Servigo

1.Introducción

Servigo es una aplicación que facilita la conexión entre usuarios que buscan ofrecer y adquirir servicios. A través de una interfaz intuitiva y amigable, los usuarios pueden crear perfiles, publicar sus servicios, navegar por las ofertas disponibles y contratar los servicios de otros usuarios.

Las tecnologías usadas son: Angular, Ionic, Node.js y MySQL

2. Configuración del Entorno de Desarrollo

Antes de comenzar, hay que asegurarse de tener instalado lo siguiente:

- Sistema operativo: Se recomienda utilizar un sistema operativo reciente y estable como Windows 10.
- Node.js: Descargar e instalar la última versión de Node.js desde el sitio web oficial: https://nodejs.org/en/download/package-manager
- npm: npm viene incluido con la instalación de Node.js. Verifique que npm esté instalado correctamente ejecutando el comando npm -v en la terminal.
- Git: Instale Git, un sistema de control de versiones, para clonar el repositorio de GitHub.

Instalación de Angular y Ionic

Ejecute el siguiente comando para instalar la Angular CLI globalmente:

```
npm install -g @angular/cli
```

Ejecute el siguiente comando para instalar Ionic globalmente:

```
npm install -g ionic
```

Clonar el Repositorio

Para clonar el repositorio abra su terminal y navegue hasta la ubicación donde desea clonar el repositorio. Ejecute el siguiente comando:

```
git clone https://github.com/smonmir/TFG
```

Navegue hasta la carpeta backend dentro del directorio del proyecto y ejecute el siguiente comando para instalar las dependencias de Node.js:

```
npm install
```

Navegue hasta la carpeta frontend dentro del directorio del proyecto y ejecute el siguiente comando para instalar las dependencias de la aplicación Angular e Ionic:

```
npm install
```

Variables de entorno

En el proyecto backend hay un archivo llamado .env que contiene las variables de entorno. Estas variables se utilizan para configurar aspectos como la conexión a la base de datos, los puertos del servidor y las credenciales de autenticación.

Ejemplo de variables de entorno:

DB_HOST=localhost

DB_USER= mysql_user

DB_PASSWORD=mysql_password

DB_NAME=mysql_database

PORT=3000

Las variables de entorno se cargan en el código de la aplicación utilizando la biblioteca dotenv, desde el archivo “config.js”.

Configuración de la base de datos MySQL

La configuración de la base de datos MySQL se encuentra en un archivo “config.js”.

Asegúrese de que la configuración de la base de datos en el archivo coincida con la información de su base de datos MySQL. Esto incluye parámetros como el host, el nombre de usuario, la contraseña y el nombre de la base de datos

Ejecución de la aplicación

Para iniciar el servidor backend abra una nueva terminal y navegue hasta la carpeta del proyecto backend y ejecute el siguiente comando para iniciar el servidor Node.js:

```
npm run dev
```

Para iniciar la aplicación frontend, en una terminal separada, navegue hasta la carpeta raíz de su proyecto de angular y ejecute el siguiente comando para iniciar el desarrollo de la aplicación Angular:

```
ionic serve
```

Con esta configuración básica, podrá comenzar a desarrollar la aplicación.

3.Frontend (Angular/Ionic)

Componentes principales

En el proyecto se usan principalmente Pages, que son componentes de página completos que encapsulan HTML, CSS y JavaScript.En la carpeta llamada “pages” están almacenados todas las Pages de la aplicación. Las principales pages son:

-detalle-servicio

-home

-login

-mis-servicios

-pago

-pedidos

-perfil

-registro

-tabs

Para crear un nuevo componente Ionic Page Ejecuta el siguiente comando en la terminal, ubicándote en la carpeta pages, reemplazando “mi-nueva-page” con el nombre deseado:

```
ionic g page mi-nueva-page
```

El comando anterior creará una carpeta y archivos para tu nuevo componente. Dentro de la carpeta encontrarás principalmente:

mi-nueva-page.page.html: Contiene el código HTML para la vista del componente.

mi-nueva-page.page.scss: Contiene el código CSS para el estilo del componente.

mi-nueva-page.page.ts: Contiene la lógica de TypeScript para el componente.

