Aplicación Web usando HTML, CSS, Servlets & JSP

Autor:

- Francisco Javier Mesa Martín

ÍNDICE

1. Introducción

En esta documentación se abordan los aspectos más significativos del proceso de desarrollo e implementación de la aplicación para la gestión de usuarios, restaurantes, platos y pedidos basada en Just Eat.

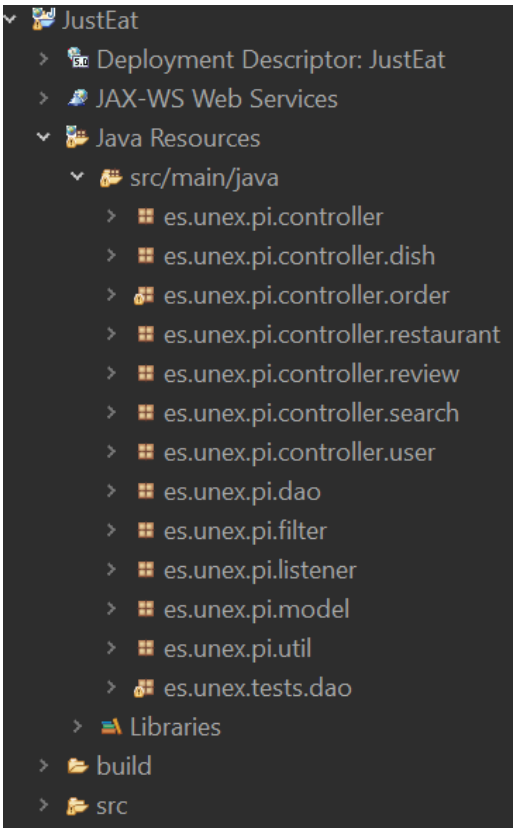
En base a lo anterior, a continuación, se proporciona un vistazo a la aplicación como producto final y a los detalles más importantes que deben ser considerados a la hora de evaluar la consecución de los requisitos mínimos impuestos, funcionales y no funcionales, y de las características adicionales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

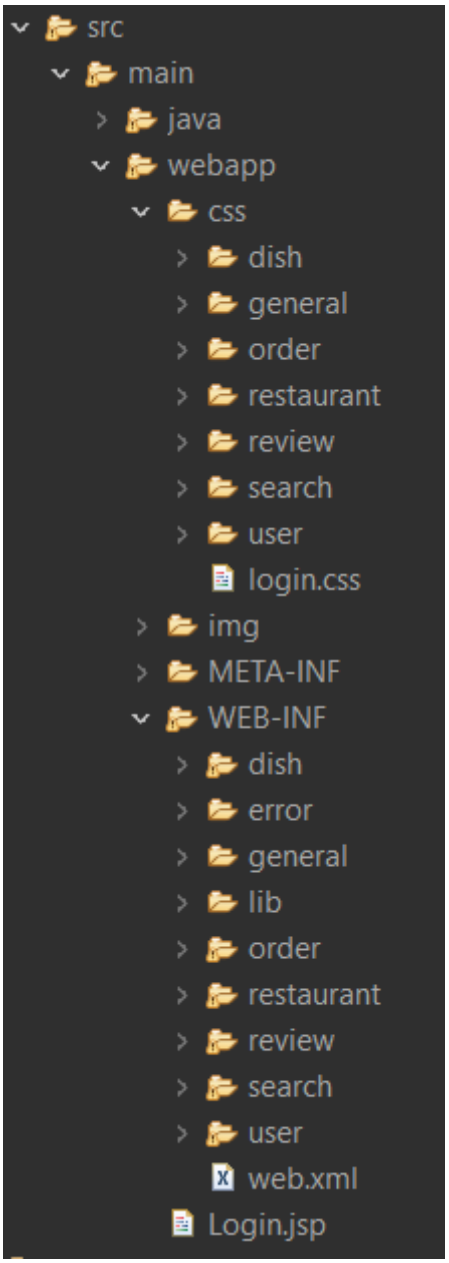
Así, en primer lugar se muestra la organización de los ficheros dentro del directorio *src/main/java* en Java Resources:



