

Proyecto Integrado

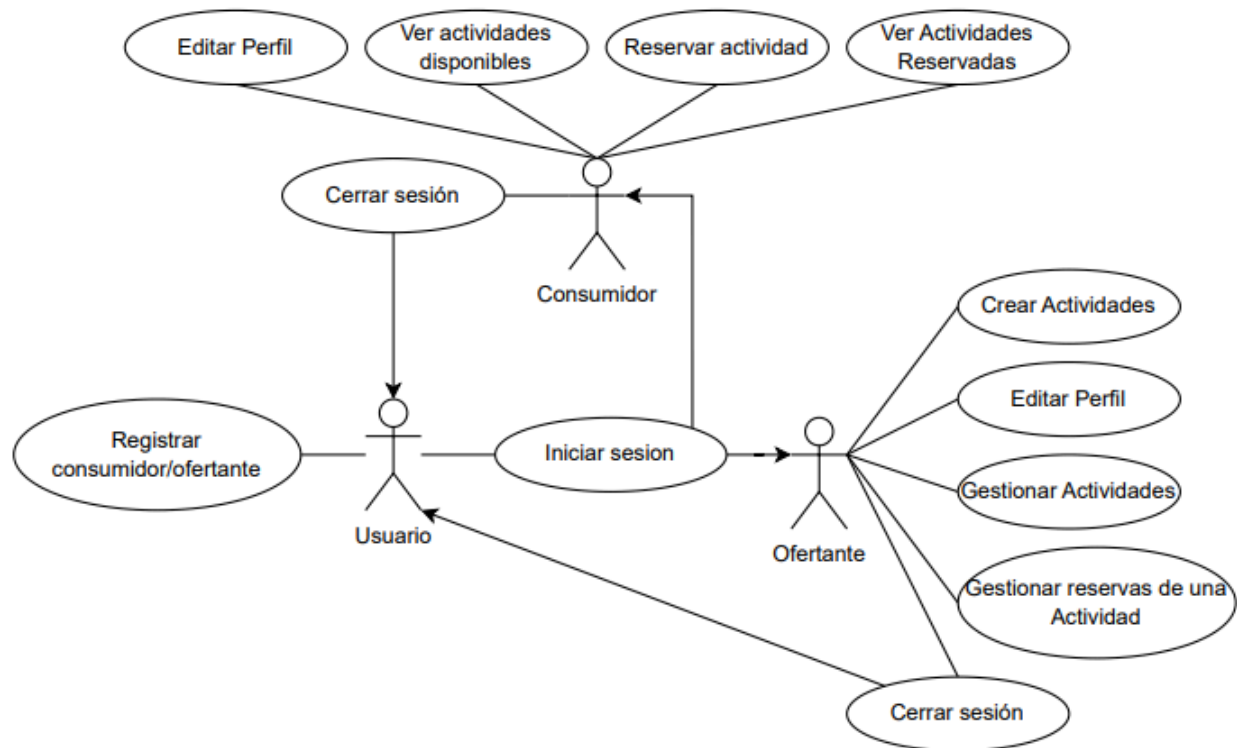
Documentación

Manuel Rubio Álvarez

Desarrollo de Aplicaciones Web - 2023/2024

Manual de Usuario - Funcionamiento de la Aplicación

Explicaremos un poco lo que podrá hacer el usuario cuando entre a la aplicación:



Como podemos ver en la foto, el usuario lo primero que hará será registrarse como consumidor u ofertante, o iniciar sesión en caso de que ya disponga de una cuenta. En caso de que entre como ofertante, este podrá crear una nueva actividad, gestionar las actividades que ya ha creado anteriormente, gestionar las reservas que los consumidores han hecho de sus actividades y por último cerrar sesión para volver a entrar como ofertante o como consumidor, depende de lo que guste en ese momento. En caso de que entre como consumidor puede ver las actividades que hay disponibles, reservar su plaza en la actividad que quiera, ver las actividades en las que te has suscrito (Has hecho reserva anteriormente), editar tu perfil y, como hemos explicado también con ofertantes, también puedes cerrar sesión para entrar como ofertante o como consumidor, depende de lo que guste en ese momento.

Manual de Usuario - Backend del Proyecto

Introducción:

En este apartado daremos una guía sobre el funcionamiento del backend del proyecto. Nos enfocaremos en explicar las clases y sus propósitos.

Estructura, clases y su funcionamiento

Empecemos explicando la **estructura** del proyecto. En primer lugar tenemos la clase de configuración, *WebConfig.java*, esta clase nos sirve para habilitar CORS (Cross-Origin Resource Sharing), sin esta clase no podríamos lanzar la aplicación en otras rutas. Seguimos con las clases de excepciones personalizadas, en nuestro caso tenemos dos, una que usamos para excepciones de cualquier tipo, *CustomException.java*, y otra centrada en excepciones de llave foránea, *ForeignKeyException.java*. También tenemos un manejador de excepciones, esta clase se encarga de manejar las clases de las excepciones. Continuemos con los servicios, estos se usan para manejar la lógica de negocio de la aplicación, en nuestro caso contamos con cinco servicios, que son los siguientes:

1. **UsuarioService:** Maneja la lógica de negocio relacionada con usuarios. Funcionalidad:
 - a. Buscar usuario por email.
 - b. Guardar usuario.
 - c. Registrar un nuevo usuario.
 - d. Obtener usuario por ID.
2. **ReservaService:** Maneja la lógica de negocio relacionada con reservas. Funcionalidad:
 - a. Crear una nueva reserva.
 - b. Obtener reservas por consumidor con paginación.
 - c. Obtener reservas por actividad.
 - d. Cancelar una reserva.
 - e. Verificar si existe una reserva para una actividad y consumidor específicos.
3. **OfertanteService:** Maneja la lógica de negocio relacionada con ofertantes. Funcionalidad:
 - a. Buscar ofertante por ID.
 - b. Guardar ofertante.
 - c. Actualizar ofertante.
 - d. Eliminar ofertante por ID.
4. **ConsumidorService:** Maneja la lógica de negocio relacionada con consumidores. Funcionalidad:
 - a. Buscar consumidor por ID.

- b. Guardar consumidor.
 - c. Actualizar consumidor.
 - d. Eliminar consumidor por ID.
5. **ActividadService:** Maneja la lógica de negocio relacionada con actividades.
- Funcionalidad:
- a. Obtener todas las actividades con paginación.
 - b. Obtener actividades por ID de ofertante con paginación.
 - c. Guardar actividad.
 - d. Eliminar actividad por ID.
 - e. Buscar actividad por ID.

Continuemos con los **repositorios**, estos son como una interfaz que ofrece operaciones relacionadas con una clase de dominio para interactuar con la base de datos, y en nuestro realizan varias tareas relacionadas con la persistencia de datos y nos facilitan las operaciones CRUD, como son los siguientes:

1. **ActividadRepository:** Manejamos la tabla Actividad de la base de datos. Funcionalidad:
 - a. Buscar todas las actividades con paginación.
 - b. Buscar actividades por ID de ofertante con paginación.
2. **UsuarioRepository:** Manejamos la tabla Usuario de la base de datos. Funcionalidad:
 - a. Buscar usuario por email.
3. **ReservaRepository:** Manejamos la tabla Reserva de la base de datos. Funcionalidad:
 - a. Buscar reservas por ID de actividad.
 - b. Buscar reservas por ID de consumidor con paginación.
 - c. Verificar si existe una reserva por ID de actividad y consumidor.
4. **OfertanteRepository:** Manejamos la tabla Ofertante de la base de datos. Funcionalidad:
 - a. Eliminar ofertante por ID.
5. **ConsumidorRepository:** Manejamos la tabla Consumidor de la base de datos. Funcionalidad:
 - a. Eliminar consumidor por ID.


Tras explicar los repositorios, podemos explicar los **controladores**, estas clases son las responsables de manejar las solicitudes HTTP entrantes y enviar respuestas. En nuestro proyecto tenemos las siguientes:

1. **UsuarioController:** Maneja solicitudes relacionadas con usuarios. Endpoints:
 - a. POST /api/usuarios/login: Autenticar usuario.
 - b. POST /api/usuarios/register: Registrar un nuevo usuario.
 - c. GET /api/usuarios/email/{email}: Obtener usuario por email.
 - d. GET /api/usuarios/{id}: Obtener usuario por ID.

2. **ReservaController:** Maneja solicitudes relacionadas con reservas. Endpoints:
 - a. POST /api/reservas: Crear una nueva reserva.
 - b. GET /api/reservas/consumidor/{consumidorId}: Obtener reservas por ID de consumidor con paginación.
 - c. GET /api/reservas/actividad/{actividadId}: Obtener reservas por ID de actividad.
 - d. DELETE /api/reservas/{reservald}: Cancelar una reserva.
 - e. GET /api/reservas/existe: Verificar si existe una reserva.
3. **OfertanteController:** Maneja solicitudes relacionadas con ofertantes. Endpoints:
 - a. GET /api/ofertantes/{id}: Obtener ofertante por ID.
 - b. PUT /api/ofertantes/editar/{id}: Editar un ofertante.
 - c. DELETE /api/ofertantes/eliminar/{id}: Eliminar un ofertante.
4. **ConsumidorController:** Maneja solicitudes relacionadas con consumidores. Endpoints:
 - a. GET /api/consumidores/{id}: Obtener consumidor por ID.
 - b. PUT /api/consumidores/editar/{id}: Editar un consumidor.
 - c. DELETE /api/consumidores/eliminar/{id}: Eliminar un consumidor.
5. **ActividadController:** Maneja solicitudes relacionadas con actividades. Endpoints:
 - a. GET /api/actividades/ofertante/{ofertanteld}: Obtener actividades por ID de ofertante con paginación.
 - b. POST /api/actividades/crear: Crear una nueva actividad.
 - c. PUT /api/actividades/editar/{id}: Editar una actividad.
 - d. DELETE /api/actividades/eliminar/{id}: Eliminar una actividad.
 - e. GET /api/actividades/todas: Obtener todas las actividades con paginación.
 - f. GET /api/actividades/{id}: Obtener actividad por ID.

Vamos con los últimos apartados de nuestro manual enfocado al backend, ahora tocan los **DTOs (Data Transfer Objects)** que hemos usado en nuestra aplicación, como su nombre indica, estas clases las usamos para transportar datos, y en nuestro proyecto hemos implementado las siguientes.

1. **RegisterRequest:** Transferimos datos de actividades. Campos:
 - a. nombre, apellido, email, password, dirección, teléfono y role.
2. **LoginRequest:** Transferimos datos de actividades. Campos:
 - a. email, password y role.
3. **JwtResponse:** Transferimos datos de actividades. Campos:
 - a. token y usuario.
4. **ActividadDTO:** Transferimos datos de actividades. Campos:
 - a. titulo, descripcion, duracion, numMinParticipantes, numMaxParticipantes, fecha, lugar, materialNecesario, materialOfrecido, tipo y ofertanteld.



Y por último, pasemos a explicar el **modelo** que he decidido implementar en mi proyecto. Estas clases, también conocidas como entidades, son cruciales para presentar y gestionar los datos de la base de datos de la aplicación. Proporcionan un mapeo entre los objetos Java y las tablas de la base de datos, definen relaciones entre entidades, permiten la validación de datos y facilitan el uso de repositorio para operaciones CRUD. En nuestro proyecto hemos usado las siguientes:

1. **Usuario:** Representa al usuario en la aplicación, sus campos son: id, nombre, apellido, email, password, direccion, telefono. Sus relaciones son:
 - a. Uno a uno con 'Consumidor'
 - b. Uno a uno con 'Ofertante'
2. **Reserva:** Representa una reserva en la aplicación, y consta de los campos: id, consumidor, actividad, fechaReserva. Sus relaciones son:
 - a. Muchos a uno con 'Consumidor'
 - b. Muchos a uno con 'Actividad'
3. **Consumidor:** Representa a un consumidor en la aplicación, y consta de los campos: id, usuario, nombre, apellidos, telefono, direccion. Sus relaciones son:
 - a. Uno a uno con 'Usuario'
 - b. Uno a muchos con 'Reserva'
4. **Ofertante:** Representa a un ofertante en la aplicación, y consta de los campos: id, usuario, nombre, apellidos, descripcion, telefono, direccion. Sus relaciones son:
 - a. Uno a uno con 'Usuario'
 - b. Uno a muchos con 'Actividad'
5. **Actividad:** Representa una actividad en la aplicación, y consta de los campos: id, titulo, descripcion, duracion, numMinParticipantes, numMaxParticipantes, fecha, lugar, materialNecesario, materialOfrecido, tipo. Sus relaciones son:
 - a. Muchos a uno con Ofertante.
 - b. Uno a muchos con Reserva.

Manual de Usuario - Frontend del Proyecto

Introducción

En este apartado explicaremos el funcionamiento de las clases del front-end del proyecto, explicando su uso y propósito. Este proyecto está desarrollado en Angular 17 y cubre funcionalidades relacionadas con usuarios (consumidores y ofertantes), actividades y reservas.

Clases y su funcionamiento

El proyecto está estructurado en dos tipos de clases, servicios y componentes, a su vez, cada componente consta de html, css y ts. Empecemos explicando los **servicios**. Los servicios cumplen varias funciones cruciales que permiten que la aplicación funcione de manera eficiente y estructurada. Servicios usados en la aplicación:

1. **ReservaService:** Gestiona todas las operaciones relacionadas con las reservas de las actividades. *Funciones:*
 - a. Crear Reservas: Permite a los consumidores reservar actividades.
 - b. Obtener Reservas por Consumidor: Proporciona a los consumidores la posibilidad de ver sus reservas con paginación.
 - c. Verificar Existencia de Reservas: Evita que los consumidores dupliquen reservas para la misma actividad.
 - d. Obtener Reservas por Actividad: Permite a los ofertantes ver todas las reservas asociadas a sus actividades.
 - e. Cancelar Reservas: Ofrece la funcionalidad para que los consumidores cancelen sus reservas si es necesario.
2. **RegisterService:** Gestiona el registro de nuevos usuarios, ya sean consumidores u ofertantes. *Funciones:*
 - a. Registrar Usuarios: Facilita la creación de nuevas cuentas de usuario en la aplicación, asegurando que los datos se envían y almacenan correctamente.
3. **OfertanteService:** Administra las operaciones de los ofertantes. Con él hacemos el crud de ofertantes. *Funciones:*
 - a. Obtener Datos del Ofertante: Recupera la información detallada de un ofertante.
 - b. Editar Ofertantes: Permite a los ofertantes actualizar su perfil y detalles.
 - c. Eliminar Ofertantes: Proporciona la capacidad de eliminar una cuenta de ofertante, asegurando que no se dejen datos residuales.
4. **LoginService:** Gestiona la autenticación y la gestión de la sesión de usuarios. *Funciones:*

- a. Inicio de Sesión: Autentica a los usuarios y maneja la lógica para mantener su sesión activa.
 - b. Cerrar Sesión: Gestiona la terminación de la sesión del usuario, limpiando los datos relacionados.
 - c. Obtener Detalles del Usuario: Recupera la información del usuario actualmente autenticado.
 - d. Actualizar Datos del Usuario Actual: Permite la actualización de los datos del usuario que está en sesión, reflejando cambios inmediatos en la aplicación.
5. **ConsumidorService**: Gestiona las operaciones de los consumidores, podemos hacer el CRUD de consumidores. *Funciones*:
- a. Obtener Datos del Consumidor: Recupera la información detallada de un consumidor.
 - b. Editar Consumidores: Permite a los consumidores actualizar su perfil y detalles.
 - c. Eliminar Consumidores: Proporciona la capacidad de eliminar una cuenta de consumidor, asegurando que se manejen correctamente las dependencias como reservas activas.
6. **ActividadService**: Gestiona las operaciones relacionadas con las actividades, podemos hacer el CRUD completo de actividades. *Funciones*:
- a. Obtener Todas las Actividades: Facilita la búsqueda y visualización de todas las actividades disponibles con soporte para paginación.
 - b. Obtener Actividad por ID: Proporciona los detalles de una actividad específica.
 - c. Obtener Actividades por Ofertante: Permite a los ofertantes ver sus actividades con soporte para paginación.
 - d. Crear Actividades: Facilita la creación de nuevas actividades.
 - e. Editar Actividades: Permite a los ofertantes actualizar los detalles de sus actividades.
 - f. Eliminar Actividades: Proporciona la funcionalidad para eliminar actividades, manejando adecuadamente las reservas activas.
 - g. Verificar Reservas Activas: Permite a los ofertantes verificar si una actividad tiene reservas activas antes de realizar ciertas acciones.

Básicamente, los servicios son esenciales para gestionar las operaciones relacionadas con los usuarios, actividades y reservas.

Expliquemos ahora los **componentes**, estos son fundamentales para construir la interfaz del usuario. Cada componente sirve para una cosa, y guarda tanto la lógica (ts) como la presentación (html y css) de la aplicación, en ellos es donde llamamos a los servicios para que todo funcione correctamente. Expliquemos los presentes en mi proyecto:

1. **RegisterComponent:** Permite a los usuarios registrarse en la aplicación, ya sea como consumidor o como ofertante. En cuanto a su funcionalidad, presenta un formulario de registro, valida los datos ingresados y envía la información al servicio de registro.
2. **LoginComponent:** Maneja el inicio de sesión de los usuarios. En cuanto a su funcionalidad, presenta un formulario de inicio de sesión, autentica al usuario y redirige a la página correspondiente según su rol, consumidor u ofertante.
3. **MenuOfertanteComponent:** Proporciona un menú de navegación específico para los ofertantes. En cuanto a su funcionalidad, permite a los ofertantes acceder a diferentes secciones de la aplicación, como su perfil, actividades y cerrar sesión.
4. **MenuConsumidorComponent:** Proporciona un menú de navegación específico para los consumidores. En cuanto a su funcionalidad, permite a los consumidores acceder a diferentes secciones de la aplicación, como su perfil, actividades y cerrar sesión.
5. **HomeOfertanteComponent:** Muestra las actividades creadas por el ofertante y permite gestionar estas actividades. En cuanto su funcionalidad, muestra una lista paginada de actividades del ofertante, permite crear, editar y eliminar actividades, y gestionar las reservas de cada actividad.
6. **HomeConsumidorComponent:** Muestra las actividades disponibles para los consumidores y se les permite realizar la actividad que deseen. En cuanto su funcionalidad, muestra una lista paginada de actividades que les permite reservar la actividad que deseen.
7. **GestionReservasComponent:** Permite a los oferentes gestionar las reservas de sus actividades. En cuanto a su funcionalidad, muestra las reservas de una actividad específica y permite cancelarlas si gusta.
8. **EditarActividadComponent:** Permite a los ofertantes editar sus actividades. En cuanto a su funcionalidad, presenta un formulario con los datos de la actividad a editar, valida los cambios y los envía al servicio de actividades para su actualización.
9. **CrearActividadComponent:** Permite a los ofertantes crear una nueva actividad. En cuanto a su funcionalidad, presenta un formulario para ingresar los datos de una nueva actividad, valida la información y la envía al servicio de actividades para su creación.
10. **ConsumidorPerfilComponent:** Muestra y permite la edición del perfil del consumidor, así como gestionar sus reservas. En cuanto a su funcionalidad, muestra los datos del perfil del consumidor y sus reservas, permite actualizar la información del perfil y cancelar reservas.



Conclusión personal

Puedo decir que estoy contento con el resultado de mi aplicación, es concisa pero funciona correctamente y he conseguido implementar el funcionamiento que quería. Aunque he tenido muchísimas dificultades.

El hecho de que al desplegar la aplicación no pueda recargar la página porque peta me ha desestructurado muchas cosas de mi proyecto. Usaba el método `window.location.reload()` en alguna que otra clase, pero al no poder usarlo, he tenido que buscar la forma de implementar algo que funcione como ese método, y encontré una librería de angular con la que podía hacer algo parecido, [ChangeDetectorRef](#), esta librería tiene un método que es `detectChanges`, lo que hace es forzar la detección de cambios, lo que hace que se actualice la vista, que es lo que buscaba para reemplazar ese método. Me ha pasado lo mismo para actualizar los datos de un ofertante, pero en este caso no he podido usar esta librería, por lo que se me ocurrió crear un método que navegara a otro componente de forma temporal y volviera al componente al que estaba, y con eso se solucionó.

Otra dificultad que he encontrado a sido el empaginado, ya que anteriormente en el mismo método tenía tanto el paginado de las reservas como el de las actividades juntos, entonces se me petaba mucho porque detectaba cosas diferentes de un método a otro, la solución fue facil. Cuando estaba estructurando la pagina, encontrándome en el diseño decidí que era mejor separarlo, y así fue como se solucionó.

Y sin duda alguna, donde más he encontrado dificultades es a la hora de desplegar la aplicación al proxmox. No paraban de encontrarme con errores continuamente, errores con maven, con tomcat, a la hora de crear la carpeta target en el back, de todo, por suerte y con esfuerzo conseguí solucionarlo todo.

Esas son algunas de las dificultades que me he encontrado durante el desarrollo de mi aplicación, digamos que estoy orgulloso de cómo he trabajado estos días y de la resolución que he empleado para esos problemas.

Y por último, lo que más me ha gustado hacer es el home del ofertante, ya que tiene muchas cosas diferentes, y reflejo muchas de las cosas que hemos dado en angular.

