Sistema de Gestión de Mantenimiento de la Universidad Técnica Nacional de Venado Tuerto

Documentación de Usuario
Desarrollado por: Federico Braga, Jhon Escobar, Ignacio Cifuentes y Santiago Agonil
Contacto de soporte:

Propósito del Proyecto

El Sistema de Gestión de Mantenimiento Universitario fue desarrollado para mejorar la eficiencia en la gestión de órdenes de trabajo dentro del campus universitario. Este sistema facilita la asignación, supervisión y resolución de tareas relacionadas con el mantenimiento de la infraestructura, permitiendo una mejor comunicación entre los administradores y los operarios.

Objetivo General

Diseñar e implementar una solución tecnológica que permita gestionar de manera integral las órdenes de trabajo y los datos asociados al mantenimiento de la universidad, mejorando la organización y el tiempo de respuesta ante las solicitudes de reparación y mantenimiento.

Objetivos Específicos

1. Desarrollar un módulo de administración para la creación, asignación y supervisión de órdenes de trabajo.
2. Implementar un módulo de operarios que permita recibir y actualizar las órdenes asignadas.
3. Incorporar funcionalidades para el manejo de datos relacionados con edificios, activos y personal.
4. Garantizar la seguridad y privacidad mediante un sistema de autenticación con roles diferenciados.
5. Diseñar una interfaz responsiva que facilite el uso del sistema en distintos dispositivos.

Alcance

El proyecto abarca:

1. Gestión de Órdenes de Trabajo: Creación, asignación, seguimiento y actualización del estado de las tareas.
2. Manejo de Bases de Datos: Gestión de activos, ubicaciones, operarios y otros datos relevantes al mantenimiento.
3. Roles de Usuario:
    - Administradores: Usuarios con permisos para gestionar órdenes y modificar datos del sistema.

    - Operarios: Usuarios encargados de recibir y ejecutar órdenes de trabajo asignadas.
4. Autenticación de Usuarios: Sistema de registro y login con permisos específicos según el rol.
5. Interfaz de Usuario: Diseño claro y sencillo, optimizado para computadoras de escritorio y dispositivos móviles.

Tecnologías Utilizadas

- Frontend: Angular (TypeScript, HTML, CSS).
- Backend: Node.js (Express).
- Base de Datos: MySQL.
- Control de versiones: Git (GitHub).
- Herramientas de Desarrollo: Visual Studio Code, ThunderClient.

Estructura del Documento

1. Introducción: Contexto y objetivos del proyecto.
2. Estructuras del proyecto: Arquitectura del sistema.
3. Manual del Usuario: Instrucciones detalladas para administradores y operarios.
4. Diagrama de Arquitectura: Representación gráfica del sistema.
5. Manual Técnico: Explicación de componentes clave del código, configuración del entorno y despliegue.
6. Conclusiones y Recomendaciones: Resumen de logros y posibles mejoras.

Arquitectura General

El sistema sigue una arquitectura cliente-servidor:

- **Frontend:** Desarrollado en Angular, encargado de la interfaz de usuario y la comunicación con el backend.
- **Backend:** Implementado en Node.js con Express, que maneja la lógica de negocio y la interacción con la base de datos.
- **Base de Datos:** MySQL, para el almacenamiento persistente de datos del sistema.

Dependencias y Librerías del Proyecto

En este apartado se encuentran todas las dependencias y librerías necesarias para el correcto funcionamiento del proyecto.

1. Dependencias de Angular

Estas son las dependencias principales del framework Angular utilizadas en el proyecto:

- `@angular-devkit/build-angular@17.3.11` → Herramientas de compilación y desarrollo para Angular.
- `@angular/animations@17.3.12` → Soporte para animaciones en Angular.
- `@angular/cli@17.3.11` → Interfaz de línea de comandos de Angular.
- `@angular/common@17.3.12` → Módulos y funciones comunes en Angular.
- `@angular/compiler-cli@17.3.12` → Compilador de Angular para TypeScript.
- `@angular/compiler@17.3.12` → Permite la compilación de plantillas de Angular.
- `@angular/core@17.3.12` → Núcleo del framework Angular.
- `@angular/forms@17.3.12` → Módulo para formularios en Angular.
- `@angular/platform-browser-dynamic@17.3.12` → Permite la ejecución de aplicaciones Angular en navegadores.
- `@angular/platform-browser@17.3.12` → Módulo para interactuar con la plataforma del navegador.
- `@angular/router@17.3.12` → Módulo de enrutamiento en Angular.

2. Dependencias de Estilos y Diseño

Estas librerías ayudan a mejorar la apariencia del proyecto:

- `@popperjs/core@2.11.8` → Biblioteca para manejar elementos emergentes como tooltips y popovers.
- `bootstrap@5.3.3` → Framework de CSS para diseño responsivo y componentes.
- `bootstrap-icons@1.11.3` → Conjunto de íconos vectoriales para Bootstrap.

3. Dependencias de Utilidad y Funcionalidades Adicionales

Librerías que agregan funcionalidades específicas al proyecto:

- `ngx-cookie-service@17.1.0` → Servicio para manejar cookies en Angular.
- `ngx-toastr@17.0.2` → Notificaciones tipo toast para Angular.
- `qrcode@1.5.4` → Generación de códigos QR.
- `@types/qrcode@1.5.5` → Definiciones de tipo para la librería de códigos QR.
- `rxjs@7.8.1` → Programación reactiva en Angular.
- `tslib@2.8.1` → Biblioteca de compatibilidad con TypeScript.
- `zone.js@0.14.10` → Permite la gestión de cambios en Angular.

4. Dependencias de Pruebas y Desarrollo

Librerías utilizadas para pruebas unitarias y herramientas de desarrollo:

- `@types/jasmine@5.1.4` → Definiciones de tipo para Jasmine.
- `jasmine-core@5.1.2` → Framework de pruebas para JavaScript.
- `karma@6.4.4` → Ejecutor de pruebas para JavaScript.

- `karma-chrome-launcher@3.2.0` → Ejecuta pruebas en Chrome mediante Karma.
- `karma-coverage@2.2.1` → Genera reportes de cobertura de código.
- `karma-jasmine@5.1.0` → Adaptador de Jasmine para Karma.
- `karma-jasmine-html-reporter@2.1.0` → Reporte HTML de pruebas en Karma.

- @compodoc/compodoc@1.1.26 → Documentación del frontend.

5. Dependencias de TypeScript

- `typescript@5.4.5` → Compilador de TypeScript utilizado en el proyecto.

6. Dependencias Específicas del Proyecto

- `gm-frvt@0.0.0` → Dependencia local del proyecto.

Compodoc

Compodoc es una herramienta utilizada para generar documentación detallada del código en proyectos Angular. Se encuentra en la carpeta `documentation` dentro de `frontend` y permite visualizar información sobre módulos, componentes, servicios y más.