Se tienen los siguientes paquetes:

- **es.unex.pi.controller**: contiene los servlets que se encarga del inicio de sesión y del cierre de sesión.
- **es.unex.pi.controller.dish**: contiene los servlets encargados de implementar la lógica de las operaciones CRUD de los platos de los restaurantes.
- **es.unex.pi.controller.order**: contiene los servlets que se encargan de implementar la lógica de las operaciones para añadir platos a un pedido, realizar un pedido, eliminar un plato de un pedido y listar la información de un pedido.
- **es.unex.pi.controller.restaurant**: contiene los servlets que implementan la lógica de las operaciones CRUD de los restaurantes que pueden gestionar los usuarios.
- **es.unex.pi.controller.review**: contiene un único servlet que se encarga de añadir una valoración a un restaurante.
- **es.unex.pi.controller.search**: contiene los servlets que gestionan las búsquedas de restaurantes que se pueden realizar en la aplicación. Concretamente, búsqueda por categoría, búsqueda por cadena de texto introducida (incluyendo disponibilidad y orden por valoración) y la página principal en la que se recogen los dos anteriores tipos de búsqueda.
- **es.unex.pi.controller.user**: contiene los servlets que implementan la lógica de las operaciones CRUD de los usuarios.
- **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.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:



- **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.
- **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.

```
<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>
```

```
<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>
```

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 situaciones, en un principio, fueron controladas a través de filtros (*EditAndDeleteDishFilter.java, ListOrderDishesFilter.java y RestaurantFilter.java*). Sin embargo, para evitar llamadas a la base de datos innecesarias, se ha decidido repetir varias líneas de código en cada uno de los servlets pertinentes. Un ejemplo de medida de seguridad en los servlets es el siguiente:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    //Se recupera la conexión con la base de datos
    ServletContext servletContext = getServletContext();
    Connection conn = (Connection) servletContext.getAttribute("dbConn");

    try {
        // Se obtiene el ID del restaurante de la request
        Long idr = Long.valueOf(request.getParameter("id"));

        // Se obtiene el restaurante de la BD
        RestaurantDAO restaurantDAO = new JDBCRestaurantDAOImpl();
        restaurantDAO.setConnection(conn);
        Restaurant restaurant = restaurantDAO.get(idr);

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

        if(restaurant != null && restaurant.getIdu() == user.getId()) {
            request.setAttribute("restaurant", restaurantDAO.get(idr));

