INGENIERÍA DEL SOFTWARE II

DOCUMENTACIÓN

GRUPO 5: PABLO ALCOLEA, IKER GARCÍA, UNAI LEÓN

Repositorio de GitHub
https://github.com/unai002/RidesJSFG5

11/12/2024

El proyecto se ha desarrollado en dos partes, implementando primero el framework Java Server Faces (JSF) y después Hibernate.

En el caso de JSF tenemos las siguientes utilidades:

- QueryRides y CreateRide.
- Login y Register.
- Apuntarse a viajes y Acceso a viajes propios (utilidades extra).

Y en Hibernate contamos con la lógica de negocio y el acceso a datos.

En cada parte se explicarán las utilidades por separado, las implementaciones destacables para su correcto funcionamiento y, en caso de haber, los controles de validación que se realizan.

1. JAVA SERVER FACES (JSF)

a. Consultar viajes disponibles

Esta función está disponible para todo tipo de User (más adelante se profundizará en esto). Para ello se ha creado una página HTML (**QueryRides.xhtml**), que contiene desplegables para seleccionar las ciudades de origen y destino y un calendario para seleccionar el día del viaje.

Funcionamiento

El desplegable de ciudad de origen contiene las ciudades que aparezcan como ciudad de origen en algún viaje ya creado. El desplegable de la ciudad de destino se muestra deshabilitado mientras no se haya seleccionado ninguna ciudad de origen y solo muestra ciudades de destino que contenga algún viaje con la ciudad de origen seleccionada. El calendario es únicamente para seleccionar un día y en caso de que en ese día haya viajes disponibles, se desplegará una tabla con la siguiente información:

- Nombre del conductor (Driver) que ofrece el viaje.
- Asientos disponibles para el viaje.
- Precio por asiento.
- Botón reserva (más adelante se profundizará en esto).

Estaba previsto que el calendario resaltase los días que tienen un viaje pero no conseguimos que el atributo **highlightedDates** funcionara como lo planeado.

b. Ofertar viajes

Esta función está solo disponible para los usuarios registrados como Driver (más adelante se profundizará en esto). Para ello se ha generado una página HTML (**CreateRide.xhtml**)

que contiene cuadros de texto para escribir las ciudades de origen y destino, el número de asientos disponibles y el precio del asiento y un calendario para seleccionar el día en el que se quiere ofertar el viaje. La lógica para el funcionamiento de la página está en **CreateRideBean**.

Funcionamiento

En los cuadros de texto se escribirán los valores correspondientes a cada campo. Una vez rellenos los campos, se elegirá el día en el que se quiere ofertar el viaje y se pulsará el botón **PUBLICAR OFERTA**.

Controles de validación

- **Ciudad de salida y de destino**. Cadena de texto formada solo por caracteres no numéricos. Se ha añadido una expresión regular para que se den como válidos todos los caracteres no numéricos de todos los alfabetos, por lo que acepta los caracteres “ñ” y las tildes y diéresis.
- **Número de plazas**. Número entero entre 1 y 30.
- **Precio por plaza (€)**. Número entero entre 1 y 150.
- **Publicar oferta**. La oferta se publica si se cumplen las condiciones explicadas previamente y si el Driver no tiene un viaje ese mismo día. En caso contrario, se muestra un mensaje de error en la pantalla.

c. Registro de usuarios

Para el registro de usuarios, se han desarrollado dos páginas HTML. En una de ellas se encuentra el formulario de registro (**Register.xhtml**) y la otra (**RegisterConfirmation.xhtml**) es una pantalla que informa del éxito de la operación y ofrece la posibilidad de dirigirse a la página de inicio de sesión.

Todos los campos del formulario de registro son obligatorios y los controles de validación se realizan en la página .xhtml utilizando funcionalidades de JSF (excepto la coincidencia de contraseñas que se gestiona mediante código en el bean).

Hemos añadido la posibilidad de crear **diferentes tipos de usuarios**: conductores (Driver) y pasajeros (Passenger). En la implementación se reflejan como subclases que heredan de una superclase usuario (User), clase extra añadida. Actualmente, las subclases no contienen ninguna característica extra aparte de los métodos y atributos que heredan de User; pero decidimos implementarlo de esta forma para facilitar posibles extensiones de la aplicación en el futuro.

Como se ha mencionado, ambos tendrán acceso a las funcionalidades de consultar los viajes disponibles y sus viajes; pero solo los que sean conductores tendrán acceso a la funcionalidad de ofertar viajes. Además, los pasajeros podrán reservar viajes en la funcionalidad de consulta de viajes (más adelante se profundizará en esto).

Controles de validación

- **Nombre de usuario**. Cadena de texto formada solo por letras y números.
- **Correo electrónico**. Cadena de texto con formato válido para correo. La organización (después del @) no puede contener números o caracteres especiales. El tipo (después del punto) no puede contener números y debe tener al menos dos caracteres.
- **Contraseña**. La contraseña y la confirmación deben coincidir.

d. Inicio de sesión

Para gestionar la sesión actual se ha creado un bean (**SessionManagementBean**), que encapsula el uso de los métodos de FacesContext para ejecutar operaciones sobre el usuario que ha iniciado sesión. Para realizar el inicio de sesión, se ha implementado una página con el formulario (**Login.xhtml**) y un bean (**LoginBean**) que hace uso de las operaciones de sesión mencionadas antes.

Para realizar el inicio de sesión, debe introducirse un usuario existente (registrado) y la contraseña correcta.

Controles de validación

- Todos los campos son obligatorios.

e. Acceso a los viajes del propio usuario

Existe también la funcionalidad para todos los usuarios de consultar los viajes que han reservado (si son Passenger) u ofertado (si son Driver). La implementación en ambos casos es la misma, dado que ambos tipos de usuario heredan la lista de objetos Ride de la superclase User. Para un Passenger, esta lista representa los viajes a los que se ha apuntado; y para un Driver, representa los viajes que ha ofertado. Por tanto, se ha gestionado todo en una sola página (**MyRides.xhtml**) y su correspondiente bean (**MyRidesBean**).

Para decidir cómo se muestra la página, se hace uso del SessionManagementBean mencionado para obtener información sobre el usuario que ha iniciado sesión. Dependiendo del tipo de usuario, la página adapta el texto dinámicamente. Esto se gestiona, principalmente, utilizando el atributo “rendered” de las etiquetas HTML. Permite renderizar o no ciertos elementos según la respuesta de un método del bean.

f. Apuntarse a un viaje

Esta función está solo disponible para los usuarios registrados como Passenger y está dentro de la vista de consulta de viajes (**QueryRides.xhtml**). Es un botón que permite al usuario apuntarse a un viaje.

Funcionamiento

Una vez rellenos los campos de las ciudades de origen y destino y seleccionado un día con viajes, en la tabla con la información del viaje aparece un botón de **RESERVA** que permite apuntarse a un viaje en caso de que haya asientos disponibles en ese viaje. Una vez reservado el viaje, el botón se deshabilita para todos los viajes de ese mismo día (un usuario solo puede hacer un viaje por día). En caso de que no haya asientos libres, el botón también estará deshabilitado.

2. HIBERNATE

a. Lógica de negocio

La lógica de negocio se define en la clase **BLFacadeImplementation** dentro del paquete **modelo.businessLogic**. Hace uso de los métodos definidos en la clase de acceso a datos y se encarga de crear la base de datos e inicializarla. Sigue el patrón Singleton para que solo se pueda instanciar una instancia de lógica de negocio, evitando errores de solapamiento de datos.

Para utilizar la lógica de negocio en la aplicación, se ha creado un bean adicional (**BLFacadeImplementationBean**), que simplemente obtiene la instancia Singleton de la lógica de negocio. Los otros beans lo utilizan para acceder a los métodos. De esta forma, añadimos un nivel de abstracción entre las vistas y la implementación de la lógica de negocio, que podría variar en algún punto. Si la clase donde se encuentra la implementación cambiase, solo haría falta modificar el bean correspondiente.

b. Acceso a datos

Para gestionar la base de datos utilizando Hibernate, hemos adaptado los métodos de la clase DataAccess para que trabajen utilizando dicho framework. Ahora, la clase de acceso a datos se llama **DataAccessHibernate** y se encuentra en el paquete **modelo.dataAccess**.

La base de datos se ha configurado con MariaDB en local, no se ha añadido al proyecto. Para que funcione correctamente, debe existir una base de datos llamada **rides** en el equipo donde se ejecute la aplicación. Actualmente su configuración permite que la base de datos se reinicialice cada vez que se ejecuta la aplicación.

Mapeo de las clases Java al modelo relacional

- **User**, **Passenger** y **Driver**.

  Las clases Passenger y Driver heredan de la clase User. Para guardar estas relaciones en la base de datos, hemos optado por una estrategia “joined” (anotación **@Inheritance**): para cada subclase se creará una tabla nueva, relacionada con la

tabla de la superclase. Se ha implementado de este modo para facilitar futuras adiciones en las clases Passenger y Driver, si las hubiera.

La clase User contiene también la lista de viajes (Ride) del usuario. La relación es tipo varios a varios (anotación **@ManyToMany**). Como los User pueden ser tanto Driver como Passenger, es posible que varios usuarios tengan los mismos viajes; y que un viaje esté en la lista de varios usuarios (aunque algunos lo hayan reservado y algún otro lo haya ofertado).

- **Ride**.

  Contiene un Driver que se ha mapeado con una relación varios a uno (anotación **@ManyToOne**) porque varios viajes pueden estar asociados a un mismo conductor. También tiene una lista de Passengers, que es la lista de pasajeros apuntados al viaje. Es una relación varios a varios (anotación **@ManyToMany**) porque a un mismo viaje pueden apuntarse varios usuarios y un mismo usuario puede apuntarse a varios viajes.

  Además, también se ha añadido una estrategia de generación incremental para los números de los viajes (el número identificador) (anotación **@GeneratedValue**) y varias condiciones para que los valores de los atributos no puedan ser nulos (anotación **@Column** con nullable en false).

De esta forma, obtenemos las siguientes tablas en la base de datos.

- **driver**. Tabla de las instancias de Driver.
- **passenger**. Tabla de las instancias de Passenger.
- **user**. Tabla de las instancias de User.
- **ride**. Tabla de las instancias Ride.
- **ride_passenger**. Tabla de relación Many to Many entre Ride y Passenger (la lista de pasajeros de un viaje).
- **user_ride**. Tabla de relación Many to Many entre User y Ride (los viajes que un usuario a reservado / ofertado).

3. OTROS

a. Problema con la base de datos del proyecto original

Durante el comienzo del desarrollo, para poder probar la implementación de Java Server Faces, era necesario utilizar una base de datos para poder gestionar la información

mostrada en las vistas. Tuvimos problemas para utilizar la base de datos proporcionada en el proyecto original mediante el archivo JAR, y finalmente no conseguimos ponerla en marcha correctamente.

Solucionamos el problema creando una clase de acceso a datos temporal, llamada **DataAccessMock** (en el paquete **modelo.mocks**). Esta clase implementa los métodos del acceso a datos utilizando ArrayLists de Java. Simulan tablas de una base de datos, y los métodos recorren las listas para gestionar los objetos. De esta forma, pudimos simular la recuperación y creación de objetos en la base de datos, y probar las vistas desarrolladas con JSF antes de empezar a adaptar la clase DataAccess para que trabajase con Hibernate.

Hemos dejado el paquete mocks en el proyecto como muestra del proceso seguido durante el desarrollo.