Modelos

Se utilizan modelos para representar los datos que se manejan en la aplicación. Los modelos son clases que definen la estructura de los datos y proporcionan métodos para acceder y manipular sus propiedades. Los modelos de la aplicación están ubicados en la carpeta ../clases/modelos. Para crear un nuevo modelo, ejecuta el siguiente comando en la terminal, reemplazando mi-nuevo-modelo con el nombre deseado:

```
ionic g class mi-nuevo-modelo
```

Servicios

Los servicios en Angular/Ionic son clases reutilizables que proporcionan funcionalidades específicas para la aplicación. Por ejemplo en el contexto de la comunicación con el backend, se utiliza el servicio HttpClientService para realizar peticiones HTTP a la API del backend, obtener datos y procesarlos antes de pasarlos a los componentes. Los servicios de la aplicación están ubicados en la carpeta ../clases/servicios

Para crear un nuevo servicio, ejecuta el siguiente comando en tu terminal, reemplazando mi-nuevo-servicio con el nombre deseado:

```
ionic g service mi-nuevo-servicio
```

Para utilizar un servicio, se debe de inyectar en el constructor de los componentes que lo necesiten, por ejemplo:

```
export class MiNuevoComponente {
    constructor(private servicio: mi-nuevo-servicio) { }.
}
```

Luego puedes llamar a los métodos del servicio desde el componente.

Rutas

Las rutas principales definen los caminos URL que corresponden a las diferentes páginas de la aplicación. Se configuran en el módulo principal de Angular app-routing.module.ts y, como se hace uso de tabs, también se configuran en el modulo tbs-routing.module.ts en la carpeta tabs.

Para cada Ionic Page que se desee incluir en la aplicación, se debe definir una ruta en el array routes ya sea en modulo principal de Angular o en el modulo de tabs. Al hacerlo, se debe especificar el componente de página correspondiente.

Guards

En la aplicación se usan guards, clases reutilizables que se utilizan para proteger las rutas de la aplicación y controlar el acceso a determinadas funcionalidades. En este caso, se utilizan dos guards: rol y auth, ubicados en la carpeta guard.

rol guard: Este guard se utiliza para verificar si el usuario actual tiene un rol específico que le permite acceder a una ruta.

auth guard: Este guard se utiliza para verificar si el usuario está autenticado en la aplicación.

4.Backend (Node.js)

Estructura del proyecto

El proyecto backend está organizado en las siguientes carpetas:
- app: Contiene el archivo principal app.js que configura la aplicación Express y monta las rutas de la API.
- controllers: Contiene los archivos de controladores que implementan la lógica de negocio para cada modelo de la API.
- db: Contiene los archivos de configuración y conexión a la base de datos Sequelize.
- middleware: Contiene los archivos de middleware que se aplican a las rutas de la API.
- models: Contiene los modelos Sequelize que definen la estructura de las tablas en la base de datos.
- routes: Contiene los archivos de rutas que definen los endpoints para cada modelo de la API.

Flujo de ejecución

El servidor se inicia ejecutando el archivo index.js. Este archivo importa el archivo app.js que configura la aplicación Express, conecta a la base de datos Sequelize y monta las rutas de la API.
Las rutas se definen en archivos individuales dentro de la carpeta routes. Cada archivo de ruta importa las funciones controladoras correspondientes del directorio controllers.
Cuando se recibe una solicitud a la API, se enruta al archivo de ruta correspondiente según el endpoint. La función controladora correspondiente se ejecuta para manejar la solicitud y devolver una respuesta.

API Endpoints

La API se consume mediante solicitudes HTTP estándar. Se pueden utilizar herramientas como Postman o navegadores web con extensiones para realizar pruebas y consultar la API.

A continuación una lista de todos los endpoint disponibles:

Para el modelo de Usuraio:

-GET /api/usuario

-GET /api/usuario/me

-POST /api/usuario/login

- POST /api/usuario

- POST /api/usuario/código/enviarCodigo

- POST /api/usuario/verificarCodigo
- PATCH /api/usuario/:id
- DELETE /api/usuario/:id

Para el modelo Servicio:

- GET /api/servicio
- GET /api/servicio/paginados
-GET /api/servicio/usuario/:usuarioId
-POST /api/servicio
-PATCH /api/servicio/:id
-DELETE /api/servicio/:id

Para el modelo Rol:

-GET /api/rol
-GET /api/rol/:id
-POST /api/rol
-PATCH /api/rol/:id
-DELETE /api/rol/:id

Para el modelo Pedido:

-GET /api/pedido
-GET /api/pedio/:id
- GET /api/pedio/usuario/:usuarioId
-POST /api/pedido
-PATCH /api/pedido/:id
-DELETE /api/pedido/:id

Para el modelo Estado:

-GET /api/estado
-GET /api/estado/:id
-POST /api/estado
-PATCH /api/estado/:id
-DELETE /api/estado/:id

Para el modelo Valoracion:

-GET /api/valoracion

-GET /api/valoracion/:servicioId

-POST /api/valoracion

-PATCH /api/valoracion/:id

-DELETE /api/valoracion/:id

Controladores

Los controladores son los encargados de procesar las solicitudes HTTP entrantes para cada modelo de la API, interactúan con la base de datos Sequelize y generan las respuestas correspondientes. Cada función controladora maneja un endpoint específico (GET, POST, PATCH, DELETE) definido en la ruta correspondiente.

La función recibe como argumentos el objeto req (request) y el objeto res (response) de Express, procesa la solicitud e interactúa con el modelo Sequelize correspondiente utilizando métodos como findOne, findAll, create, update, destroy, etc. Se genera una respuesta con el código de estado HTTP apropiado y el contenido de la respuesta en formato JSON.

Modelos

Los modelos representan la estructura de las tablas en la base de datos y definen las propiedades de los objetos que se manejan en la aplicación.

Un archivo de modelo contiene la definición de un modelo usando la función define de Sequelize.

Los modelos existentes son: Usuario, Servicio, Pedido, Valoracion, Rol y Estado.

A continuación se detalla la definición de cada modelo utilizado en la API:

Usuario (usuarioModel.js):

Representa la tabla usuarios.

Atributos:

-id: Identificador único del usuario (entero autoincrementable, clave primaria).

-nombre: Nombre del usuario (cadena de texto obligatoria).

-email: Correo electrónico del usuario (cadena de texto única obligatoria, validada como email).

-contrasena: Contraseña del usuario (cadena de texto obligatoria).

-telefono: Teléfono del usuario (cadena de texto).

-rol_id: Identificador del rol asociado al usuario (entero obligatorio, con referencia a la tabla roles).

Relación:

belongsTo(Rol): Un usuario pertenece a un único rol.

Servicio (servicioModel.js):

Representa la tabla servicios.

Atributos:

-id: Identificador único del servicio (entero autoincrementable, clave primaria).

-nombre: Nombre del servicio (cadena de texto obligatoria).

-descripcion: Descripción del servicio (cadena de texto obligatoria).

-precio: Precio del servicio (número decimal con dos decimales, obligatorio).

-imagen: URL de la imagen del servicio (cadena de texto obligatoria).

-usuario_id: Identificador del usuario que ofrece el servicio (entero obligatorio, con referencia a la tabla usuarios).

Relación:

belongsTo(Usuario): Un servicio pertenece a un único usuario.

Rol (rolModel.js):

Representa la tabla roles.

Atributos:

-id: Identificador único del rol (entero autoincrementable, clave primaria).

-nombre: Nombre del rol (cadena de texto obligatoria).

-descripcion: Descripción del rol (cadena de texto obligatoria).

Pedido (pedidoModel.js):

Representa la tabla pedidos.

Atributos:

-id: Identificador único del pedido (entero autoincrementable, clave primaria).

-fecha: Fecha del pedido (fecha obligatoria).

-precio: Precio total del pedido (número decimal con dos decimales, obligatorio).

-direccion: Dirección de entrega del pedido (cadena de texto).

-usuario_id: Identificador del usuario que realiza el pedido (entero obligatorio, con referencia a la tabla usuarios).

-servicio_id: Identificador del servicio asociado al pedido (entero obligatorio, con referencia a la tabla servicios).

-estado_id: Identificador del estado actual del pedido (entero obligatorio, con referencia a la tabla estados).

Relaciones:

belongsTo(Usuario): Un pedido pertenece a un usuario.

belongsTo(Servicio): Un pedido pertenece a un servicio.

belongsTo(Estado): Un pedido tiene un estado asociado.

Estado (estadoModel.js):

Representa la tabla estados.

Atributos:

-id: Identificador único del estado (entero autoincrementable, clave primaria).

-nombre: Nombre del estado del pedido (cadena de texto obligatoria).

-descripcion: Descripción del estado del pedido (cadena de texto obligatoria).

Valoracion (valoracionModel.js):

Representa la tabla valoraciones.

Atributos:

-id: Identificador único de la valoración (entero autoincrementable, clave primaria).

-puntuacion: Puntuación otorgada al servicio por el usuario (entero obligatorio, entre 1 y 10).

-comentario: Comentario del usuario sobre el servicio (texto obligatorio).

-usuario_id: Identificador del usuario que realiza la valoración (entero obligatorio, con referencia a la tabla usuarios).

-servicio_id: Identificador del servicio valorado (entero obligatorio, con referencia a la tabla servicios).

Relaciones:

belongsTo(Usuario): Una valoración pertenece a un usuario.

belongsTo(Servicio): Una valoración pertenece a un servicio.

Middleware

El middleware es ejecutado durante una solicitud HTTP que puede realizar tareas como la autenticación, la carga de archivos y la modificación de la solicitud y la respuesta.

El backend contiene una carpeta middleware con archivos que definen funciones middleware personalizadas para la API. En este caso, la carpeta contiene dos archivos:

authMiddleware.js: Implementa la autenticación basada en token JWT.

uploadMiddleware.js: Proporciona funcionalidad para la carga de archivos.

El archivo authMiddleware.js define un middleware llamado authMiddleware que verifica la autenticación de un usuario en las solicitudes entrantes.

Se ejecuta antes de continuar con la solicitud, intenta verificar el token de autenticación del usuario y si la validación es exitosa, adjunta el usuario encontrado a la solicitud (req.user).

El archivo uploadMiddleware.js define un middleware llamado upload que utiliza la librería multer para gestionar la carga de archivos

Ejemplo de uso:

const { **authMiddleware** } = require('../middleware/authMiddleware');

router.get('/usuarios/me', **authMiddleware**, controller.getUsuarioAutenticado);

const **upload** = require('../middleware/uploadMiddleware');

router.post('/servicios', **upload**.single('imagen'), controller.crearServicio);

El primer ejemplo utiliza authMiddleware para verificar la autenticación antes de llamar al controlador getUsuarioAutenticado.

El segundo ejemplo utiliza upload.single('imagen') para manejar la carga de un archivo llamado "imagen" antes de llamar al controlador crearServicio.

5.Base de Datos (MySQL)

Tablas de la Base de Datos:

usuarios:

-ID: Identificador único del usuario (entero autoincrementable, clave primaria).

-nombre: Nombre del usuario (cadena de texto obligatoria).

-email: Correo electrónico del usuario (cadena de texto única obligatoria, validada como email).

-contrasena: Contraseña del usuario (cadena de texto obligatoria).

-telefono: Teléfono del usuario (cadena de texto).

-rol_id: Identificador del rol asociado al usuario (entero obligatorio, con referencia a la tabla roles).

servicios:

-ID: Identificador único del servicio (entero autoincrementable, clave primaria).

-nombre: Nombre del servicio (cadena de texto obligatoria).

-descripcion: Descripción del servicio (cadena de texto obligatoria).

-precio: Precio del servicio (número decimal con dos decimales, obligatorio).

-imagen: URL de la imagen del servicio (cadena de texto obligatoria).

-usuario_id: Identificador del usuario que ofrece el servicio (entero obligatorio, con referencia a la tabla usuarios).

roles:

-ID: Identificador único del rol (entero autoincrementable, clave primaria).

-nombre: Nombre del rol (cadena de texto obligatoria).

-descripcion: Descripción del rol (cadena de texto obligatoria).

pedidos:

-ID: Identificador único del pedido (entero autoincrementable, clave primaria).

-fecha: Fecha del pedido (fecha obligatoria).

-precio: Precio total del pedido (número decimal con dos decimales, obligatorio).

-direccion: Dirección de entrega del pedido (cadena de texto).

-usuario_id: Identificador del usuario que realiza el pedido (entero obligatorio, con referencia a la tabla usuarios).

-servicio_id: Identificador del servicio asociado al pedido (entero obligatorio, con referencia a la tabla servicios).

-estado_id: Identificador del estado actual del pedido (entero obligatorio, con referencia a la tabla estados).

estados:

-ID: Identificador único del estado (entero autoincrementable, clave primaria).

-nombre: Nombre del estado del pedido (cadena de texto obligatoria).

-descripcion: Descripción del estado del pedido (cadena de texto obligatoria).

valoraciones:

-ID: Identificador único de la valoración (entero autoincrementable, clave primaria).

-puntuacion: Puntuación otorgada al servicio por el usuario (entero obligatorio, entre 1 y 10).

-comentario: Comentario del usuario sobre el servicio (texto obligatorio).

-usuario_id: Identificador del usuario que realiza la valoración (entero obligatorio, con referencia a la tabla usuarios).

-servicio_id: Identificador del servicio valorado (entero obligatorio, con referencia a la tabla servicios).

Relaciones entre Tablas:

Un usuario (usuarios) puede pertenecer a un único rol (roles).

La tabla usuarios tiene una columna rol_id que hace referencia a la clave primaria (ID) de la tabla roles.

Esta relación se implementa utilizando la opción references al definir el modelo Usuario en Sequelize.

Un servicio (servicios) pertenece a un único usuario (usuarios).

La tabla servicios tiene una columna usuario_id que hace referencia a la clave primaria (ID) de la tabla usuarios.

Esta relación se implementa utilizando la opción references al definir el modelo Servicio en Sequelize.

Un pedido (pedidos) pertenece a un único usuario (usuarios).

La tabla pedidos tiene una columna usuario_id que hace referencia a la clave primaria (ID) de la tabla usuarios.

Esta relación se implementa utilizando la opción references al definir el modelo Pedido en Sequelize.

Un pedido (pedidos) pertenece a un único servicio (servicios).

La tabla pedidos tiene una columna servicio_id que hace referencia a la clave primaria (ID) de la tabla servicios.

Esta relación se implementa utilizando la opción references al definir el modelo Pedido en Sequelize.

Un pedido (pedidos) tiene un único estado asociado (estados).

La tabla pedidos tiene una columna estado_id que hace referencia a la clave primaria (ID) de la tabla estados.

Esta relación se implementa utilizando la opción references al definir el modelo Pedido en Sequelize.

Una valoración (valoraciones) pertenece a un único usuario (usuarios).

La tabla valoraciones tiene una columna usuario_id que hace referencia a la clave primaria (ID) de la tabla usuarios.

Esta relación se implementa utilizando la opción references al definir el modelo Valoracion en Sequelize.

Una valoración (valoraciones) pertenece a un único servicio (servicios).

La tabla valoraciones tiene una columna servicio_id que hace referencia a la clave primaria (ID) de la tabla servicios.