            // Se despacha la request
            RequestDispatcher view = request.getRequestDispatcher("WEB-INF/restaurant/DeleteRestaurantConfirmation.jsp");
            view.forward(request, response);
        } else {
            // No existe restaurante con el ID pasado o no pertenece al usuario
            response.sendRedirect("UserProfileServlet.do");
        }
    }
    catch (NumberFormatException e) {
        // No se ha intoducido un ID numérico
        response.sendRedirect("UserProfileServlet.do");
    }
}
```

Como se puede observar, se comprueba si el ID pasado en la petición es numérico, si pertenece a un objeto que exista y si el usuario propietario de dicho objeto es el usuario que ha iniciado sesión en la aplicación. En caso de no cumplirse alguna de estas condiciones, se redirige a una página concreta de la aplicación, como podría ser el perfil del usuario que ha iniciado sesión o la página principal desde la que se puede realizar las búsquedas.

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.

En tercer y último lugar, se ha evitado el poder realizar **peticiones no idempotentes** a través de la aplicación del patrón **PRG (Post-Redirect-Get)**. Es decir, se ha evitado que peticiones tipo POST que alteren el estado de la base de datos puedan realizarse consecutivamente (por ejemplo, recargando la página para enviar de nuevo la petición).

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/WEB-INF/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. 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 JSP para la creación/edición de usuarios, restaurantes y platos.

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.

En este enlace se deja un vídeo en el que se muestra la funcionalidad de la aplicación y su interfaz.

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 (SerachAndCategoriesServlet.do). 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 a UserProfileServlet.do.
    - **En UserProfileServlet.do**:
        - **doGet**: recupera de la sesión el usuario que ha iniciado sesión. Con esta información, recupera de la base de datos los restaurantes que

el usuario ha creado y los pedidos que ha realizado. Esto último se almacena todo en la petición y se despacha a UserProfile.jsp. Dentro de este JSP se da la opción a editar el perfil de usuario o eliminarlo.
        - **doPost**: este método únicamente redirige al método doGet del mismo servlet, puesto que no necesita funcionalidad ninguna.
- Dentro del perfil, el usuario puede pulsar en “Editar perfil”. Esta acción redirigirá a EditUserServlet.do.
    - **En EditUserServlet.do**:
        - **doGet**: rellena EditUser.jsp con la información propia del usuario para que al editar el perfil ya esté la información precargada.
        - **doPost**: se comprueba que los nuevos datos introducidos sean correctos. Como se da la opción de cambiar el email, se debe comprobar de nuevo que no exista otro usuario con el mismo email. Si todo es correcto, se actualiza el usuario en la base de datos y en la sesión.
- Dentro del perfil el usuario también puede eliminar su cuenta pulsando en “Eliminar cuenta”. De forma breve, **DeleteUserServlet.do** se encargará de ello. Para esto, el doGet muestra una página de confirmación de borrado. En caso de aceptar, el doPost elimina el usuario de la base de datos y de la sesión, redirigiendo después al inicio de sesión. El borrado del usuario supone el borrado de todos sus restaurantes, valoraciones y pedidos.
- 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. Esto llamará a **CreateRestaurantServlet.do**. En su método doGet se despachará al formulario de creación (EditRestaurant.jsp, que será reutilizado para la edición) junto a las categorías disponibles para asignar. 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. El método doPost del servlet será el encargado de validar la información. Se ha tenido en cuenta lo siguiente:
    - 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 **RestaurantProfileServlet.do**. El método doGet de este servlet recupera la siguiente información para despacharsela a RestaurantProfile.jsp y mostrarla por pantalla:
    - Toda la información propia del restaurante, incluyendo las categorías.
    - Las valoraciones (o reseñas) que ha recibido el restaurante.
    - Los platos creados para el restaurante.

    - Los restaurantes relacionados por categorías, ciudad (localidad), rango de precios y valoración media. Esto pertenece al requisito extra de Restaurantes relacionados.
- Dentro del perfil del restaurante se puede editar su información pulsando en “Editar este restaurante”. Esto redirigirá a **EditRestaurantServlet.do**. En él, el método doGet recupera la información del restaurante y la precarga en el formulario ofrecido por EditRestaurant.jsp, para que el usuario no deba introducir toda la información de nuevo. Introducidos los nuevos datos, el método doPost se encarga de validar la información. En caso de que sea todo correcto, se actualiza el restaurante en la base de datos. En caso contrario, se muestran los errores cometidos para volver a mandar los datos.
- Por último, dentro del propio perfil del restaurante también se puede proceder a su borrado pulsando en “Eliminar este restaurante”. El servlet **DeleteRestaurantServlet.do** se encarga de gestionar el proceso de borrado. Al igual que sucedía con los usuarios, el método doGet lleva a una página de confirmación de borrado (DeleteRestaurant.jsp). El método doPost se encargará de borrar el restaurante de la base de datos en caso de proceder con la eliminación. 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.

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). Esto redirigirá a **CreateDishServlet.do**. El método doGet de este servlet despacha la petición a EditDish.jsp, que proporciona el formulario necesario para crear un plato. Este JSP es reutilizado para editar los platos. Introducidos los datos, el método doPost comprueba que la información sea válida. Se ha considerado que un mismo restaurante no puede tener dos platos con el mismo nombre. Si esto se cumple, el plato es almacenado en la base de datos. En caso contrario, se informa al usuario del error para volver a introducir los datos.
