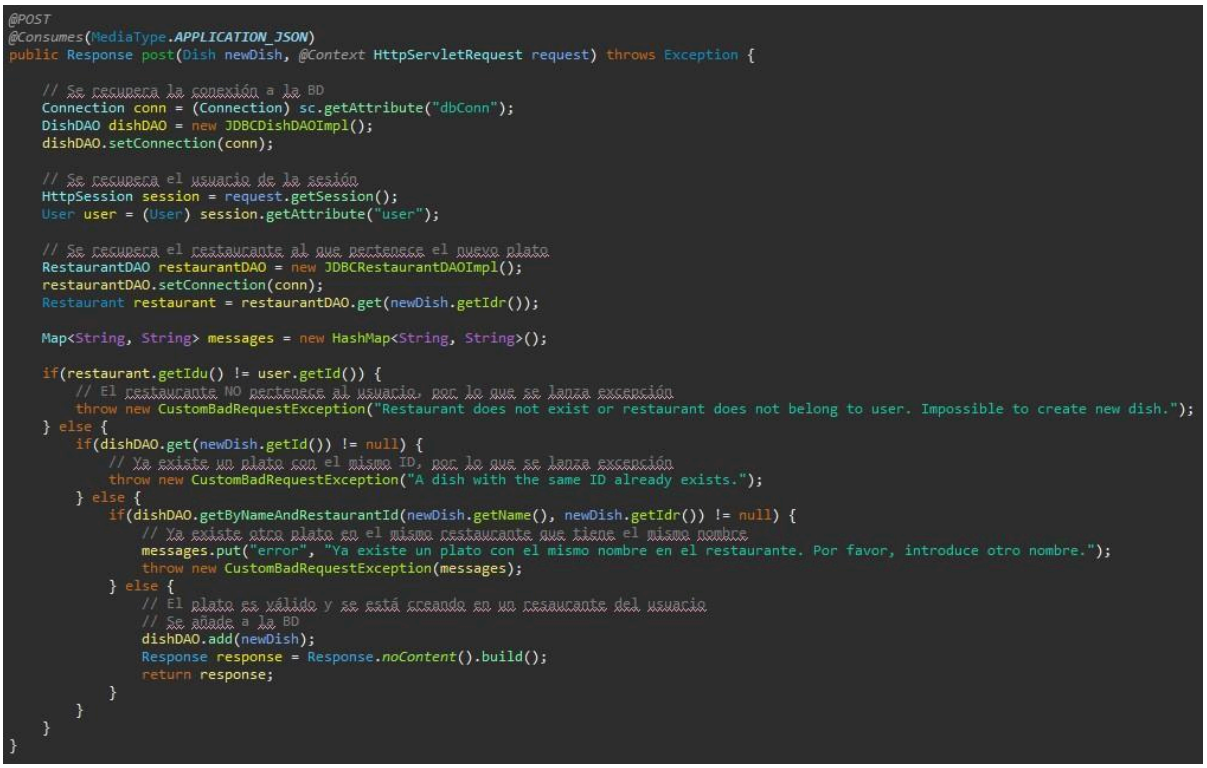
```
<Connector SSLEnabled="true" maxThreads="150" port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol">
    <UpgradeProtocol className="org.apache.coyote.http2.Http2Protocol"/>
    <SSLHostConfig>
        <Certificate certificateKeystoreFile="${user.home}/.keystore" certificateKeystorePassword="FranContrasena10" type="RSA"/>
    </SSLHostConfig>
</Connector>
```

Figura 4. Definición certificado SSL dentro del server.xml.

En segundo lugar, se ha evitado que los usuarios registrados en la aplicación puedan **ver, editar o eliminar información privada asociada a otros usuarios**, ya sea a través de escribir / modificar las URLs o empleando herramientas de desarrollo del navegador. Concretamente, se han controlado las siguientes situaciones:

- **Restaurantes**:
    - No se puede editar el restaurante de otro usuario.
    - No se puede borrar el restaurante de otro usuario.
- **Platos**:
    - No se puede crear un plato en el restaurante de otro usuario.
    - No se puede editar un plato de un restaurante de otro usuario.
    - No se puede borrar un plato de un restaurante de otro usuario.
- **Pedidos**:
    - No se puede ver la información de un pedido que ha realizado otro usuario.

Las anteriores medidas de seguridad son controladas a través de los diferentes servicios proporcionados por la API de cada recursos. Un ejemplo de medida de seguridad en los recursos de la API es el siguiente:



```java
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response post(Dish newDish, @Context HttpServletRequest request) throws Exception {

    // Se recupera la conexión a la BD
    Connection conn = (Connection) sc.getAttribute("dbConn");
    DishDAO dishDAO = new JDBCDishDAOImpl();
    dishDAO.setConnection(conn);

    // Se recupera el usuario de la sesión
    HttpSession session = request.getSession();
    User user = (User) session.getAttribute("user");

    // Se recupera el restaurante al que pertenece el nuevo plato
    RestaurantDAO restaurantDAO = new JDBCRestaurantDAOImpl();
    restaurantDAO.setConnection(conn);
    Restaurant restaurant = restaurantDAO.get(newDish.getIdr());

    Map<String, String> messages = new HashMap<String, String>();

    if(restaurant.getIdu() != user.getId()) {
        // El restaurante NO pertenece al usuario, por lo que se lanza excepción
        throw new CustomBadRequestException("Restaurant does not exist or restaurant does not belong to user. Impossible to create new dish.");
    } else {
        if(dishDAO.get(newDish.getId()) != null) {
            // Ya existe un plato con el mismo ID, por lo que se lanza excepción
            throw new CustomBadRequestException("A dish with the same ID already exists.");
        } else {
            if(dishDAO.getByNameAndRestaurantId(newDish.getName(), newDish.getIdr()) != null) {
                // Ya existe otro plato en el mismo restaurante que tiene el mismo nombre
                messages.put("error", "Ya existe un plato con el mismo nombre en el restaurante. Por favor, introduce otro nombre.");
                throw new CustomBadRequestException(messages);
            } else {
                // El plato es válido y se está creando en un resaurante del usuario
                // Se añade a la BD
                dishDAO.add(newDish);
                Response response = Response.noContent().build();
                return response;
            }
        }
    }
}
```

Figura 5. Ejemplo de medida de seguridad en los ficheros de recursos de la API.

Como se puede observar, se comprueba si el ID del dueño del restaurante y el usuario que ha logueado coinciden, si existe algún plato en el restaurante con el mismo identificador, y si existe algún plato en el restaurante que tiene el mismo nombre, para controlar platos llamados de la misma manera. En caso de no cumplirse alguna de estas condiciones, se lanzarán diferentes excepciones que mostrarán diversos mensajes según la condición que no haya sido cumplida.

El resto de situaciones directamente no son posibles debido a implicaciones de la gestión de la sesión, puesto que para acceder a cualquier funcionalidad de la aplicación el usuario debe haber iniciado sesión (el filtro LoginFilter.java se asegurará de esto). Tener la sesión iniciada desde un principio evita situaciones como las siguientes:

- **Restaurantes**:
    - No se puede crear un restaurante en nombre de otro usuario.
- **Pedidos**:
    - No se puede añadir un plato al pedido de otro usuario.
    - No se puede eliminar un plato del pedido de otro usuario.
    - No se puede realizar un pedido en nombre de otro usuario.
- **Valoraciones**:
    - No se puede realizar una valoración en nombre de otro usuario.
- **Usuarios**:
    - No se puede editar la información de otro usuario.
    - No se puede eliminar el perfil de otro usuario.
    - No se puede ver el perfil de otro usuario.