Uso básico:

Para generar la documentación, ejecutar:

npx compodoc -p tsconfig.json -s

Esto iniciará un servidor local con la documentación accesible en el navegador.

Estructura del Proyecto Frontend

El frontend del sistema está desarrollado en Angular, y su organización sigue una estructura modular que facilita la escalabilidad y el mantenimiento del código. A continuación, se describe la disposición de carpetas y archivos clave:

Directorio `src/`

El directorio raíz `src/` contiene los elementos esenciales del proyecto Angular. Su contenido incluye:

- **`app/`**: Carpeta principal del frontend, donde se organiza la lógica del proyecto.
- **`assets/`**: Recursos estáticos como imágenes, íconos, y estilos adicionales.
- **`environments/`**: Configuración de entornos (`development` y `production`) para manejar URLs y parámetros según el entorno.
- **`index.html`**: Página principal que carga la aplicación Angular en el navegador.
- **`styles.css`**: Archivo de estilos globales aplicados a toda la aplicación.
- **`angular.json`**: Archivo de configuración del framework Angular, utilizado para definir rutas, compilaciones y configuraciones específicas.

Directorio `app/`

Dentro de la carpeta `app/` se organiza toda la lógica principal de la aplicación. Sus subcarpetas y propósitos son los siguientes:

1. `components/`

- Contiene componentes reutilizables que representan elementos visuales específicos.
- Ejemplo de componentes:
  - **`header/`**: Encabezado común en toda la aplicación.
  - **`footer/`**: Pie de página con enlaces o información estática.

2. `guards/`

- Define las **rutas protegidas** y las reglas de acceso según el rol del usuario.
- Ejemplo:
  - **`role.guard.ts`**: Verifica si el usuario está autenticado antes de permitir el acceso a rutas protegidas.

3. `interceptors/`

- Contiene interceptores para modificar o controlar las solicitudes HTTP.
- Ejemplo:
  - **`manejoDeErrores.interceptor.ts`**: Añade el token de autenticación en las cabeceras de cada solicitud al backend.

4. `interfaces/`

- Define las **interfaces** que estructuran los datos utilizados en la aplicación.
- Ejemplo:
  - **`usuario.ts`**: Interfaz para los datos del usuario.
  - **`ubicacion.ts`**: Interfaz para las ubicaciones.
  - **`tarea.ts`**: Interfaz para las tareas.

5. `pages/`

- Contiene los **componentes de páginas completas**, cada una representando una sección importante del sistema.
- Ejemplo de páginas:
    - `landing/`: Página inicial o de bienvenida del sistema.
    - `login/`: Página para iniciar sesión.
    - `registro/`: Página para registrar nuevos usuarios.

Fuera de `app/`

1. `assets/`

- Carpeta para recursos estáticos, como imágenes y archivos externos.
- Ejemplo: Íconos utilizados en la interfaz o imágenes del sistema.

2. `environments/`

- Contiene configuraciones específicas para cada entorno (desarrollo y producción).
- Ejemplo:
    - `environment.ts`: Configuración para desarrollo (URLs de API locales).
    - `environment.development.ts`: Configuración para producción (URLs finales del servidor).

3. `services/`

- Contiene los **servicios** que manejan la lógica de negocio y la interacción con el backend mediante HTTP.
- Ejemplo:
    - `auth.service.ts`: Servicio para autenticación de usuarios.
    - `orden-trabajo.service.ts`: Servicio para gestionar órdenes de trabajo.

`app.module.ts`

El archivo `app.module.ts` es el módulo principal de la aplicación Angular. Su objetivo es configurar y registrar todos los módulos, componentes, servicios e interceptores necesarios para el correcto funcionamiento del frontend.

Estructura del Archivo

1. **Importaciones**
2. **Decorador `@NgModule`**
3. **Exportación de la Clase `AppModule`**

1. Importaciones
La sección de importaciones contiene las dependencias necesarias, organizadas de la siguiente manera:

Módulos del núcleo de Angular:

- `BrowserModule`: Servicios esenciales para que la aplicación funcione en el navegador.
- `AppRoutingModule`: Configuración del sistema de rutas de la aplicación.
- `FormsModule`: Soporte para formularios reactivos y template-driven.
- `BrowserAnimationsModule`: Permite el uso de animaciones en Angular.
- `HttpClientModule`: Facilita las solicitudes HTTP para la comunicación con el backend.

Librerías externas:

- `ngx-cookie-service`: Manejo de cookies.
- `ngx-toastr`: Generación de notificaciones emergentes (toasts).

Módulos personalizados del proyecto:

- `FooterModule`: Manejo del pie de página.
- `HeaderModule` y `HeaderLoggedModule`: Para las cabeceras estándar y autenticadas.
- `LandingModule`: Contiene los componentes relacionados con la página principal informativa.
- `FormulariosModule`: Módulo que agrupa componentes relacionados con formularios específicos.
- `OrdenTrabajoModule`: Módulo para gestionar las funcionalidades relacionadas con las órdenes de trabajo.

Componentes individuales:
Incluye varios componentes utilizados en diferentes páginas o secciones de la aplicación, como:

- Componentes de páginas (`LoginComponent`, `RegistroComponent`, `NotFoundComponent`, etc.).
- Componentes relacionados con órdenes de trabajo (`CuerpoComponent`, `FechasComponent`, `SelectoresComponent`, y `TareasComponent`).

Interceptors:

- `manejoErroresInterceptor`: Un interceptor que maneja errores HTTP globalmente.

2. Decorador @NgModule
El decorador `@NgModule` organiza los elementos principales del módulo a través de las siguientes propiedades:

declarations:
Lista los componentes que forman parte de este módulo. Algunos de los componentes declarados son:

- `AppComponent`: Componente raíz de la aplicación.
- `LoginComponent`: Página de inicio de sesión.
- `RegistroComponent`: Página de registro.
- `NotFoundComponent`: Página para rutas no existentes.
- Componentes específicos de funcionalidad: `CuerpoComponent`, `FechasComponent`, `SelectoresComponent`, y `TareasComponent`.

imports:
Lista de módulos importados en este módulo. Incluye tanto los módulos propios de Angular como los módulos personalizados y externos:

- `BrowserModule` y `AppRoutingModule` para la configuración básica.
- `LandingModule`, `HeaderModule`, y `FooterModule` para encapsular vistas reutilizables.
- `HttpClientModule` para el manejo de solicitudes HTTP.
- `BrowserAnimationsModule` y `ToastrModule.forRoot()` para soporte de animaciones y notificaciones.
- `OrdenTrabajoModule` para la gestión de las órdenes de trabajo.

providers:
Define los servicios globales y los interceptores:

- `CookieService`: Servicio para manejar cookies.
- `provideHttpClient`: Incluye el interceptor `manejoErroresInterceptor` para el manejo centralizado de errores HTTP.

bootstrap:
Especifica el componente raíz (`AppComponent`) que Angular inicializa al arrancar la aplicación.

Clase `AppModule`

La clase `AppModule` exportada actúa como el punto de inicio del frontend. Contiene la configuración general que ensambla todos los elementos y hace que la aplicación funcione correctamente.

`app.routing.module.ts`

El `AppRoutingModule` es el módulo encargado de gestionar las rutas de la aplicación Angular. Define las diferentes rutas disponibles y establece la relación entre las rutas y los componentes correspondientes.

Importaciones

- `RouterModule` y `Routes`: Proveen la funcionalidad de enrutamiento de Angular.
- Componentes de formularios (`FormActivoComponent`, `FormEdificioComponent`, etc.).
- Componentes de páginas (`LandingPageComponent`, `LoginComponent`, `RegistroComponent`, etc.).
- `roleGuard`: Un guardia que restringe el acceso a ciertas rutas según el rol del usuario.

Configuración de Rutas

- `''`: Ruta raíz, que muestra la `LandingPageComponent`.
- `login`: Muestra el `LoginComponent`.
- `registro`: Muestra el `RegistroComponent`.
- `ordenTrabajo`: Renderiza el componente `CuerpoComponent`, que maneja la vista de órdenes de trabajo.
- `formOT`, `formActivo`, `formEdificio`, etc.: Corresponden a formularios de distintas entidades dentro de la aplicación.
- `inicioOperario` e `inicioAdmin`: Requieren autenticación y un rol específico (`operario` o `admin` respectivamente), validado por `roleGuard`.
- `404`: Muestra la página `NotFoundComponent` cuando la ruta solicitada no existe.

Módulo de Rutas

- Se usa `RouterModule.forRoot(routes)` para registrar las rutas principales de la aplicación.
- `RouterModule` es exportado para ser utilizado en otros módulos.

Funcionamiento

Este módulo permite la navegación dentro de la aplicación, definiendo qué componente se carga según la URL ingresada. Además, protege ciertas rutas con `roleGuard` para restringir el acceso según el rol del usuario.

Carpeta `components`

La carpeta `components` contiene todos los componentes reutilizables y módulos necesarios para la interfaz de usuario de la aplicación. Estos componentes se dividen en varias subcarpetas según su función y estilo. A continuación se describen las principales subcarpetas y archivos de `components`:

`footer/`

El componente `footer/` se encarga de mostrar el pie de página en la interfaz de la aplicación, proporcionando información relevante como el logo de la institución, la dirección, los números de contacto y los derechos de autor. Está compuesto por tres subcomponentes: `footer`, `footer-data` y `footer-logo`. Al mismo tiempo incluye el archivo `footer.module.ts`.

- **`footer`**: Define la estructura general del pie de página, incluyendo la sección con el logo de la institución y los datos de contacto.
- **`footer-data`**: Muestra la dirección y los números de teléfono de contacto.
- **`footer-logo`**: Muestra el logo de la universidad.

Los subcomponentes están diseñados para mantener una estructura modular, de forma que cada parte del pie de página puede ser actualizada de forma independiente.

`footer.module.ts`

Este módulo declara y exporta los siguientes componentes:

- **`FooterComponent`**
- **`FooterDataComponent`**
- **`FooterLogoComponent`**

También importa **`CommonModule`** para habilitar las funcionalidades comunes de Angular.

`formularios/`

La carpeta `formularios` contiene componentes que permiten a los administradores interactuar con diferentes tablas de la base de datos a través de formularios. Cada componente tiene funciones específicas para crear, editar, eliminar y visualizar registros en su respectiva tabla.

form-activo

- **Descripción**: Este formulario permite gestionar los activos en el sistema (crear, editar, eliminar y visualizar). Se conecta a un servicio que maneja las operaciones CRUD en la base de datos de los activos.
- **Archivos**:
    - **form-activo.component.html**:
        - Contiene los campos para ingresar los datos del activo como nombre, abreviación, existencia, entre otros.

        - Incluye botones para crear o editar el activo.
    - **form-activo.component.ts**:
        - **ngOnInit()**: Se encarga de inicializar el formulario y cargar los datos si es necesario (cuando se edita un activo).
        - **guardarActivo()**: Método para guardar el activo, que decide si crear o actualizar según el caso.
        - **eliminarActivo()**: Método para eliminar un activo.
    - **form-activo.component.css**: Estilos para la interfaz del formulario.

form-edificio

- **Descripción**: Permite gestionar los edificios (crear, editar, eliminar y visualizar). Similar al formulario de activos, interactúa con un servicio para obtener y guardar los datos de los edificios.
- **Archivos**:
    - **form-edificio.component.html**:
        - Contiene los campos para definir los datos del edificio, como nombre, calle y otros detalles relevantes.
        - Botones para realizar las operaciones de crear y editar.
    - **form-edificio.component.ts**:
        - **ngOnInit()**: Inicializa el formulario y carga los datos de un edificio si es necesario (en caso de edición).
        - **guardarEdificio()**: Método para guardar el edificio (crear o actualizar según el caso).
        - **eliminarEdificio()**: Elimina un edificio de la base de datos.
    - **form-edificio.component.css**: Estilos del formulario.

form-labor

- **Descripción**: Gestiona las labores o tareas asignadas a los operarios. Permite realizar las operaciones CRUD en las labores.
- **Archivos**:
    - **form-labor.component.html**:
        - Formulario con campos para ingresar los detalles de una labor.
        - Incluye botones de acción para crear, editar o eliminar.
    - **form-labor.component.ts**:
        - **ngOnInit()**: Inicializa el formulario, y en caso de edición, carga los datos existentes.
        - **guardarLabor()**: Método para guardar o actualizar la labor.
        - **eliminarLabor()**: Elimina la labor seleccionada.
    - **form-labor.component.css**: Estilos para el formulario de labores.

form-ot

- **Descripción**: Este formulario permite gestionar las órdenes de trabajo en el sistema. Incluye los campos necesarios para crear y editar las órdenes, y permite visualizarlas.
- **Archivos**:

    - **form-ot.component.html**:
        - Contiene campos para definir los detalles de la orden de trabajo, como el activo relacionado, operarios asignados, y otros datos relevantes.
        - Botones para crear o editar la orden de trabajo.
    - **form-ot.component.ts**:
        - **ngOnInit()**: Carga los datos necesarios para el formulario (si se está editando una orden).
        - **guardarOT()**: Guarda la orden de trabajo, bien sea creando una nueva o actualizando una existente.
        - **eliminarOT()**: Elimina una orden de trabajo del sistema.
    - **form-ot.component.css**: Estilos visuales del formulario.

form-piso

- **Descripción**: Permite gestionar los pisos dentro de los edificios. Permite crear, editar y eliminar pisos.
- **Archivos**:
    - **form-piso.component.html**:
        - Campos para definir el nombre del piso, el edificio al que pertenece, y otros detalles necesarios.
        - Botones de acción para gestionar los pisos.
    - **form-piso.component.ts**:
        - **ngOnInit()**: Inicializa el formulario y carga los datos del piso si se está editando.
        - **guardarPiso()**: Método para crear o actualizar un piso.
        - **eliminarPiso()**: Elimina un piso del sistema.
    - **form-piso.component.css**: Estilos visuales del formulario.

form-sector

- **Descripción**: Este formulario gestiona los sectores dentro de los pisos. Permite agregar, modificar o eliminar sectores.
- **Archivos**:
    - **form-sector.component.html**:
        - Contiene los campos para definir el nombre del sector, el piso al que pertenece, entre otros.
    - **form-sector.component.ts**:
        - **ngOnInit()**: Inicializa el formulario y carga los datos del sector si se está editando.
        - **guardarSector()**: Método para guardar el sector (crear o actualizar).
        - **eliminarSector()**: Elimina el sector seleccionado.
    - **form-sector.component.css**: Estilos visuales para el formulario de sectores.

form-tareas

- **Descripción**: Gestiona las tareas disponibles que pueden ser asignadas a activos u operarios. Permite crear, editar y eliminar tareas.

- **Archivos**:
    - **form-tareas.component.html**:
        - Contiene los campos necesarios para definir una tarea, como el nombre, la descripción, etc.
    - **form-tareas.component.ts**:
        - **ngOnInit()**: Inicializa el formulario y carga los datos de la tarea si se está editando.
        - **guardarTarea()**: Método para guardar o actualizar la tarea.
        - **eliminarTarea()**: Elimina la tarea seleccionada.
    - **form-tareas.component.css**: Estilos para el formulario de tareas.

form-ubicacion

- **Descripción**: Permite gestionar las ubicaciones de los activos dentro de los edificios. Asegura que los activos sean correctamente asignados a sus ubicaciones.
- **Archivos**:
    - **form-ubicacion.component.html**:
        - Contiene los campos necesarios para definir la ubicación de un activo (por ejemplo, edificio, piso, sector).
    - **form-ubicacion.component.ts**:
        - **ngOnInit()**: Inicializa el formulario y carga los datos de ubicación si se está editando.
        - **guardarUbicacion()**: Método para crear o actualizar la ubicación.
        - **eliminarUbicacion()**: Elimina la ubicación seleccionada.
    - **form-ubicacion.component.css**: Estilos visuales del formulario de ubicación.

form-usuarios

- **Descripción**: Gestiona los usuarios del sistema (crear, editar, eliminar y visualizar). Los campos incluyen nombre, email, y tipo de usuario.
- **Archivos**:
    - **form-usuarios.component.html**:
        - Contiene los campos para ingresar los datos del usuario.
    - **form-usuarios.component.ts**:
        - **ngOnInit()**: Inicializa el formulario y carga los datos del usuario si se está editando.
        - **guardarUsuario()**: Método para crear o actualizar el usuario.
        - **eliminarUsuario()**: Elimina el usuario seleccionado.
    - **form-usuarios.component.css**: Estilos para el formulario de usuarios.

`formularios.module.ts`

Este módulo agrupa todos los formularios de la aplicación relacionados con la gestión de datos como activos, edificios, órdenes de trabajo, usuarios, entre otros. Este módulo encapsula los formularios para que sean reutilizables y puedan integrarse fácilmente en otras partes de la aplicación.

Detalles del Código:

- **Declaraciones (`declarations`)**:
  Se registran los componentes de formularios individuales, como `FormActivoComponent`, `FormEdificioComponent`, `FormUsuariosComponent`, etc., para que sean reconocidos y utilizados dentro del módulo.
- **Importaciones (`imports`)**:
  - `CommonModule`: Proporciona directivas y funcionalidades esenciales de Angular como `ngIf` y `ngFor`.
  - `FormsModule`: Permite trabajar con formularios de plantilla para capturar y manejar datos del usuario.
- **Exportaciones (`exports`)**:
  Los componentes de formularios declarados en el módulo se exportan para que puedan utilizarse en otros módulos de la aplicación, como el módulo principal o módulos específicos.

`header/`

El **header/** se encarga de mostrar la cabecera en la landing page antes de iniciar sesión, proporcionando elementos clave como el logotipo, una barra de navegación y botones para acceder o registrarse. Está compuesto por tres subcomponentes: **logo**, **menu** y **navegacion**, además del archivo **header.module.ts**.

logo:
Define la sección visual del encabezado que muestra el logotipo de la institución.

- Utiliza una imagen (`LogoUTNBlanco.png`) ubicada en los recursos de la aplicación para proporcionar identidad visual.

menu:
Actúa como contenedor para la barra de navegación principal.

- Incluye el subcomponente **navegacion**, delegando en él la gestión de los enlaces y botones.

navegacion:
Es el componente principal de la barra de navegación.

- Integra el logotipo mediante el subcomponente **logo**.
- Incluye botones funcionales para **iniciar sesión** y **registrarse**, los cuales redirigen al usuario a las rutas correspondientes utilizando el servicio de Angular **Router**.
- Diseñado para ser modular y facilitar su integración o actualización.

Todos los subcomponentes están organizados de manera modular para mantener una separación clara de responsabilidades y facilitar su mantenimiento.

header.module.ts

Este módulo declara y exporta los siguientes componentes:

- **LogoComponent**
- **MenuComponent**
- **NavegacionComponent**

Además, importa **CommonModule** para utilizar funcionalidades comunes de Angular, como directivas y componentes básicos.

header-logged/

El **header-logged/** se encarga de mostrar la cabecera en la interfaz de usuario cuando el usuario ya ha iniciado sesión. Está compuesto por tres subcomponentes: **buscador**, **logo-nav** y **nav**, además del archivo **header-logged.module.ts**.

buscador:
Define la sección de búsqueda dentro del encabezado, permitiendo al usuario realizar búsquedas rápidas en la aplicación.

- Se encuentra un campo de texto en el que el usuario puede escribir su consulta.
- Al presionar "Enter" o al hacer clic en el botón de búsqueda, el sistema redirige al usuario a la ruta correspondiente basada en el término de búsqueda.
- Si el término de búsqueda no coincide con ninguna clave predefinida, se redirige al usuario a una página de error (404).
- **ngModel** se utiliza para el enlace bidireccional del valor del campo de texto.

logo-nav:
Este componente muestra el logotipo de la aplicación dentro del encabezado, específicamente cuando el usuario está logueado.

- Permite identificar visualmente la marca o el nombre de la aplicación.

nav:
Define la barra de navegación para los usuarios que ya han iniciado sesión.

- Contiene un botón para cerrar sesión, que elimina el token del almacenamiento local y redirige al usuario a la página de inicio.
- Incluye un campo de búsqueda, donde se integra el componente **buscador**.
- Está diseñado para ser responsive, con un menú emergente (popup) que aparece en pantallas más pequeñas.
- También maneja la visibilidad del menú desplegable y su cierre si se hace clic fuera de este.

header-logged.module.ts

Este módulo declara y exporta los siguientes componentes:

- **BuscadorComponent**
- **LogoNavComponent**
- **NavComponent**

El módulo importa **CommonModule** para habilitar las funcionalidades comunes de Angular y **FormsModule** para gestionar el enlace de formularios y la validación.

`landing-informacion/`

El componente `landing-informacion/` se encarga de mostrar las secciones informativas en la interfaz de la página de aterrizaje (landing page). Cada sección proporciona detalles específicos sobre el sistema de gestión de mantenimiento, sus beneficios y funcionalidades. Está compuesto por cuatro subcomponentes: `landing-info-1`, `landing-info-2`, `landing-info-3` y `landing-info-4`. Además, incluye el archivo `landing.module.ts`.

landing-info-1:

Define la estructura de la primera sección informativa, que destaca el **sistema de gestión de mantenimiento** con una breve descripción y una imagen visual representativa.

landing-info-2:

Muestra la segunda sección informativa, que se enfoca en el **mantenimiento inteligente del futuro**, resaltando la capacidad de planificar mantenimientos preventivos y crear OT's digitalmente.

landing-info-3:

Presenta la tercera sección que subraya la funcionalidad de los **códigos QR** para identificar activos, facilitando el control técnico y comercial, y el seguimiento de garantías y mantenimientos.

landing-info-4:

Define la cuarta sección informativa, que resalta la **digitalización del mantenimiento**, permitiendo un control más automatizado de tareas, procesos y recursos, eliminando el uso de hojas de cálculo.

`landing.module.ts`

Este módulo declara y exporta los siguientes componentes:

- `LandingInfo1Component`
- `LandingInfo2Component`

- `LandingInfo3Component`
- `LandingInfo4Component`

También importa `CommonModule` para habilitar las funcionalidades comunes de Angular.

`orden-trabajo/`

La carpeta **`orden-trabajo`** contiene todos los elementos relacionados con el manejo de **Órdenes de Trabajo (OT)** dentro del sistema. Esto incluye los componentes necesarios para crear, editar, visualizar y gestionar las órdenes de trabajo, así como su integración con otras entidades del sistema como activos, usuarios, y tareas.

CuerpoComponent

Este componente gestiona la parte principal de la Orden de Trabajo (OT). Es responsable de la interacción con el usuario para crear una nueva OT, incluyendo la obtención de información de los selectores y fechas.

Responsabilidades:

- Recibir los datos seleccionados por el usuario (como activo, edificio, operario, etc.) desde los componentes `SelectoresComponent` y `FechasComponent`.
- Invocar el servicio `OtService` para crear una nueva OT en la base de datos.
- Comunicar la creación de la OT al backend y manejar la respuesta.
- **Archivo**: `cuerpo.component.ts`
- **Funciones**:
    - **`ngOnInit()`**:
        - Inicializa los datos del componente, obteniendo la información necesaria desde los servicios correspondientes (por ejemplo, activos, edificios, operarios).
        - Prepara los valores predeterminados para el formulario de la Orden de Trabajo.
    - **`crearOrdenTrabajo()`**:
        - Llama al servicio `OtService` para crear una nueva OT utilizando los valores del formulario actual.
        - Recibe una respuesta del backend, indicando si la OT fue creada con éxito o si hubo un error.
        - En caso de éxito, muestra un mensaje de confirmación y reinicia el formulario. En caso de error, muestra un mensaje de error al usuario.
    - **`resetFormulario()`**:
        - Restaura los valores del formulario a su estado inicial, limpiando los campos seleccionados y reseteando las fechas y datos de la OT.

fechas/

Este componente se encarga de gestionar las fechas asociadas a la **Orden de Trabajo**. Permite al usuario asignar una fecha de inicio y finalización, así como la asignación de un operario para la OT.

Responsabilidades:

- Permitir la selección de fechas clave, como la fecha de inicio y finalización de la OT.
- Gestionar la asignación de un **operario** para la OT, consultando la lista de operarios disponibles.
- Facilitar la asignación de operarios usando el servicio `UsuariosService`.
- **Archivo**: `fechas.component.ts`
- **Funciones**:
    - **`ngOnInit()`**:
        - Inicializa las fechas predeterminadas de la Orden de Trabajo, asegurándose de que los valores de fecha sean correctos.
    - **`setFechaInicio(fechaInicio: Date)`**:
        - Establece la fecha de inicio de la OT. Este valor es luego pasado al componente principal para su uso en la creación de la OT.
    - **`setFechaFin(fechaFin: Date)`**:
        - Establece la fecha de finalización de la OT, que se utilizará en la creación o edición de la OT.
    - **`asignarOperario(operarioId: number)`**:
        - Permite seleccionar un operario para la OT, asignando su ID al campo correspondiente.
        - Utiliza el servicio `UsuariosService` para obtener y asignar los operarios disponibles.

selectores/

Este componente es el encargado de gestionar la selección de diferentes parámetros para la **Orden de Trabajo**, como el activo, edificio, sector y otros datos clave.
Responsabilidades:

- Obtener los activos, edificios, pisos, y otros datos necesarios desde los respectivos servicios como `ActivoService` y `CodigoService`.
- Proveer un conjunto de selectores para que el usuario elija el activo, el edificio y otros detalles relacionados con la OT.
- Comprimir los datos obtenidos y pasar esta información al **CuerpoComponent** para su uso en la creación de la OT.
- **Archivo**: `selectores.component.ts`
- **Funciones**:
    - **`ngOnInit()`**:

        - Obtiene los datos de los selectores (como activos, edificios, sectores) utilizando los servicios correspondientes (`ActivoService`, `CodigoService`, etc.).
        - Se suscribe a los observables de estos servicios para cargar dinámicamente las opciones disponibles en los selectores del formulario.
    - **`onChangeActivo(idActivo: number)`**:
        - Función que se ejecuta cuando el usuario selecciona un nuevo activo.
        - Actualiza el estado del componente con el nuevo activo seleccionado y pasa esta información al componente principal para que se pueda utilizar en la creación de la OT.
    - **`onChangeEdificio(idEdificio: number)`**:
        - Similar a `onChangeActivo`, actualiza el estado con el nuevo edificio seleccionado y pasa el valor al componente principal.

tareas/

- Este componente se encarga de la gestión de **tareas** relacionadas con una **Orden de Trabajo**. Aunque no se detalló completamente en tu solicitud, se infiere que su función es permitir la asignación y el manejo de tareas dentro de la OT.
- **Responsabilidades**:
    - Gestionar la creación, edición y asignación de las tareas asociadas a cada OT.
    - Probablemente se integre con otros servicios para obtener información sobre las tareas disponibles o asignadas a los activos.
    - Ofrecer la interfaz al usuario para interactuar con las tareas y vincularlas a la OT correspondiente.
- **Archivo**: `tareas.component.ts`
- **Funciones**:
    - **`ngOnInit()`**:
        - Inicializa el componente cargando las tareas asociadas a la OT o el activo, utilizando el servicio `TareasService` o un servicio similar.
        - Se suscribe a los datos del backend y actualiza la interfaz con las tareas disponibles.
    - **`agregarTarea(tareaId: number)`**:
        - Permite al usuario seleccionar una tarea específica y asignarla a la OT.
        - Se comunica con el servicio correspondiente para añadir la tarea a la OT en el backend.
    - **`eliminarTarea(tareaId: number)`**:
        - Permite eliminar una tarea previamente asignada a la OT.
        - Llama al servicio correspondiente para eliminar la tarea de la OT en el backend y actualiza el estado de la interfaz.

Carpeta `guards`

La carpeta `guards` contiene archivos encargados de proteger las rutas de la aplicación según condiciones específicas. Los guards en Angular permiten restringir el acceso a ciertas páginas en función del estado del usuario, como su autenticación o rol.

En este caso, el archivo `role.guard.ts` se encarga de verificar que el usuario tenga los permisos adecuados antes de acceder a una ruta protegida.

`role.guard.ts`

Este archivo define el guard `roleGuard`, cuya función es permitir o denegar el acceso a rutas en función del rol del usuario autenticado.

Funcionamiento:

1. **Dependencias**
    - Se importa `inject` de Angular para inyectar dependencias sin necesidad de un constructor.
    - Se usa `CanActivateFn`, una función que actúa como guardia de activación de rutas.
    - Se importa `Router` para gestionar redirecciones en caso de que el usuario no tenga acceso.
    - Se utiliza `AuthService` para obtener el rol del usuario autenticado.
2. **Lógica de restricción**
    - Se obtiene el servicio de autenticación (`AuthService`) y se extrae el rol del usuario con `getUserRole()`.
    - Se verifica si el rol del usuario está dentro de los roles permitidos para la ruta (`route.data['roles']`).
    - Si el usuario tiene un rol autorizado, se permite el acceso (`return true`).
    - Si el usuario es un operario, se le redirige a la página `'/inicioOperario'`.
    - Si no cumple ninguna de estas condiciones, el acceso es denegado (`return false`).

Este guard se usa en las rutas para asegurar que solo los usuarios con los permisos adecuados puedan acceder a determinadas secciones de la aplicación.

Carpeta `interceptors`

La carpeta `interceptors` contiene interceptores HTTP que permiten manipular las solicitudes y respuestas de la aplicación antes de que lleguen a los servicios o

componentes. Estos interceptores pueden modificar encabezados, manejar errores globalmente o realizar acciones específicas en cada petición HTTP.

En este caso, el archivo `manejoDeErrores.interceptor.ts` implementa un interceptor para capturar y gestionar errores en las respuestas del backend.

`manejoDeErrores.interceptor.ts`

Este archivo define el interceptor `manejoErroresInterceptor`, cuya función es interceptar respuestas HTTP con errores y proporcionar un manejo centralizado de estos.

Funcionamiento:

1. **Dependencias**
   - Se importa `HttpErrorResponse` para identificar respuestas HTTP con errores.
   - Se usa `HttpInterceptorFn`, un tipo de función que permite interceptar solicitudes y respuestas.
   - Se importan `catchError` y `throwError` de `rxjs` para gestionar los errores de manera reactiva.
2. **Interceptación de errores**
   - El interceptor toma una solicitud HTTP (`req`) y la envía al siguiente manejador (`next(req)`).
   - Si la solicitud genera un error, `catchError` lo captura y ejecuta la lógica de manejo.
3. **Clasificación del error**
   - Si el error proviene del cliente (`ErrorEvent`), se muestra un mensaje específico.
   - Si el error es del servidor, se clasifica según su código de estado (`status`):
     - `0`: Problema de conexión con el servidor.
     - `400`: Solicitud inválida.
     - `401`: No autorizado, requiere autenticación.
     - `403`: Acceso denegado.
     - `404`: Recurso no encontrado.
     - `500`: Error interno del servidor.
     - Otros códigos generan un mensaje de error genérico.
4. **Manejo del error**
   - Se imprimen los errores en la consola para depuración.
   - Se retorna un objeto con:
     - `errorMessage`: Mensaje detallado del error.
     - `userMessage`: Mensaje amigable para el usuario.
     - `status`: Código de error HTTP.

Este interceptor garantiza que todos los errores HTTP sean manejados de manera uniforme en la aplicación, facilitando la comunicación de fallos tanto a los desarrolladores como a los usuarios.

Carpeta `interfaces`

La carpeta `interfaces` contiene todas las definiciones de interfaces TypeScript que representan las estructuras de datos de las tablas en la base de datos. Estas interfaces son esenciales para tipar correctamente los datos en la aplicación, asegurando coherencia y facilitando el desarrollo con TypeScript.

Estructura:

Cada archivo dentro de esta carpeta define una interfaz con el mismo nombre que la tabla correspondiente en la base de datos.

Propiedades comunes:

- **Tipos de datos bien definidos:** Se utilizan `number`, `string` o valores específicos como `'admin' | 'operario'` para restringir los valores posibles de ciertos campos.
- **Correspondencia con la base de datos:** Cada interfaz refleja exactamente los nombres de los campos en la base de datos, permitiendo una integración fluida entre el backend y el frontend.
- **Facilita el uso de servicios y componentes:** Al usar estas interfaces en servicios como `UsuarioService` o `ActivoService`, se garantiza que los datos manejados cumplen con la estructura esperada.

Carpeta `pages`

La carpeta `pages` contiene los principales componentes de vista de la aplicación, cada uno representando una página accesible a través del sistema de rutas. Estos componentes son los responsables de mostrar la interfaz de usuario en diferentes secciones de la aplicación.

`inicioAdmin/`

El componente `inicioAdmin` contiene los archivos necesarios para la vista principal del administrador una vez iniciada la sesión. Esta interfaz permite la navegación a distintos formularios y secciones del sistema mediante botones interactivos. Se compone de:

inicioAdmin.component.html:

Este archivo define la estructura de la interfaz de usuario para la pantalla de inicio del administrador. Contiene:

- Un componente de navegación (`<app-nav></app-nav>`).
- Un conjunto de botones generados dinámicamente para acceder a diferentes formularios.
- Botones de navegación para desplazarse entre las opciones disponibles.
- Eventos táctiles para mejorar la experiencia en dispositivos móviles.

inicioAdmin.component.ts:

Define la lógica del componente `inicioAdminComponent`. Sus principales funciones incluyen:

- **`buttons`**: Lista de objetos con los enlaces, imágenes, textos alternativos y etiquetas de los botones disponibles.
- **`maxVisible`**: Número máximo de botones visibles a la vez.
- **`currentIndex`**: Índice actual de la lista de botones visibles.
- **Métodos de navegación (`next()` y `prev()`)**: Permiten desplazarse entre los botones disponibles.
- **Métodos de interacción táctil (`onTouchStart()` y `onTouchMove()`)**: Detectan gestos de deslizamiento para cambiar de opción en dispositivos móviles.

inicioAdmin.component.css:

Este archivo contiene los estilos asociados a la vista del administrador, asegurando una interfaz responsiva y clara para la navegación.

Funcionamiento:

1. Al ingresar a la vista de administrador, se muestran hasta cuatro botones de acceso a distintas secciones.
2. El usuario puede desplazarse entre las opciones usando los botones "Anterior" y "Siguiente".
3. En dispositivos móviles, se permite la navegación mediante gestos táctiles.
4. Al hacer clic en un botón, se redirige a la sección correspondiente.

`inicioOperario/`

El componente `inicioOperario` contiene los archivos necesarios para la vista principal del operario una vez iniciada la sesión. Esta interfaz permite la navegación a las órdenes de trabajo asignadas mediante un botón interactivo. Se compone de:

inicioOperario.component.html:

Este archivo define la estructura de la interfaz de usuario para la pantalla de inicio del operario. Contiene:

- Un componente de navegación (`<app-nav></app-nav>`).
- Un único botón generado dinámicamente que redirige a la vista de órdenes asignadas.

inicioOperario.component.ts:

Define la lógica del componente `inicioOperarioComponent`. Sus principales funciones incluyen:

- **buttons**: Lista de objetos con el enlace, imagen, texto alternativo y etiqueta del botón disponible.
- **maxVisible**: Número máximo de botones visibles a la vez (configurado en 4, aunque solo hay un botón en la lista).
- **currentIndex**: Índice actual de la lista de botones visibles.
- **Métodos de navegación (`next()` y `prev()`)**: Aunque el operario solo tiene acceso a un botón, estas funciones se mantienen para coherencia con la estructura del código.

inicioOperario.component.css:

Este archivo contiene los estilos asociados a la vista del operario, asegurando una interfaz responsiva y clara para la navegación.

Funcionamiento:

- Al ingresar a la vista del operario, se muestra un botón de acceso a la sección de órdenes asignadas.
- Al hacer clic en el botón, se redirige al formulario correspondiente.
- La navegación mediante los botones "Anterior" y "Siguiente" está presente en la estructura del código pero no se utiliza activamente debido a la existencia de una única opción de acceso.

`landing/`

El componente `landing` contiene los archivos necesarios para la página de inicio del sistema antes de iniciar sesión. Esta interfaz presenta información general sobre la plataforma y proporciona un acceso directo al formulario de registro.

landing-page.component.html:

Este archivo define la estructura de la interfaz de usuario para la pantalla principal del sistema antes de la autenticación. Contiene:

- Un componente de menú de navegación (`<app-menu></app-menu>`).
- Cuatro secciones de información (`<app-landing-info-1></app-landing-info-1>`, `<app-landing-info-2></app-landing-info-2>`,

`<app-landing-info-3></app-landing-info-3>`, `<app-landing-info-4></app-landing-info-4>`).
- Un botón de acceso (`Comenzar`) que redirige a la página de registro.

landing-page.component.ts:

Define la lógica del componente `LandingPageComponent`. Sus principales funciones incluyen:

- **redirigirARegistro()**: Método que redirige al usuario a la página de registro al hacer clic en el botón "Comenzar".

landing-page.component.css:

Este archivo contiene los estilos asociados a la página de inicio, asegurando una presentación clara y accesible.

Funcionamiento:

- Al ingresar a la página principal sin haber iniciado sesión, se presentan varias secciones de información sobre el sistema.
- Se muestra un botón "Comenzar" que permite a los usuarios acceder directamente al formulario de registro.
- La navegación dentro de la página es facilitada por el componente de menú y la disposición estructurada de la información.

`login/`

La carpeta `login` contiene los archivos necesarios para la vista de inicio de sesión. Este componente permite a los usuarios autenticarse mediante credenciales de correo electrónico y contraseña, con la opción de recordar la sesión.

login.component.html:

Este archivo define la estructura de la interfaz de usuario para la pantalla de inicio de sesión. Contiene:

- Un contenedor principal con el formulario de autenticación.
- Un campo de entrada para el correo electrónico.
- Un campo de entrada para la contraseña.
- Una casilla de verificación opcional para recordar la sesión.
- Un botón de ingreso que ejecuta la autenticación al enviarse el formulario.
- Un enlace que redirige a la pantalla de registro si el usuario no tiene cuenta.

login.component.ts:

Define la lógica del componente `LoginComponent`. Sus principales funciones incluyen:

- **`email` y `password`**: Variables que almacenan las credenciales ingresadas por el usuario.
- **`login()`**:
  - Verifica que los campos de email y contraseña no estén vacíos.
  - Llama al servicio de autenticación (`AuthService`) para validar las credenciales.
  - Si el inicio de sesión es exitoso, redirige a la vista de administración (`/inicioAdmin`).
  - Si ocurre un error, muestra una alerta informando el fallo en la autenticación.
- **`goToRegister()`**: Redirige a la página de registro (`/registro`).

login.component.css:

Este archivo contiene los estilos asociados a la vista de inicio de sesión, asegurando una interfaz clara y estructurada.

Funcionamiento:

- Al ingresar a la vista de inicio de sesión, el usuario puede completar sus credenciales y presionar el botón "Ingresar".
- Si los datos son correctos, se redirige a la vista de administración.
- Si los datos son incorrectos o hay un error en la autenticación, se muestra un mensaje de alerta.
- En caso de no tener cuenta, el usuario puede hacer clic en el enlace de registro para crear una nueva.

`registro/`

El componente `registro` contiene los archivos necesarios para la vista de registro de usuario. Este componente permite a los usuarios crear una cuenta proporcionando sus datos personales y seleccionando un rol dentro del sistema.

registro.component.html:

Este archivo define la estructura de la interfaz de usuario para la pantalla de registro. Contiene:

- Un contenedor principal con el formulario de registro.
- Un área para subir una imagen de perfil opcional.
- Campos de entrada para nombre, correo electrónico, contraseña y confirmación de contraseña.
- Un selector de rol con opciones predefinidas (`admin` y `operario`).
- Un botón de registro que envía el formulario.

- Un enlace que redirige a la pantalla de inicio de sesión si el usuario ya tiene una cuenta.

registro.component.ts:

Define la lógica del componente `RegistroComponent`. Sus principales funciones incluyen:

- **`user`**: Objeto de tipo `IRegister` que almacena los datos del usuario a registrar.
- **`roles`**: Arreglo que contiene los roles disponibles (`admin` y `operario`).
- **`imageUrl`**: Variable para almacenar la imagen de perfil seleccionada por el usuario.
- **`register()`**:
  - Verifica que todos los campos obligatorios estén completos.
  - Verifica que la contraseña y la confirmación coincidan.
  - Envía los datos al servicio de autenticación (`AuthService`) para registrar al usuario.
  - Si el registro es exitoso, muestra una alerta y redirige a la pantalla de inicio de sesión.
  - Si ocurre un error, muestra una alerta con el mensaje correspondiente.
- **`goToLogin()`**: Redirige a la página de inicio de sesión (`/login`).
- **`onImageSelected(event)`**:
  - Captura la imagen seleccionada por el usuario y la convierte a formato `base64`.
  - Asigna la imagen al objeto `user.imagenperfil` para su posterior envío.

registro.component.css:

Este archivo contiene los estilos asociados a la vista de registro, asegurando una interfaz clara y estructurada.

Funcionamiento:

- El usuario ingresa su información en los campos requeridos y selecciona un rol.
- Si lo desea, puede subir una imagen de perfil haciendo clic en el icono.
- Al hacer clic en "Registrarse", se validan los datos y se envían al backend.
- Si el registro es exitoso, el usuario es redirigido a la pantalla de inicio de sesión.
- Si el usuario ya tiene cuenta, puede hacer clic en el enlace para acceder a la pantalla de inicio de sesión.

`not-found/`

El componente `NotFoundComponent` es una pantalla que se muestra cuando el usuario busca algo que no tiene resultados o intenta acceder a una ruta inexistente dentro de la aplicación.

not-found.component.html:

Define la estructura de la interfaz cuando no se encuentra un resultado:

- Incluye el componente de navegación `<app-nav>`.
- Muestra un mensaje indicando que la búsqueda no tuvo resultados.
- Contiene una breve descripción sugiriendo perfeccionar la búsqueda o recargar la página.

not-found.component.ts:

Este componente no maneja lógica específica, ya que solo se encarga de mostrar el mensaje de error.

not-found.component.css:

Archivo de estilos encargado de:

- Centrar el contenido en la pantalla.
- Aplicar diseño y espaciado adecuados para mejorar la experiencia del usuario.

Funcionamiento:

- Cuando el usuario accede a una ruta no válida o no hay resultados en una búsqueda, se renderiza esta vista.
- Se le informa al usuario que no se encontraron resultados y se le sugiere mejorar la búsqueda o recargar la página.

Carpeta environments

La carpeta `environments` contiene los archivos de configuración de los entornos de desarrollo y producción. Estos archivos almacenan variables de entorno clave como la URL base de la API, que varía según el entorno en el que se ejecute la aplicación.

environment.development.ts:

Este archivo contiene las configuraciones específicas para el entorno de desarrollo. Establece la URL base de la API a un servidor local de desarrollo.

- **urlBase**: Define la URL base de la API utilizada durante el desarrollo.

environment.ts:

Este archivo contiene las configuraciones específicas para el entorno de producción. En este archivo, además de la URL base de la API, se establece la propiedad `production` a `true` para indicar que la aplicación está ejecutándose en modo de producción.

- **urlBase**: Define la URL base de la API utilizada en producción.
- **production**: Define el modo de ejecución de la aplicación, que está configurado en `true` para producción.

Carpeta services

La carpeta `services` contiene los servicios que gestionan la comunicación entre la aplicación Angular y el backend. Estos servicios utilizan `HttpClient` para realizar solicitudes HTTP y manejar datos relacionados con distintas entidades del sistema. Además, implementan autenticación mediante `AuthService`, utilizando tokens en las cabeceras de las peticiones.

Cada servicio sigue la misma estructura:

- **Obtener registros**: Retorna una lista de registros de la entidad correspondiente.
- **Obtener registro por ID**: Retorna un solo registro basado en su identificador.
- **Crear registro**: Envía datos al backend para crear un nuevo registro.
- **Editar registro**: Modifica un registro existente basado en su ID.
- **Eliminar registro**: Elimina un registro basado en su ID.

Servicios Generales

Los siguientes servicios manejan operaciones CRUD para distintas entidades de la base de datos. Todos siguen la misma estructura y exponen los siguientes métodos:

- `obtenerTodos()`: Retorna todos los registros de la entidad.
- `obtenerPorId(id: number)`: Retorna un registro específico por su ID.
- `crear(datos: any)`: Crea un nuevo registro en la base de datos.
- `editar(id: number, datos: any)`: Modifica un registro existente.
- `eliminar(id: number)`: Elimina un registro por su ID.

Lista de servicios CRUD

- **`EdificioService`**
- **`PisoService`**
- **`SectorService`**
- **`UbicacionService`**
- **`ActivoService`**
- **`TareaService`**
- **`ActivoTareaService`**
- **`CantidadService`**
- **`OperarioService`**
- **`AdminService`**
- **`UsuariosService`**
- **`TagService`**
- **`OrdenTrabajoService`**

Estos servicios manejan la interacción con la API y encapsulan la lógica de negocio necesaria para cada entidad.

Servicios Especiales

Algunos servicios tienen funcionalidades específicas y requieren una descripción más detallada.

AuthService

Este servicio maneja la autenticación de usuarios y la gestión de sesiones.

Métodos

- `login(credenciales: { usuario: string, contrasena: string })`: Envía las credenciales al backend y obtiene un token de autenticación.
- `logout()`: Elimina el token de sesión y finaliza la autenticación del usuario.
- `obtenerUsuarioActual()`: Retorna la información del usuario autenticado.
- `verificarSesion()`: Comprueba si el usuario sigue autenticado.

FiltrosService

Encargado de aplicar y gestionar filtros en la visualización de registros en tablas.

Métodos

- `aplicarFiltro(tipo, valor)`: Aplica un filtro específico a los datos mostrados.
- `removerFiltro(tipo)`: Elimina un filtro aplicado previamente.
- `obtenerFiltrosActivos()`: Devuelve la lista de filtros actualmente en uso.

SelectionService

Este servicio proporciona listas de selección para formularios en la aplicación.

Métodos

- `obtenerEdificios()`: Retorna una lista de edificios disponibles.
- `obtenerSectores(edificioId: number)`: Retorna los sectores de un edificio específico.
- `obtenerActivos(sectorId: number)`: Retorna los activos de un sector específico.
- `obtenerOperarios()`: Retorna una lista de operarios registrados.

Este servicio se usa para poblar los selectores dinámicos en los formularios.

Conclusión

Esta documentación busca servir como una guía integral para comprender, implementar y mantener el proyecto en cuestión. Hemos cubierto las principales áreas de interés, incluyendo la estructura del proyecto, los servicios esenciales, las configuraciones necesarias, las dependencias empleadas y las herramientas utilizadas, como Compodoc para generar y visualizar documentación adicional.

El objetivo principal es facilitar la colaboración entre desarrolladores, garantizar la escalabilidad y mantener la calidad del código a lo largo del ciclo de vida del proyecto. En caso de duda o aspectos que puedan mejorarse, se está invitado a colaborar y contribuir a esta documentación para que siga siendo una herramienta útil y actualizada para todos los miembros del equipo.