- 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”. Esta acción redirigirá a **EditiDishServlet.do**, cuyo método doGet precargará la información del plato antes de despachar a EditDish.jsp. Al igual que al crear el plato, al editarlo se comprobará en el doPost que no existe otro plato en el restaurante con el mismo nombre. Si esto se cumple, se actualiza el plato en la base de datos. Si no, se informa del error para volver a editar el plato.
- 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. En este caso, se redirige a **DeleteDishServlet.do**. El método doGet de este servlet muestra al usuario una página de confirmación para la eliminación del plato. Si el usuario confirma el borrado, se llama al método doPost para eliminar el plato de la base de datos y redirigir al perfil del restaurante en cuestión. El borrado de un plato no supone ningún cambio en cascada 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”, accede a **CreateOrderServlet.do**. Su método doGet muestra los platos que el usuario ha seleccionado hasta el momento, el precio total (calculado de forma automática) y la opción de efectuar su pedido. La gestión de los platos añadidos al pedido se realiza a través de la sesión con un mapa de objetos Triplet con el ID del plato como clave. En cada objeto Triplet se tiene:
    - **OrderDishes**: contiene el ID del plato y el ID del pedido (vacío ya que aún el pedido no se ha realizado). Adicionalmente se ha añadido a este objeto un nuevo campo que indica la cantidad de unidades que se piden del plato.
    - **Dish**: la información del plato.
    - **Restaurant**: la información del restaurante al que pertenece el plato.

  Tener toda esta información desde el principio evita tener que realizar posteriormente llamadas innecesarias a la base de datos.
  Desde la lista de platos mostrados se ofrece la posibilidad al usuario de eliminarlos del pedido al pulsar “Eliminar del pedido”
- Al pulsar en la anterior opción de eliminar, se ejecuta el método doGet de **DeleteDishFromOrderServlet.do**. Este se encarga de recuperar el plato que se desea borrar y despacha la petición a DeleteDishFromOrder.jsp para que el usuario confirme el borrado. Al confirmarlo, el método doPost del mismo servlet recupera el plato a borrar y lo elimina del anterior mapa, redirigiendo después a la lista de platos del pedido (CreateOrderServlet.do).
- En caso de añadir nuevos platos al pedido, el usuario se debe dirigir al perfil del restaurante en cuestión (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. De esta forma, al seleccionar una cantidad (por defecto 1) y pulsa en añadir a pedido, se ejecuta el método doPost de **AddDishToOrderServlet.do**. Este comprobará si ya existe el plato en el pedido. En caso afirmativo, se acumulan las cantidades. En caso contrario, se añade un nuevo objeto Triplet al mapa ya comentado.
- 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. Este enlace lleva a **ListOrderDishesServlet.do**. El método doGet de este servlet recupera todos los platos del pedido y los muestra por pantalla junto a las unidades pedidas y el precio total. 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.

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. Por ello, **RestaurantProfileServlet.do** se encarga de recuperar, en el doGet, todas las valoraciones del restaurante junto a

todos los usuarios que han realizado valoraciones (ya que se requiere de los nombres de los usuarios que han realizado dichas reseñas) antes de despachar a RestaurantProfile.jsp.

- 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". Esta opción se hará visible en el JSP si en la colección de usuarios pasados que han realizado reseñas no se encuentra el usuario de la sesión. Al pulsar en la anterior opción se redirigirá a **CreateReviewServlet.do**. El doGet de este servlet comprueba también que el usuario no haya añadido ya una reseña al restaurante, ya que alguien podría acceder al formulario a través de URL. En este caso, se dirigiría a la página principal. En caso de no haber introducido una reseña, se despacha a CreateReview.jsp, que se encarga de mostrar al usuario el formulario para una nueva reseña. Así, el usuario deberá indicar su valoración (del 1 al 5) y un comentario (opcional). Al añadir la reseña, el método doPost será el responsable de almacenarla en la base de datos y actualizar, también en base de datos, la valoración media del restaurante en cuestión. Para esto último se ha añadido un nuevo método al DAO del restaurante (*updateGradesAverage*). Tras esto, redirigirá al perfil del restaurante que se ha valorado.
- Desde la página principal, SearchAndCategories.jsp, 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 la página ofrecida por **SearchAndCategoriesServlet.do** en su método doGet. Este, antes de despachar a SearchAndCategories.jsp recupera todas las categorías existentes para ofrecer a los usuarios dos métodos de búsqueda:
    - **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.

- **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 por **RestaurantSearchListServlet.do** en su método doGet. 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 servlet 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 **RestaurantCategoryList.do** en su método doGet.

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 al servlet **RestaurantSearchList.do** y se encarga de consultar los restaurantes sin tener en cuenta la disponibilidad).

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, el método doGet de **RestaurantProfileServlet.do** es el encargado de recuperar de base de datos toda esta información antes de despachar a RestaurantProfile.jsp. Para ello, 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).