3. 1. 3. Buenas prácticas

En cuanto a buenas prácticas, cabe destacar los dos siguientes puntos:

- **Patrón PRG**: como ya se ha comentado en el apartado anterior, se ha aplicado este patrón para evitar que peticiones no idempotentes generen efectos adversos en el estado de la base de datos.
- **Reutilización de código**: se ha buscado reutilizar el máximo código posible sin que esto supusiera una complejidad excesiva en la lectura e interpretación del código. De esta forma, en webapp/pages/general se tiene el código HTML reutilizado. Este último se corresponde con el pie de página que se incluye en varias páginas de la aplicación, así como las páginas HTML que contienen los mensajes de borrado. Por otro lado, en webapp/css/general se tienen varias hojas de estilo que son aplicadas a fragmentos de código HTML similares a lo largo de la aplicación. Adicionalmente, también se han reutilizado ficheros de tipo *factories* para la creación/edición de usuarios, restaurantes, platos, pedidos y reseñas.

3. 2. Requisitos funcionales

En los siguientes requisitos funcionales, las opciones relativas a Editar, Eliminar o Crear elementos, sólo estarán visibles en pantalla si el usuario que ha iniciado sesión tiene autorización para ello. Por ejemplo, solo el usuario que ha creado un restaurante tendrá habilitada la opción para crear un plato dentro del restaurante. El resto de usuarios no verán dichas opciones.

3. 2. 1. Registro y acceso de usuarios

Como se ha diseñado la aplicación para comenzar por el inicio de sesión, esta será la primera funcionalidad con la que podrá interactuar el usuario. Así, la implementación se ha llevado a cabo de la siguiente forma:

- El usuario introduce sus credenciales (email y contraseña) en el formulario de inicio de sesión. Al pulsar en iniciar sesión, se llama el método doPost de LoginServlet.do.
    - **En LoginServlet.do**:
        - **doGet**: despacha la petición al JSP que muestra el formulario de inicio de sesión (Login.jsp).
        - **doPost**: comprueba si las credenciales introducidas en el formulario de inicio de sesión (Login.jsp) son correctas. En caso afirmativo, se almacena el usuario en la sesión y se redirige a la página principal (index.html). En caso contrario, se añade un error indicando que el usuario no existe y se despacha junto a la petición a Login.jsp. Adicionalmente, si un usuario que ya ha iniciado sesión introduce de nuevo credenciales en el anterior formulario (presionando “back”, por ejemplo), la sesión actual será destruida. Es decir, volver al inicio de sesión e introducir nuevas credenciales (válidas o no) supone el cierre de la sesión anterior.
- Si el usuario no dispone de cuenta, puede pulsar uno de los enlaces en Login.jsp a SignUpServlet.do para efectuar su registro.
    - **En SignUpServlet.do**:

        - **doGet**: despacha la petición al JSP que muestra el formulario de registro (EditUser.jsp). Este JSP es reutilizado para la edición de los datos del usuario.
        - **doPost**: al igual que sucedía en el inicio de sesión, si un usuario que ya ha iniciado sesión vuelve al registro (con “back”) e introduce nuevos datos (válidos o no), se elimina la sesión actual. Por otra parte, se comprueba que los datos introducidos sean válidos. Para ello, en la clase User del modelo se han incluido métodos para validar el nombre, el apellido, la contraseña y el email. Todas estas validaciones son llamadas en un solo método denominado validateUser(), también definido dentro de la clase User. Este método realiza TODAS las validaciones y devuelve verdadero o falso en función de si se cumplen todas o no. Se ejecutan todas las validaciones para mostrar al usuario, en caso de ser necesario, todos los errores en los datos introducidos. En caso de que no haya ningún error en los datos, se comprueba que no exista ya un usuario con el mismo email, es decir, no puede haber dos usuarios con el mismo email en el sistema. Si lo hay, se despacha al mismo formulario con el error. En caso contrario, se añade el usuario a la base de datos, a la sesión y se redirige a la página principal.
- Una vez iniciada sesión, el usuario puede acceder a su perfil desde “Perfil” en las cabeceras de las páginas. Se trata de un enlace el cual estará gestionado por el controlador userProfileCtrl.js, y utilizará las funciones definidas en las factorías usersFactory.js, restaurantsFactory.js y ordersFactory.js para recibir sus datos.
    - **En userProfileCtrl.js**:
        - **userProfileViewModel**: almacena la referencia del controlador.
        - **userProfileViewModel.user**: almacena los datos del usuario que se obtendrán a partir de la API.
        - **userProfileViewModel.restaurants**: almacena los datos de los restaurantes creados por el usuario.
        - **userProfileViewModel.orders**: almacena la información del historial de pedidos realizados por el usuario.
        - **userProfileViewModel.functions**: define las funciones para operar sobre el perfil del usuario. Estas funciones son las siguientes:
            - **readUser**: obtiene la información del usuario que ha realizado el inicio de sesión, es decir, sus datos personales, sus restaurantes y su historial de pedidos, y almacena los datos sobre las variables definidas anteriormente.
            - **readUserRestaurants**: obtiene la información de los restaurantes creados por los usuarios.
            - **readUserOrders**: obtiene los datos del historial de pedidos del usuario.
    - **En userFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del usuario.
        - **usersInterface**: define los métodos para operar sobre los datos del usuario. Para obtener los datos del usuario utiliza el método:
            - **getUser**: obtiene la información del usuario de la API.

    - **En restaurantsFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API de los restaurantes.
        - **restaurantsInterface**: define los métodos para operar sobre los datos de los restaurantes. Para obtener los datos de los restaurantes utiliza el método:
            - **getUserRestaurants**: obtiene la información de los restaurantes creados por el usuario desde la API.
    - **En ordersFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API de los pedidos.
        - **ordersInterface**: define los métodos para operar sobre los datos de los pedidos. Para obtener los datos de los pedidos utiliza el método:
            - **getUserOrders**: obtiene la información del historial de pedidos del usuario desde la API.

- Dentro del perfil, el usuario puede pulsar en “Editar perfil”. Esta acción está gestionada por el controlador editUserCtrl.js, y utilizará la factoría userFactory.js para operar con los datos.
    - **En editUserCtrl.js**:
        - **editUserViewModel**: almacena la referencia del controlador.
        - **editUserViewModel.user**: almacena los datos del usuario que se obtendrán a partir de la API.
        - **editUserViewModel.validationErrors**: almacena los diferentes errores que se pueden dar al operar con los datos del usuario.
    - **En userFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del usuario.
        - **usersInterface**: define los métodos para operar sobre los datos del usuario. Para obtener los datos del usuario utiliza el método:
            - **getUser**: obtiene la información del usuario de la API.
            - **updateUser**: modificar la información del usuario sobre la base de datos.

- Dentro del perfil el usuario también puede eliminar su cuenta pulsando en “Eliminar cuenta”. El controlador deleteConfirmationHandlerCtrl.js se encargará de ello, utilizando también los métodos del userFactory.js.
    - **En deleteConfirmationHandlerCtrl.js**:
        - **deleteConfirmationHandlerViewModel**: almacena la referencia del controlador.
        - **deleteConfirmationHandlerViewModel.user**: almacena los datos del usuario que se obtendrán a partir de la API.
        - **deleteConfirmationHandlerViewModel.functions**: define los métodos para eliminar cualquier recursos de la aplicación. Para eliminar el usuario se utilizará el siguiente método:
            - **deleteUser**: elimina un usuario de la base de datos. El borrado del usuario supone el borrado de todos sus restaurantes, valoraciones y pedidos.

- **En userFactory.js**:
  - **url**: almacena la URL donde se encuentran los recursos de la API del usuario.
  - **usersInterface**: define los métodos para operar sobre los datos del usuario. Para obtener los datos del usuario utiliza el método:
    - **deleteUser**: elimina un usuario de la base de datos.

- Por último, desde el perfil el usuario podrá cerrar sesión. Esto llamará a **LogoutServlet.do**. El doGet de este servlet simplemente borra al usuario de la sesión y redirige al inicio de sesión.

3. 2. 2. Gestión de restaurantes

Desde el perfil de usuario se podrá crear un restaurante. El controlador restaurantHandlerCtrl.js será el encargado de realizar las operaciones necesarias para la creación de un restaurante, estas operaciones realizarán uso de las métodos de restaurantsFactory.js y los recursos de la API:

- **En restaurantHandlerCtrl.js**:
  - **restaurantHandlerViewModel**: almacena la referencia del controlador.
  - **restaurantHandlerViewModel.restaurant**: almacena los datos del restaurante que se obtendrán a partir de la API.
  - **restaurantHandlerViewModel.functions**: define los métodos para operar con los datos de cualquier restaurante de la aplicación. Para crear un restaurante se utilizará el siguiente método:
    - **createRestaurant**: crea un restaurante con los datos determinados por el usuario, y la lista de categorías seleccionadas.
- **En restaurantsFactory.js**:
  - **url**: almacena la URL donde se encuentran los recursos de la API del restaurante.
  - **restaurantsInterface**: define los métodos para operar sobre los datos del restaurante. Para crear un restaurante utiliza el método:
    - **postRestaurant**: inserta un nuevo restaurante en la base de datos, así como la lista de categorías asociadas al mismo.
- **En RestuarantResources.java**: en la API del restaurante se han teniendo en cuenta las siguientes medidas a la hora de crear un restaurante:
  - Se ha decidido que no sea posible establecer la valoración media de un restaurante, por lo que se inicializa a 0. Del mismo modo, se ha añadido un nuevo atributo, la localidad.
  - No puede haber dos restaurantes con el mismo nombre o email.
  - Como mínimo, se debe seleccionar una categoría para el restaurante.
  - El rango de precio inferior no puede tener un valor mayor que el rango superior.

Para la validación se han creado, al igual que con los usuarios, los métodos pertinentes en la clase Restaurant del modelo.

Una vez creado el restaurante, este aparecerá listado en el perfil del usuario. Pinchando en él se podrá acceder a toda su información. Esto lo gestiona el controlador restaurantProfileCtrl.js el cual utilizará las factorías restaurantFactory.js y userFactory.js.

- **En restaurantProfileCtrl.js**:
  - **restaurantProfileViewModel**: almacena la referencia del controlador.
  - **restaurantProfileViewModel.restaurant**: almacena los datos del restaurante que se obtendrán a partir de la API.
  - **restaurantProfileViewModel.currentUser**: almacena la información del usuario logueado.
  - **restaurantProfileViewModel.categories**: almacena la información de las categorías del restaurante.
  - **restaurantProfileViewModel.dishes**: almacena la información de los platos del restaurante.
  - **restaurantProfileViewModel.reviews**: almacena las reseñas del restaurante.
  - **restaurantProfileViewModel.reviewed**: bandera booleana que indica si el restaurante ha obtenido una reseña o no. Inicialmente a false.
  - **restaurantProfileViewModel.dishesAmounts**: almacena la cantidad de pedidos de un plato.
  - **restaurantProfileViewModel.defaultAmount**: variable que indica la cantidad de pedido por defecto de un plato. Inicialmente 1.
  - **restaurantProfileViewModel.relatedRestaurantsByCategory**: almacena la información de los restaurantes relacionados por categoría.
  - **restaurantProfileViewModel.relatedRestaurantsByCity**: almacena la información de los restaurantes relacionados por ciudad.
  - **restaurantProfileViewModel.relatedRestaurantsByPrice**: almacena la información de los restaurantes relacionados por precio.
  - **restaurantProfileViewModel.relatedRestaurantsByGrad**: almacena la información de los restaurantes relacionados por valoración.
  - **restaurantProfileViewModel.functions**: define los métodos para operar con los datos de cualquier restaurante de la aplicación. Para obtener los datos de un restaurante se utilizarán los siguientes métodos:
    - **readRestaurant**: crea un restaurante con los datos determinados por el usuario, y la lista de categorías seleccionadas.
    - **readCurrentUser**: obtiene la información del usuario logueado.
    - **readRestaurantCategories**: obtiene la información de las categorías asociadas al restaurante.
    - **readRestaurantReviews**: obtiene la información de las reseñas del restaurante.
    - **userInReviews**: obtiene la información de los usuarios que han realizado una reseña.
    - **readRelatedRestaurantsByCategory**: obtiene la información de los restaurantes relacionados por categoría.

                - **readRelatedRestaurantsByCity**: obtiene la información de los restaurantes relacionados por ciudad.
                - **readRelatedRestaurantsByPrice**: obtiene la información de los restaurantes relacionados por precio.
                - **readRelatedRestaurantsByGrade**: obtiene la información de los restaurantes relacionados por valoración.
    - **En restaurantsFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del restaurante.
        - **restaurantsInterface**: define los métodos para operar sobre los datos del restaurante. Para obtener los datos de un restaurante utiliza el método:
            - **getRestaurant**: obtiene la información de un restaurante desde la API.
            - **getRestaurantCategories**: obtiene la información de las categorías de un restaurante desde la API.
            - **getRestaurantDishes**: obtiene la información de los platos de un restaurante desde la API.
            - **getRestaurantReviews**: obtiene la información de las reseñas de un restaurante desde la API.
            - **getRelatedRestaurantsByCategory**: obtiene la información de los restaurantes asociados por categoría desde la API.
            - **getRelatedRestaurantsByCity**: obtiene la información de los restaurantes asociados por ciudad desde la API.
            - **getRelatedRestaurantsByPrice**: obtiene la información de los restaurantes asociados por precio desde la API.
            - **getRelatedRestaurantsByGrade**: obtiene la información de los restaurantes asociados por valoración desde la API.
    - **En usersFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del usuario.
        - **usersInterface**: define los métodos para operar sobre los datos del usuario. Para obtener un usuario utiliza el método:
            - **getUser**: obtiene los datos del usuario de la API.

Dentro del perfil del restaurante se puede editar su información pulsando en “Editar este restaurante”. Esto es controlado por restaurantHandlerCtrl, el cual utiliza los métodos de restaurantsFactory.js.

- **En restaurantHandlerCtrl.js**:
    - **restaurantHandlerViewModel**: almacena la referencia del controlador.
    - **restaurantHandlerViewModel.restaurant**: almacena los datos del restaurante que se obtendrán a partir de la API.
    - **restaurantHandlerViewModel.functions**: define los métodos para operar con los datos de cualquier restaurante de la aplicación. Para crear un restaurante se utilizará el siguiente método:

            - **updateRestaurant**: modifica la información de un restaurante con los nuevos datos determinados por el usuario, así como la lista de categorías seleccionadas.
    - **En restaurantsFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del restaurante.
        - **restaurantsInterface**: define los métodos para operar sobre los datos del restaurante. Para modificar un restaurante utiliza el método:
            - **putRestaurant**: modifica un restaurante en la base de datos, así como la lista de categorías asociadas al mismo.

Por último, dentro del propio perfil del restaurante también se puede proceder a su borrado pulsando en "Eliminar este restaurante". El controlador deleteConfirmationHandlerCtrl.js se encargará de ello, utilizando también los métodos del restaurantsFactory.js.

- **En deleteConfirmationHandlerCtrl.js**:
    - **deleteConfirmationHandlerViewModel**: almacena la referencia del controlador.
    - **deleteConfirmationHandlerViewModel.restaurants**: almacena los datos del restaurante que se obtendrán a partir de la API.
    - **deleteConfirmationHandlerViewModel.functions**: define los métodos para eliminar cualquier recurso de la aplicación. Para eliminar un restaurante se utilizará el siguiente método:
        - **deleteRestaurant**: elimina un restaurante de la base de datos. Tras el borrado se redirigirá al perfil de usuario. El borrado de un restaurante supone el borrado de todos sus platos y valoraciones, pero no pedidos en los aparezcan algunos de sus platos.
- **En restaurantsFactory.js**:
    - **url**: almacena la URL donde se encuentran los recursos de la API del restaurante.
    - **restaurantsInterface**: define los métodos para operar sobre los datos del restaurante. Para eliminar un restaurante utiliza el método:
        - **deleteRestaurant**: elimina un restaurante en la base de datos, así como la lista de categorías asociadas al mismo.

3. 2. 3. Gestión de platos de un restaurante

Desde el perfil de uno de sus restaurantes el usuario podrá crear un nuevo plato para el restaurante en cuestión. Para ello, deberá pulsar en "Añadir nuevo plato" en la sección "Menú" (en esta sección aparecerán listados todos los platos del restaurante). Para realizar esta acción se utilizará el controlador dishHandlerCtrl.js y las factorías dishesFactory.js, restaurantsFactory.js y usersFactory.js.

- **En dishHandlerCtrl.js**:
    - **dishHandlerViewModel**: almacena la referencia del controlador.
    - **dishHandlerViewModel.restaurant**: almacena los datos del restaurante que se obtendrán a partir de la API.

        - **dishHandlerViewModel.user**: almacena los datos del usuario que se obtendrán a partir de la API.
        - **dishHandlerViewModel.functions**: define los métodos para operar con los datos de un plato. Para crear un plato se utilizarán los siguientes métodos:
            - **createDish**: crea un nuevo plato en la base de datos. Se ha considerado que un mismo restaurante no puede tener dos platos con el mismo nombre.
            - **readUser**: obtiene la información del usuario que se ha logueado. Sirve para controlar que un usuario no pueda crear un plato sobre un restaurante que no sea suyo.
            - **readRestaurant**: obtiene la información del restaurante. Sirve para controlar que un usuario no pueda crear un plato sobre un restaurante que no sea suyo.
    - **En dishesFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del plato.
        - **dishesInterface**: define los métodos para operar sobre los datos del plato. Para crear un plato utiliza el método:
            - **postDish**: crea un nuevo plato en la base de datos.
    - **En usersFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del usuario.
        - **usersInterface**: define los métodos para operar sobre los datos del usuario. Para obtener un usuario utiliza el método:
            - **getUser**: obtiene un usuario desde la API.
    - **En restaurantsFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del restaurante.
        - **restaurantsInterface**: define los métodos para operar sobre los datos del restaurante. Para crear un restaurante utiliza el método:
            - **getRestaurant**: obtiene un restaurante desde la API.

En caso de querer editar un plato existente, desde el perfil del restaurante se puede buscar el plato en la sección de “Menú” y pulsar en “Editar plato”. Para realizar esta acción se utilizará el controlador dishHandlerCtrl.js y la factoría dishesFactory.js.

- **En dishHandlerCtrl.js**:
    - **dishHandlerViewModel**: almacena la referencia del controlador.
    - **dishHandlerViewModel.functions**: define los métodos para operar con los datos de un plato. Para crear un plato se utilizarán los siguientes métodos:
        - **updateDish**: actualiza la información de un plato.
- **En dishesFactory.js**:
    - **url**: almacena la URL donde se encuentran los recursos de la API del plato.
    - **dishesInterface**: define los métodos para operar sobre los datos del plato. Para modificar un plato utiliza el método:

- **putDish**: modifica un nuevo plato en la base de datos.

En último lugar, si el usuario desea eliminar un plato, deberá pulsar en “Eliminar plato”. Esta opción se encuentra justo a la derecha de la anterior. Para realizar esta acción se utilizará el controlador deleteConfirmationHandlerCtrl.js y la factoría dishesFactory.js.

- **En deleteConfirmationHandlerCtrl.js**:
  - **deleteConfirmationHandlerViewModel**: almacena la referencia del controlador.
  - **deleteConfirmationHandlerViewModel.restaurants**: almacena los datos del restaurante que se obtendrán a partir de la API.
  - **deleteConfirmationHandlerViewModel.functions**: define los métodos para eliminar cualquier recurso de la aplicación. Para eliminar un plato se utilizará el siguiente método:
    - **deleteDish**: elimina un plato de la base de datos. El borrado de un plato no supone ningún cambio en cascada en la base de datos.
- **En dishesFactory.js**:
  - **url**: almacena la URL donde se encuentran los recursos de la API del plato.
  - **dishesInterface**: define los métodos para operar sobre los datos del plato. Para eliminar un plato utiliza el método:
    - **deleteDish**: elimina un nuevo plato en la base de datos.

3. 2. 4. Gestión de pedidos

Un usuario tendrá en todo momento un apartado en su perfil para acceder a su pedido actual. A través de este apartado, denominado “Mi pedido”, el controlador orderHandlerCtrl.js gestiona la creación de pedidos.

- **En orderHandlerCtrl.js**:
  - **orderHandlerViewModel**: almacena la referencia del controlador.
  - **orderHandlerViewModel.functions**: define los métodos para operar con los datos de un pedido. Para crear un pedido se utiliza el siguiente método:
    - **submitOrder**: crea un nuevo pedido para el usuario logueado.
- **En ordersFactory.js**:
  - **url**: almacena la URL donde se encuentran los recursos de la API del pedido.
  - **ordersInterface**: define los métodos para operar sobre los datos del pedido. Para crear un pedido utiliza el método:
    - **postOrder**: crea un nuevo pedido sobre la base de datos en la base de datos.

Para añadir un plato sobre el pedido creado anteriormente se debe utilizar el controlador restaurantProfileCtrl.js y la factoría ordersFactory.js.

- **En restaurantProfileCtrl.js**:
  - **restaurantProfileViewModel**: almacena la referencia del controlador.

        - **restaurantProfileViewModel.functions**: define los métodos para operar con los datos de un restaurante. Para añadir un plato al pedido se utiliza el siguiente método:
            - **addDishToOrder**: añade un plato al pedido creado por el usuario logueado. En un mismo pedido se pueden añadir platos de diferentes restaurantes. En la lista de platos del restaurante se tendrá, por cada plato, la opción de "Añadir a pedido" siempre y cuando el restaurante acepte pedidos. En caso contrario, la opción no aparecerá. Antes de añadir al pedido, el usuario tiene la opción de indicar la cantidad de unidades que quiere del plato.
    - **En ordersFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del pedido.
        - **ordersInterface**: define los métodos para operar sobre los datos del pedido. Para añadir plato al pedido se utiliza el método:
            - **postDishToOrder**: se encarga de crear un pedido realizando una llamada a la API.

Para poder eliminar un plato del pedido será necesario utilizar el controlador deleteConfirmationHandlerCtrl.js y la factoría ordersFactory.js.

- **En deleteConfirmationHandlerCtrl.js**:
    - **deleteConfirmationHandlerViewModel**: almacena la referencia del controlador.
    - **deleteConfirmationHandlerViewModel.functions**: define los métodos para eliminar cualquier recurso de la aplicación. Para eliminar un plato del pedido se utilizará el siguiente método:
        - **deleteDishFromOrder**: elimina un plato del pedido creado por el usuario.
- **En ordersFactory.js**:
    - **url**: almacena la URL donde se encuentran los recursos de la API del pedido.
    - **ordersInterface**: define los métodos para operar sobre los datos del pedido. Para eliminar un plato del pedido se utiliza el método:
        - **postDishToOrder**: se encarga de eliminar un plato del pedido realizando una llamada a la API.

Por último, realizado un pedido, el usuario podrá visualizar en la última sección de su perfil una lista con todos los pedidos que ha realizado. Esto simplemente mostrará un código de pedido, el precio total y un enlace para ver sus detalles. Todas estas acciones son gestionadas por los controladores orderHandlerCtrl.js y userProfileCtrl.js.

- **En orderHandlerCtrl.js**:
    - **orderHandlerViewModel**: almacena la referencia del controlador.
    - **orderHandlerViewModel.totalBill**: almacena la información del precio total de un pedido. Inicialmente 0.
    - **orderHandlerViewModel.functions**: define los métodos para operar con los datos de un pedido. Para ver los datos de un pedido de un

usuario y calcular el precio total del mismo se utilizarán los siguiente métodos:

- **readOrder**: obtiene la información de un pedido realizado por el usuario.
- **readOrderDishes**: obtiene la información de los platos de un pedido realizado por el usuario.
- **readCurrentOrderDishes**: obtiene la información de los platos del pedido actual del usuario:
- **calculateTotalBill**: realiza el cálculo del total del pedido de un pedido del usuario.

- **En ordersFactory.js**:
  - **url**: almacena la URL donde se encuentran los recursos de la API del pedido.
  - **ordersInterface**: define los métodos para operar sobre los datos del pedido. Para obtener los datos de un pedido se utilizan los métodos:
    - **getOrder**: obtiene la información de un pedido desde la API.
    - **getOrderDishes**: obtiene la información de los platos de un pedido desde la API.
    - **getCurrentOrderDishes**: obtiene la información de los platos del pedido actual desde la API.
- **En userProfileCtrl.js**:
  - **userProfileViewModel**: almacena la referencia del controlador.
  - **userProfileViewModel.orders**:: almacena la información del historial de pedidos realizados por un usuario.
  - **userProfileViewModel.functions**: define los métodos para operar con los datos de un usuario. Para ver los datos del historial de pedidos de un usuario se utilizará el siguiente método:
    - **readUserOrders**: obtiene la información del historial de pedidos realizados por el usuario. En caso de que algunos de los platos del pedido hayan sido borrados, simplemente se mostrará como que el plato ha sido eliminado, pero se seguirá mostrando el número de unidades pedidas y el precio total.
- **En usersFactory.js**:
  - **url**: almacena la URL donde se encuentran los recursos de la API del usuario.
  - **userInterface**: define los métodos para operar sobre los datos del pedido. Para obtener los datos de un pedido se utilizan los métodos:
    - **getUserOrders**: obtiene la información del historial de pedidos de un usuario desde la API.

3. 2. 5. Valoración

Las valoraciones o reseñas se muestran en el perfil del restaurante, justo a la derecha de la información básica de este. El controlador restaurantProfileCtrl.js es el encargado de determinar si un restaurante dispone de reseñas, y mostrar la información de estas junto con los usuarios que hayan indicado dichas reseñas.

- **En restaurantProfileCtrl.js**:

        - **restaurantProfileViewModel**: almacena la referencia del controlador.
        - **restaurantProfileViewModel.reviews**: almacena la información de las reseñas realizadas sobre un restaurante.
        - **restaurantProfileViewModel.reviewed**: bandera booleana que indica si un restaurante dispone de reseña o no. Inicialmente en false.
        - **restaurantProfileViewModel.functions**: define los métodos para operar con los datos de un restaurante. Para ver y gestionar las reseñas realizadas sobre un restaurante se utilizarán los siguientes métodos:
            - **readRestaurantReviews**: obtiene la información de las reseñas realizadas sobre el restaurante.
            - **userInReviews**: obtiene la información de los usuarios que han realizado un reseña sobre el restaurante.
    - **En restaurantsFactory.js**:
        - **url**: almacena la URL donde se encuentran los recursos de la API del restaurante.
        - **restaurantsInterface**: define los métodos para operar sobre los datos del pedido. Para obtener los datos de un pedido se utilizan los métodos:
            - **getRestaurantReviews**: obtiene la información de las reseñas de un restaurante desde la API.

Un usuario podrá añadir una valoración o reseña a un restaurante siempre y cuando no lo haya hecho antes. De esta forma, si el usuario en cuestión no ha realizado ninguna, le aparecerá la opción de “Añadir reseña”. Para poder crear una nueva reseña se utilizará el controlar insertReviewCtrl.js.

    - **En insertReviewCtrl.js**:
        - **insertReviewViewModel**: almacena la referencia del controlador.
        - **insertReviewViewModel.review**: almacena la información de una reseña.
        - **insertReviewViewModel.restaurant**: almacena la información del restaurante que ha recibido una reseña.
        - **insertReviewViewModel.user**: almacena la información del usuario que ha realizado una reseña
        - **insertReviewViewModel.alreadyReviewed**: bandera booleana que indica si un restaurante dispone de reseña o no. Inicialmente en false.
        - **insertReviewViewModel.functions**: define los métodos para operar con los datos de una reseña. Para crear una reseña sobre un restaurante se utilizarán los siguientes métodos:
            - **readUser**: obtiene la información del usuario que va ha realizar una reseña. Sirve para controlar qué usuario ha realizado una reseña.
            - **readRestaurant**: obtiene la información del restaurante que va a recibir una reseña.
            - **readIfRestaurantAlreadyReviewed**: indica si un restaurante ha recibido una reseña o no.
            - **createReview**: crea una nueva reseña para un restaurante.

- **En reviewsFactory.js**:
  - **url**: almacena la URL donde se encuentran los recursos de la API del restaurante.
  - **reviewsInterface**: define los métodos para operar sobre los datos del reseña. Para crear una reseña se utilizan los métodos:
    - **postReview**: crea una nueva reseña en la base de datos.
    - **getIfRestaurantReviewed**: obtiene la información desde la API sobre si un restaurante ha recibido una reseña o no.

Desde la página principal, index.html, los resultados de las búsquedas por texto (pulsar en Buscar Restaurante, independientemente de haber introducido una cadena de texto o no) podrán ordenarse por valoración (de mayor a menor o de menor a mayor).

3. 2. 6. Buscar restaurante

La búsqueda de restaurantes se realiza a través de dos controladores diferentes. Uno de ellos será el encargado de llevar a cabo la búsqueda mediante diferentes tipos de filtros, como son la disponibilidad de pedidos, el orden de valoración, y la cadena de texto introducida sobre la barra de búsqueda, la cual puede ir vacía. Por otro lado, el otro filtro se corresponde con el listado de categorías que aparece en la parte inferior de la página principal index.html.

Para poder realizar la búsqueda poder categorías o por filtro se ha utilizado el controlador searchAndCategoriesCtrl.js.

- **En searchAndCategoriesCtrl.js**:
  - **searchAndCategoriesViewModel**: almacena la referencia del controlador.
  - **searchAndCategoriesViewModel.searchText**: almacena la información con la cadena de texto introducida sobre la barra de búsqueda.
  - **searchAndCategoriesViewModel.categories**: almacena el listado de categorías de restaurantes de la aplicación.
  - **searchAndCategoriesViewModel.available**: almacena si se desea buscar restaurantes disponibles, no disponibles o ambos.
  - **searchAndCategoriesViewModel.order**: almacena el orden en el que se desea mostrar los restaurantes, es decir, de menor a mayor o de mayor a menor valoración media.
  - **searchAndCategoriesViewModel.functions**: define los métodos para operar con los datos de una búsqueda. Para realizar una búsqueda se utilizarán los siguientes métodos:
    - **readCategories**: obtiene la información de todas las categorías de restaurantes de la aplicación.
    - **submitForm**: realiza la búsqueda según los parámetros de filtro indicados por el usuario, en él se especifican una cadena de texto que puede ser opcional, la disponibilidad del restaurante, y el orden de valoración.
- **En categoriesFactory.js**:

        - **url**: almacena la URL donde se encuentran los recursos de la API de las categorías.
        - **categoriesInterface**: define los métodos para operar sobre los datos de una categoría. Para obtener todas las categorías se utiliza el siguiente método:
            - **getAllCategories**: obtiene desde la API el listado de todas las categorías de restaurantes de la aplicación.

Para poder obtener los listados de las búsquedas realizadas anteriormente se ha utilizado el controlador searchHandlerCtrl.js, el cual utilizará las factorías restaurantsFactory.js y categoriesFactory.js, para obtener dicha información.

- **En searchHandlerCtrl.js**:
    - **searchHandlerViewModel**: almacena la referencia del controlador.
    - **searchHandlerViewModel.category**: almacena la información de una categoría.
    - **searchHandlerViewModel.restaurantsByCategory**: almacena la información de todos los restaurantes que se corresponden con una categoría.
    - **searchHandlerViewModel.restaurantsByFilter**: almacena la información de todos los restaurantes que se corresponden con una consulta de filtrado.
    - **searchHandlerViewModel.functions**: define los métodos para operar con los datos de una búsqueda. Para obtener el listado de restaurantes de una búsqueda se utilizarán los siguientes métodos:
        - **readCategory**: obtiene la información de una categoría.
        - **readRestaurantsByCategory**: obtiene el listado de restaurantes correspondientes a una categoría determinada.
        - **readRestaurantsByFilter**: obtiene el listado de restaurantes correspondientes a una consulta según los parámetros utilizados sobre el filtro de búsqueda.
- **En restaurantsFactory.js**:
    - **url**: almacena la URL donde se encuentran los recursos de la API del restaurante.
    - **restaurantsInterface**: define los métodos para operar sobre los datos del restaurante. Para realizar una búsqueda a través de una categoría o un filtro se utilizan los métodos:
        - **getRestaurantsByCategory**: obtiene desde la API todos los restaurantes que pertenecen a una categoría.
        - **getRestaurantsByFilter**: obtiene desde la API todos los restaurantes que cumplen con la consulta del filtro.
- **En categoriesFactory.js**:
    - **url**: almacena la URL donde se encuentran los recursos de la API de las categorías.
    - **categoriesInterface**: define los métodos para operar sobre los datos de una categoría. Para obtener una categoría se utiliza el siguiente método:

- **getCategory**: obtiene la información de una categoría desde la API.

Para la realización de búsquedas se han considerado los siguientes aspectos clave:

- **Búsqueda a través de la barra de búsqueda**: Con esta primera opción, el usuario puede filtrar los resultados por disponibilidad y ordenarlos por valoración. Además, se podrá pulsar en “Buscar restaurantes” incluyendo o no una cadena de texto en la barra de búsqueda. Se tienen así, a su vez, dos opciones:
    - **No se incluye una cadena de texto**: se recuperan todos los restaurantes existentes con la disponibilidad seleccionada (por defecto se recuperan todos los restaurantes, tanto disponibles como no disponibles) y ordenados por valoración (por defecto se ordenan de mayor a menor valoración).
    - **Se incluye una cadena de texto**: se recuperan todos los restaurantes existentes con la disponibilidad seleccionada y ordenados por valoración, pero aplicando un filtro adicional que consiste en la cadena de texto introducida. Se ha implementado la búsqueda de tal forma que dicha cadena de texto se pueda corresponder con los tres siguientes conceptos:
        - **Ciudad o dirección**: tolerancia al fallo de 3 caracteres o inclusión de cadena. 13
        - **Nombre de restaurante**: tolerancia al fallo de 3 caracteres o inclusión de cadena.
        - **Descripción del restaurante**: tolerancia al fallo de 7 caracteres.

  Así, en el resultado de la búsqueda se mostrarán tres listas de restaurantes coincidentes por cada uno de los tres conceptos indicados. Además, como se ha indicado, se ha tenido en cuenta una pequeña tolerancia al fallo en la cadena introducida. Es decir, no es necesario que el usuario escriba una ciudad, dirección, nombre o descripción exacta. Por ejemplo, si escribe la ciudad “Cácres”, en la lista de restaurantes coincidentes por ciudad aparecerán todos aquellos cuya localidad sea “Cáceres”.

  Esta tolerancia se ha conseguido a través de la clase **LevenshteinSearch**, que cuenta con un método público y estático que calcula la diferencia entre dos cadenas. En el caso ejemplificado, la diferencia entre “Cácres” y “Cáceres” es de 1, ya que solo sería necesario añadir a la primera palabra un carácter para coincidir con la segunda.

  Todo este proceso de filtrado es llevado a cabo a través del fichero RestaurantResources donde se obtienen los datos de la API de los restaurantes. Un aspecto importante es que si el usuario modificase la URL generada con la búsqueda y, por ejemplo, alterase los valores para los filtros empleados (disponibilidad u orden por valoración) con valores erróneos, el servicio dirigiría de nuevo a la página principal de búsqueda.

- **Búsqueda a través de categorías**: al seleccionar una de las categorías mostradas se recuperarán todos los restaurantes con dicha categoría. En este segundo tipo de búsqueda no se ha incluido el filtro por disponibilidad ni orden por valoración. De mostrar los resultados se encarga nuevamente el fichero RestaurantResources

donde se obtienen los datos de la API de los restaurantes, pero esta vez desde otro método diferente al anterior.

3. 2. 7. Estado del restaurante

Tal y como se ha comentado en apartados anteriores, al crear y editar un restaurante se puede indicar su disponibilidad, es decir, si acepta pedidos (valor 1) o no (valor 0). Este valor es empleado a la hora de realizar las búsquedas que se han explicado en el apartado anterior. Así, el usuario podrá seleccionar si los restaurantes que devuelva la búsqueda aceptan pedidos (pasando el valor 1 a la consulta), no los aceptan (pasando el valor 0 a la consulta) o todos (se pasa el valor -1 para consultar los restaurantes sin tener en cuenta la disponibilidad). Todas estas condiciones son llevadas a cabo sobre el siguiente método de RestaurantResource.java:

```
@GET
@Path("/search")
@Produces(MediaType.APPLICATION_JSON)
public Map<String, List<Restaurant>> getRestaurantsByFilterJSON(@QueryParam("searchText") String searchText,
                                                                @QueryParam("available") @DefaultValue("-1") String available,
                                                                @QueryParam("order") @DefaultValue("0") String order,
                                                                 @Context HttpServletRequest request) {
```

Figura 6. Método para realizar la búsqueda de restaurantes.

4. Requisitos extra

4. 1. Restaurantes relacionados

Tal y como se ha comentado en el apartado de Gestión de restaurantes, en el perfil del restaurante además de su información básica, sus valoraciones (reseñas) y los platos que oferta, también se ha añadido una sección con los restaurantes relacionados. Esta sección está compuesta por cuatro listas de restaurantes:

- **Relacionados por categorías**: restaurantes que al menos tienen asignados una de las categorías del restaurante original.
- **Relacionados por ciudad (localidad)**: restaurantes cuya localidad coincida con el restaurante original.
- **Relacionados por rango de precios**: restaurantes cuyos valores para el rango superior e inferior varían únicamente en una unidad respecto a los valores del rango del restaurante original. Por ejemplo, si el restaurante original tiene como precio mínimo 5 € y como máximo 40 €, aparecerá como restaurante relacionado por rango de precio cualquier otro restaurante con precio mínimo de 4 o 6€ y precio máximo de 39 o 41€ (ambas condiciones deben darse simultáneamente, por lo que no valdría un restaurante que tiene de mínimo 4 €, pero de máximo 60€).
- **Relacionados por valoración media**: restaurantes cuya valoración media no supone una diferencia mayor a 0,5 respecto a la valoración media del restaurante original. Es decir, si el restaurante original tiene 4 puntos de valoración media, un restaurante valorado con 3,5 o 4,5 (o cualquier otro valor intermedio) aparecería como relacionado.

Como estas cuatro listas son mostradas en el perfil del restaurante, dentro del controlador RestaurantProfileCrtl.js se definen varias funciones, una por cada lista de restaurantes relacionados.

```
readRelatedRestaurantsByCategory : function(id) {
    restaurantsFactory.getRelatedRestaurantsByCategory(id)
        .then(function(response){
            restaurantProfileViewModel.relatedRestaurantsByCategory = response
            console.log("Reading related restaurants by category (restaurant with id : ", id,") Response: ", response)
        }, function(response){
            console.log("Error reading related restaurants by category");
        })
},
readRelatedRestaurantsByCity : function(id) {
    restaurantsFactory.getRelatedRestaurantsByCity(id)
        .then(function(response){
            restaurantProfileViewModel.relatedRestaurantsByCity = response
            console.log("Reading related restaurants by city (restaurant with id : ", id,") Response: ", response)
        }, function(response){
            console.log("Error reading related restaurants by city");
        })
},
readRelatedRestaurantsByPrice : function(id) {
    restaurantsFactory.getRelatedRestaurantsByPrice(id)
        .then(function(response){
            restaurantProfileViewModel.relatedRestaurantsByPrice = response
            console.log("Reading related restaurants by price (restaurant with id : ", id,") Response: ", response)
        }, function(response){
            console.log("Error reading related restaurants by price");
        })
},
readRelatedRestaurantsByGrade : function(id) {
    restaurantsFactory.getRelatedRestaurantsByGrade(id)
        .then(function(response){
            restaurantProfileViewModel.relatedRestaurantsByGrade = response
            console.log("Reading related restaurants by grade (restaurant with id : ", id,") Response: ", response)
        }, function(response){
            console.log("Error reading related restaurants by grade");
        })
```

Figura 7. Obtención de restaurantes relacionados en restaurantsCtrl.

Los funciones anteriores a su vez obtendrán el listado de restaurantes asociados a través de métodos definidos en la factoría restaurantsFactory.js, quien se encargará de realizar una llamada a los recursos de la API RestaurantsResources.java.

```
getRelatedRestaurantsByCategory : function(id){
    var urlid = url + id + '/relatedByCategory';
    return $http.get(urlid)
        .then(function(response){
            return response.data;
        });
},
getRelatedRestaurantsByCity : function(id){
    var urlid = url + id + '/relatedByCity';
    return $http.get(urlid)
        .then(function(response){
            return response.data;
        });
},
getRelatedRestaurantsByPrice : function(id){
    var urlid = url + id + '/relatedByPrice';
    return $http.get(urlid)
        .then(function(response){
            return response.data;
        });
},
getRelatedRestaurantsByGrade : function(id){
    var urlid = url + id + '/relatedByGrade';
    return $http.get(urlid)
        .then(function(response){
            return response.data;
        });
}
```

Figura 8. Obtención de restaurantes relacionados en restaurantsFactory.

Para poder obtener los restaurantes en la API, se han implementado en el DAO los métodos necesarios para recuperar los restaurantes bajo los criterios definidos. Adicionalmente, se ha evitado que dichos métodos recuperen el restaurante en sí (para evitar que un restaurante aparezca relacionado a sí mismo) y que no haya restaurantes repetidos (como podría ser en el caso de las categorías, ya que un mismo restaurante puede compartir varias categorías con el restaurante original).

```
@GET
@Path("/{restaurantid: "+ID_REGEXP+"}/relatedByCity")
@Produces(MediaType.APPLICATION_JSON)
public List<Restaurant> getRestaurantsRelatedByCityJSON(@PathParam("restaurantid") long restaurantid,
                                                        @Context HttpServletRequest request) {
    List<Restaurant> restaurants = null;

    // Se recupera la conexión a la BD
    Connection conn = (Connection) sc.getAttribute("dbConn");
    RestaurantDAO restaurantDAO = new JDBCRestaurantDAOImpl();
    restaurantDAO.setConnection(conn);

    Restaurant restaurant = restaurantDAO.get(restaurantid);

    if(restaurant != null) {
        restaurants = restaurantDAO.getCityRelated(restaurant);
        return restaurants;
    } else {
        throw new CustomNotFoundException("Error reading restaurants related by city: restaurant does not exist");
    }
}
```

Figura 9. Obtención de restaurantes relacionados por ciudad en restaurantsResources.

4. 2. Cálculo de la cuenta instantánea

En el apartado Gestión de pedidos se ha comentado que por cada plato a añadir al pedido se puede seleccionar su cantidad. Así, cada vez que se añade un plato al pedido y se accede al apartado “Mi pedido”, se podrá visualizar un cálculo automático del precio total del pedido en función del precio de los platos seleccionados y las unidades indicadas para cada uno de ellos.

4. 3. Interfaz de usuario de alta calidad

Uno de los principales objetivos del desarrollo de la aplicación era hacerla visualmente atractiva, por lo que se ha intentado tener el máximo cuidado posible en el diseño de la interfaz.

Adicionalmente, se ha procurado conseguir una interfaz lo más “responsive” posible. Para ello, en las hojas de estilo se ha trabajado principalmente con **Flexbox** y se han empleado **unidades relativas** como porcentajes (%) o el tamaño de fuente (em). Asimismo, se han aplicado ***media queries*** para ajustar los contenidos cuando la pantalla se reduce a unas dimensiones específicas.

A continuación se muestra un ejemplo de la interfaz conseguida y su respuesta a redimensionar la pantalla:

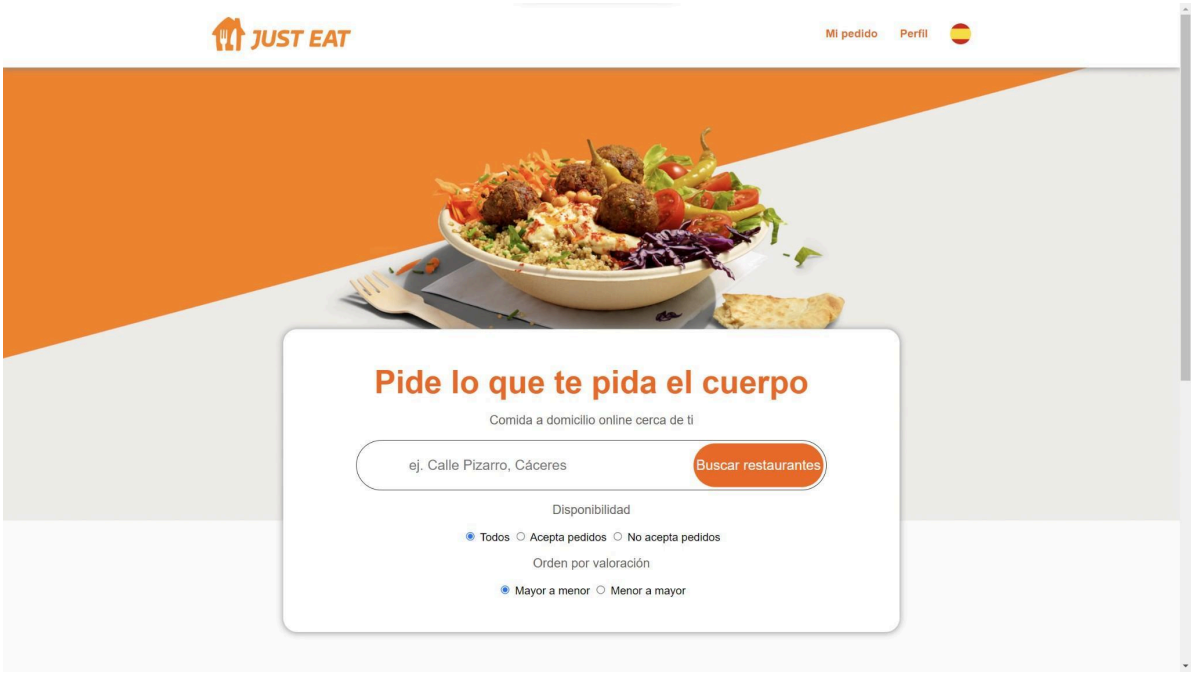


Figura 10. Página principal de la aplicación.

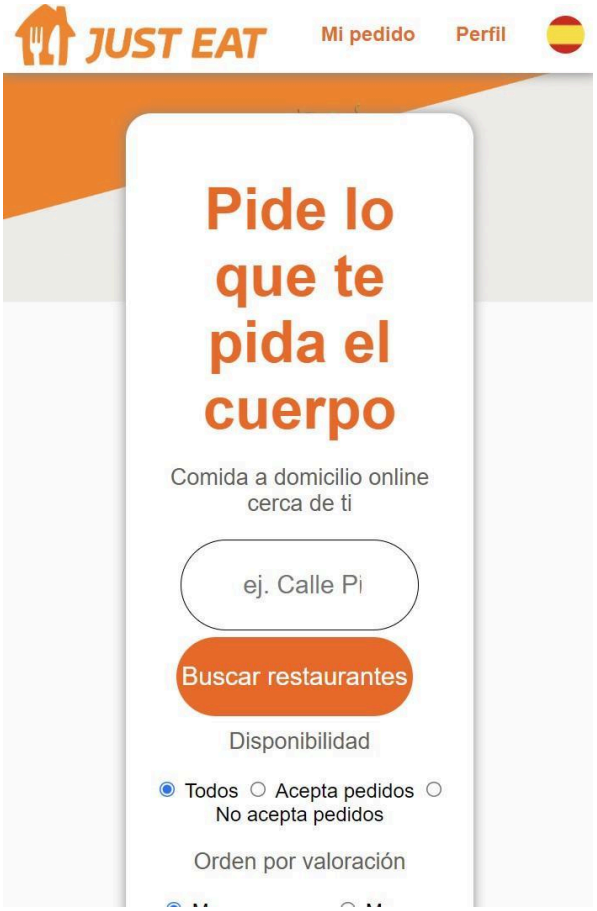


Figura 11. Página principal de la aplicación redimensionada.

4. 4. Otras funcionalidades añadidas

En este apartado destacar la inclusión de la posibilidad de poder seleccionar cantidades a la hora de añadir platos a los pedidos (Gestión de pedidos), y la tolerancia incluida con **LevenshteinSerach** en los resultados de realizar búsquedas (Buscar restaurante).

5. Situaciones a evitar - Páginas de error

Las solicitudes HTTP suelen venir junto a un código de estado el cual se encarga de determinar si dicha solicitud es correcta o no, y el tipo de información que se mostrará en cada determinado caso. En este apartado se han recopilado algunos de los errores más comunes que se pueden dar en una solicitud HTTP sobre todo aquellos errores que suelen darse por parte del cliente (códigos 4XX).

Para tratar este tipo de errores se ha modificado el código del archivo web.xml añadiendo la etiqueta , la cual permitirá redefinir el comportamiento de un determinado error, mostrando una página personalizada, con mensajes más específicos y entendibles para las personas que dispongan de conocimientos informáticos.

En este caso se han incluido el tratamiento de los errores 400, 403, 404 y 500. Los primeros son aquellos que se corresponden con errores en el lado del cliente, como la

realización de peticiones mal formuladas, acceso a recursos no disponibles, o acceso a recursos sobre los cuales no se tienen los permisos necesarios, por otro lado se incluirá un código 500 que se mostrará en caso de que se produzca un error sobre el lado del servidor y este no se encuentra disponible:

```
<!-- Error pages -->
<error-page>
  <error-code>400</error-code>
  <location>/WEB-INF/error/Error400.jsp</location>
</error-page>
<error-page>
  <error-code>403</error-code>
  <location>/WEB-INF/error/Error403.jsp</location>
</error-page>
<error-page>
  <error-code>404</error-code>
  <location>/WEB-INF/error/Error404.jsp</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/WEB-INF/error/Error500.jsp</location>
</error-page>
```

Figura 12. Definición de páginas para el control de errores conocidos.

Adicionalmente, si se produce un error no controlado (que no sea el 400, 403, 404 o 500), se ha elaborado un nuevo JSP que sea despachado en caso de producirse:

```
<error-page>
  <location>/WEB-INF/error/UnknownError.jsp</location>
</error-page>
```

Figura 13. Definición de página para el control de errores desconocidos.

Para tratar estos errores se han creado páginas JSP personalizadas donde se muestra una interfaz con el mismo “estilo” de las páginas del proyecto, donde se indicará un mensaje de error pertinente. Como ejemplo se mostrará la página de error más común, es decir, error 404, la cual indica que el recurso no se encuentra disponible:

Figura 14. Ejemplo de página de error 404.