```
//--------------------Ampliación 4-------------------------

// Se recuperan los restaurantes relacionados por categorías y se añaden a la request
// Se usa un mapa para no duplicar restaurantes
Map<Long, Restaurant> similarCategoriesMap = new HashMap<>();
for (RestaurantCategories restaurantCategories : restaurantCategoriesDAO.getAllByRestaurant(idr)) {
    for (RestaurantCategories restaurantCategories2 : restaurantCategoriesDAO.getAllByCategory(restaurantCategories.getIdct())) {
        similarCategoriesMap.put(restaurantCategories2.getIdr(), restaurantDAO.get(restaurantCategories2.getIdr()));
    }
}
similarCategoriesMap.remove(idr); // Se elimina del mapa el propio restaurante

List<Restaurant> listAux = new ArrayList<>(similarCategoriesMap.values());
request.setAttribute("similarCategoriesList", listAux);

// Se recuperan los restaurantes relacionados por localidad y se añaden a la request
listAux = restaurantDAO.getCityRelated(restaurant);
request.setAttribute("similarCityList", listAux);

// Se recuperan los restaurantes relacionados por precio y se añaden a la request
listAux = restaurantDAO.getPriceRelated(restaurant);
request.setAttribute("similarPriceList", listAux);

// Se recuperan los restaurantes relacionados por valoración y se añaden a la request
listAux = restaurantDAO.getGradeRelated(restaurant);
request.setAttribute("similarGradeList", listAux);

// Se despacha la request con toda la información del restaurante, sus categorías y sus platos
RequestDispatcher view = request.getRequestDispatcher("WEB-INF/restaurant/RestaurantProfile.jsp");
view.forward(request, response);
```

4.2. Cálculo de la cuenta instantánea

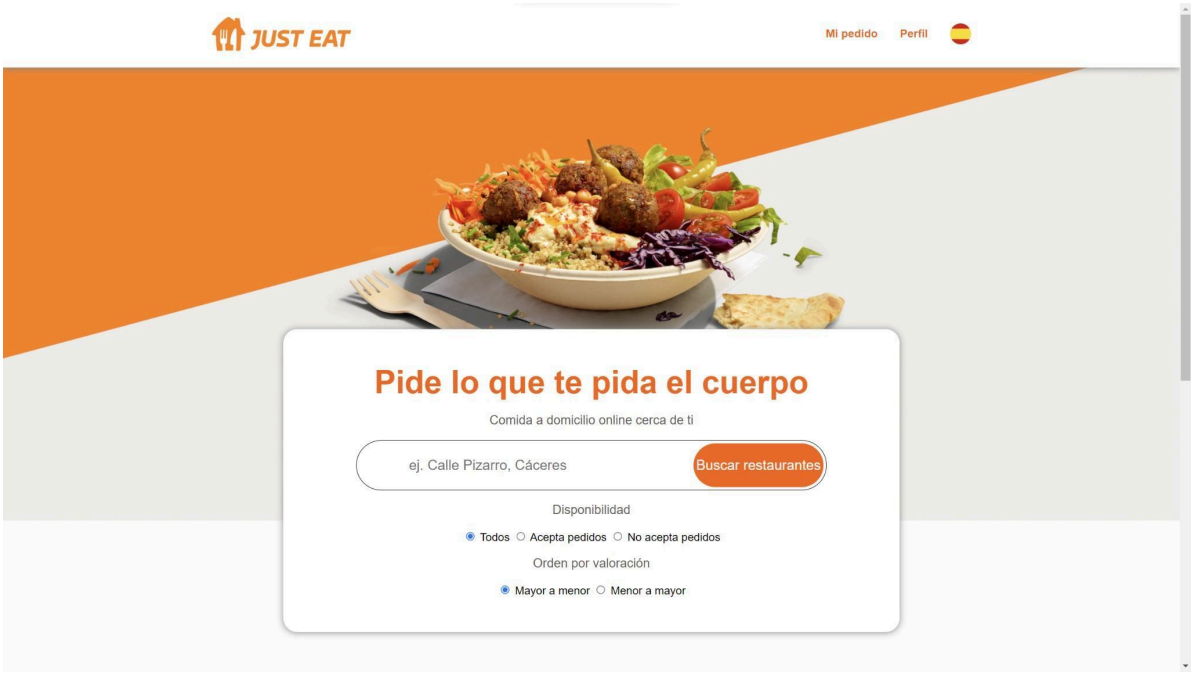
En el apartado Gestión de pedidos se ha comentado que por cada plato a añadir al pedido se puede seleccionar su cantidad. Así, cada vez que se añade un plato al pedido y se accede al apartado “Mi pedido”, se podrá visualizar un cálculo automático del precio total del pedido en función del precio de los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

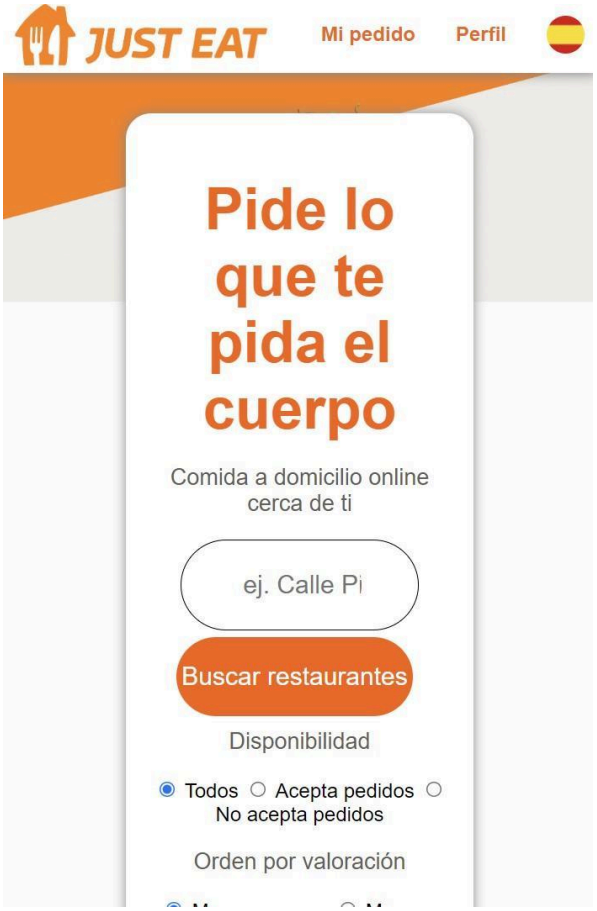
4.3. Interfaz de usuario de alta calidad

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:





En este enlace se deja un vídeo interactuando con la interfaz de las aplicación para ver cómo se ajusta a la pantalla.

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 <error-page>, 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-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>
```

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>
```

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: