MANUAL TÉCNICO

Los NullPointers

Guatemala 22 de junio 2021

Universidad San Carlos de Guatemala

INTRODUCCIÓN

El motivo fundamental de este documento es brindar el conocimiento de los requerimientos, arquitectura y el análisis de las funcionalidades del sistema, brindando un análisis de las herramientas utilizadas. Este sistema fue desarrollado mediante las técnicas de un flujo de trabajo en ramas.

REQUERIMIENTOS Web

- Sistema operativo: Windows 8 o posterior, Linux, macOS 10.7 o posterior
- CPU (s): x86-64
- Memoria: Mínimo de 4 GB
- Almacenamiento: 20 GB de espacio libre
- Ancho de banda de internet: 10 Mbps
- Instalaciones previas: NodeJS, Angular CLI
- Navegadores recomendados: Google, FireFox, Edge.

REQUERIMIENTOS App

- Sistema operativo: Android 9 o superior
- CPU (s): ARM
- Memoria: Mínimo de 3 GB
- Almacenamiento: 10 GB de espacio libre
- Ancho de banda de internet: 10 Mbps

EJECUCIÓN

Debido a que la arquitectura del software es de tipo Cliente-Servidor, su desarrollo necesito de scripts para la ejecución local. Para ejecutar la página de los empleados municipales se realiza lo siguiente:

- Con el CLI de su preferencia, y ubicado en la carpeta del código fuente de la página, ejecutar los siguientes comandos:
  - npm install
  - ng serve --port 4200
- Al finalizar la ejecución de los comandos previos, en el navegador de su preferencia diríjase a "localhost:4200" y verá la aplicación de los empleados municipales corriendo.

Para compilar y construir el apk de nuestra aplicación que sera utilizada por ciudadanos hecho con angular, utilizamos una herramienta que se llama capacitor, el cual convierte nuestro proyecto angular a un proyecto en android studio siguiendo estos pasos:

- Instalación de capacitor
  - npm install @capacitor/core @capacitor/cli
- Inicializar capacitor
  - npx cap init
- Compilar Angular
  - ng build --build-optimizer --output-hashing=none
- Preparación del entorno para android
  - npm install @capacitor/android
  - npx cap add android
- Compilar y actualizar cambios dist en android
  - npx cap sync
  - npx cap copy
  - npx cap update
  - npx cap open

- Abierto automáticamente nuestro android studio, en la opción Build> Buil Bundles (s) > Build APK (s), con esto tendremos listo nuestro apk.

Para ejecutar el Servidor se realiza lo siguiente:

- Con el CLI de su preferencia, y ubicado en la carpeta del código fuente del servidor, ejecutar los siguientes comandos:
  - npm install
  - npm run dev
- Al finalizar la ejecución de los comandos previos, el servidor estará iniciado y escuchará peticiones en el puerto 3000 de manera local.

DESPLIEGUE

Para realizar el despliegue se cuenta con la ayuda de la herramienta docker y docker-compose. Para la página del empleado municipal y para el servidor backend se realizó un archivo Dockerfile para cada proyecto el cual contiene el proceso de compilación y ejecución del proyecto para producción.

```
FROM node:14-alpine

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD [ "npm", "start" ]
```

Cada imagen creada con el script de ejecución en el Dockerfile fue enlazada por medio de un archivo llamado docker-compose.yml el cual es el encargado de realizar la compilación de la misma, exponer los puertos y reiniciar el proceso del mismo en dado caso falle.

ANALISIS Y HERRAMIENTAS

La plataforma seleccionada para el despliegue del cliente de la página web y el servidor-backend fueron contenedores Docker alojados en la nube; se optó por Docker ya que este permite la ejecución de una aplicación en cualquier entorno, y también porque la infraestructura serían máquinas virtuales del proveedor de servicios de nube, Google.

En el caso del cliente web, en el contenedor se utiliza Node.js para que permita la construcción de la aplicación desarrollada en el framework de Angular compilada a JavaScript. También se incluyó NGNIX en dicho contenedor, permitiendo escuchar peticiones HTTP y el despliegue de la aplicación en la nube.

Para el servidor-backend, el contenedor utiliza también Node.js, para realizar la ejecución y escucha de peticiones HTTP por medio de una API-REST programada en JavaScript, la cual permite recuperar la información de la base de datos.

La base de datos, con la que se manipula y se guarda la información utilizada en la aplicación, fue definida en MySQL teniendo un modelo relacional. Esta fue alojada en la nube por medio del servicio de Google especializado en el alojamiento de bases de datos.

El cliente de la aplicación móvil consiste en una APK de Android, la cual tiene el contenido de la aplicación móvil desarrollada en Angular ya compilada utilizando Capacitor. Esta debe de ser instalada en el dispositivo móvil con sistema operativo Android para su funcionamiento.

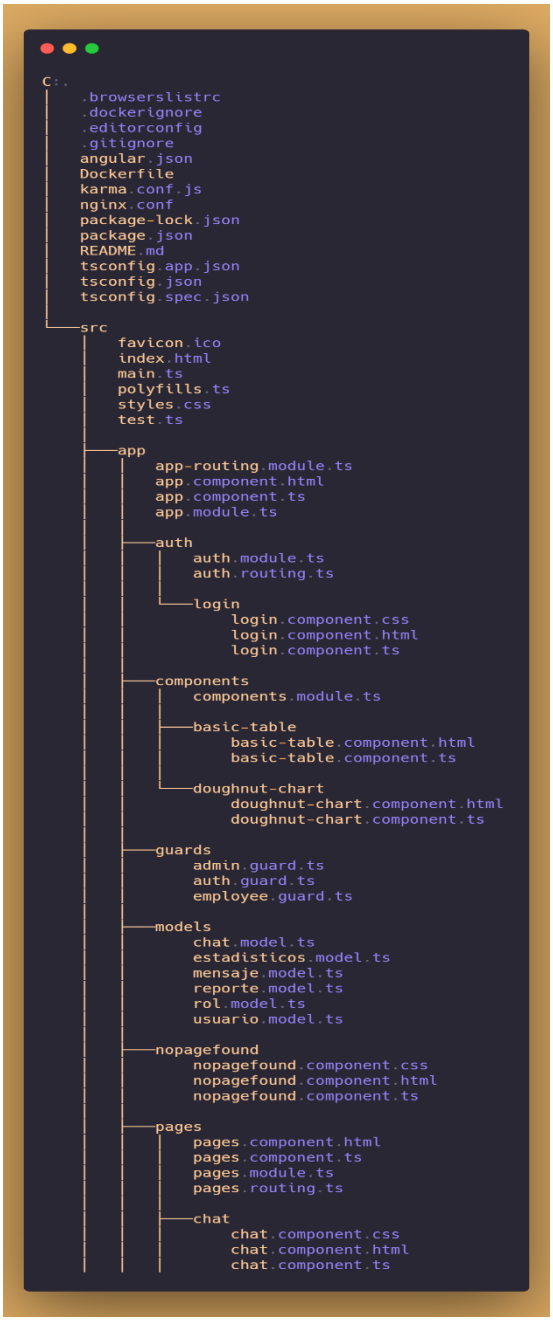
ESTRUCTURA

BACK-END

El back-end posee la siguiente estructura con el fin de volver dinámico el crecimiento de la aplicación y llevar un mejor control de las funcionalidades de esta.

PÁGINA EMPLEADOS MUNICIPALES

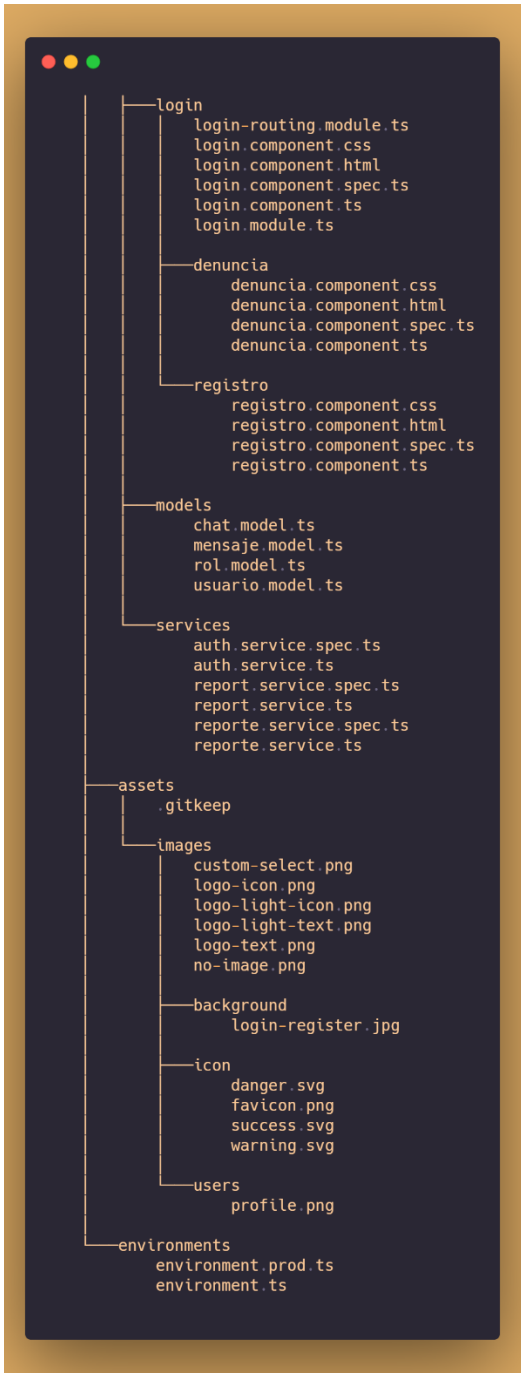
La estructura del front-end está basada en una aplicación de angular donde se utilizaron módulos para los componentes, que se despliegan dentro de las páginas, y las páginas en sí mismas; esto se hizo con el fin de facilitar el crecimiento de la aplicación y llevar un mejor control de los elementos que la conforman.



APLICACIÓN MÓVIL

La estructura de la app movil está basada en una aplicación de angular donde se utilizaron módulos para los componentes, que se despliegan dentro de las páginas, y las páginas en sí mismas; para poder hacer funcionar esta aplicación como un apk fue necesario implementar una herramienta llamada cordova, la cual nos ayuda a construir un proyecto de angular a un apk.


C:.
.browserslistrc
.editorconfig
.gitignore
angular.json
capacitor.config.ts
karma.conf.js
package-lock.json
package.json
README.md
tsconfig.app.json
tsconfig.json
tsconfig.spec.json
src
favicon.ico
index.html
main.ts
polyfills.ts
styles.css
test.ts
app
app-routing.module.ts
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
dashboard
dashboard-routing.module.ts
dashboard.component.css
dashboard.component.html
dashboard.component.spec.ts
dashboard.component.ts
dashboard.module.ts
greporte
greporte.component.css
greporte.component.html
greporte.component.spec.ts
greporte.component.ts
view-chats
view-chats-routing.module.ts
view-chats.component.css
view-chats.component.html
view-chats.component.spec.ts
view-chats.component.ts
view-chats.module.ts
services
chat.service.spec.ts
chat.service.ts
view-photos
view-photos.component.css
view-photos.component.html
view-photos.component.spec.ts
view-photos.component.ts
view-reports
view-reports.component.css
view-reports.component.html
view-reports.component.spec.ts
view-reports.component.ts
report-item
report-item.component.css
report-item.component.html
report-item.component.spec.ts
report-item.component.ts
welcome
welcome.component.css
welcome.component.html
welcome.component.spec.ts
welcome.component.ts


login
login-routing.module.ts
login.component.css
login.component.html
login.component.spec.ts
login.component.ts
login.module.ts
denuncia
denuncia.component.css
denuncia.component.html
denuncia.component.spec.ts
denuncia.component.ts
registro
registro.component.css
registro.component.html
registro.component.spec.ts
registro.component.ts
models
chat.model.ts
mensaje.model.ts
rol.model.ts
usuario.model.ts
services
auth.service.spec.ts
auth.service.ts
report.service.spec.ts
report.service.ts
reporte.service.spec.ts
reporte.service.ts
assets
.gitkeep
images
custom-select.png
logo-icon.png
logo-light-icon.png
logo-light-text.png
logo-text.png
no-image.png
background
login-register.jpg
icon
danger.svg
favicon.png
success.svg
warning.svg
users
profile.png
environments
environment.prod.ts
environment.ts

FUNCIONALIDADES PRINCIPALES

BACK-END

Las funcionalidades del back-end se encuentran distribuidas en diferentes directorios, en los cuales están los modelos, controladores, rutas y configuraciones.

En el directorio de configuraciones podemos encontrar la conexión a la base de datos y la funcionalidad de almacenamiento de imágenes.

- ALMACENAMIENTO DE IMÁGENES: Esta función lo recibe un archivo el cual se almacenará en la carpeta uploads y donde su nombre se generará por medio del módulo uuid con su respectiva extensión

```
const storage = multer.diskStorage({
    destination: 'uploads',
    filename: (_req, file, callback) => {
        callback(null, uuid.v4() + path.extname(file.originalname));
    }
});
```

- CONEXIÓN BASE DE DATOS: Para la conexión de la base de datos se necesita pasar un objeto con el usuario, contraseña, nombre de la base de datos y el host donde se encuentra la misma.

En el directorio de modelos podemos encontrar cada uno de los archivos con las consultas e importaciones necesarias para realizar las mismas.

Cada función del modelo recibe dos parámetros los cuales son una función callback y los parámetros que vienen de la petición, por medio de la destructuración se obtienen todos los parámetros necesarios y se concatenan a la consulta la cual es ejecuta pasando la misma y el callback

```
addFile(params, callback) {
    const {
        nombre,
        ruta,
        reporte,
    } = params

    const query = `INSERT INTO Archivo (nombre,ruta, reporte)
        VALUES('${nombre}','${ruta}',${reporte});`;

    return this.executeQuery(query, callback);
}
```

En el directorio de controladores encontramos la ejecución de las consultas realizadas en los modelos obteniendo de estas el resultado. También se realiza una verificación que si la ejecución de la consulta fue satisfactoria o no. Dependiendo el resultado se retorna un código 200 y los registros obtenidos si fue satisfactoria, si la consulta no fue satisfactoria se retorna el código 500 y el error que se ocasiono.

```
addFile: (req, res) => {
    fileModel.addFile(req.body, (err, results) => {
        if (err) {
            res.status(500).send({
                code: 500,
                data: err
            });
            return;
        }
        res.status(200).send({
            code: 200,
            data: results
        });
    });
}
```

En el folder de rutas podemos cada uno de los endpoints con su método rest permitido.


router.route('/')
.post(fileController.addFile)
router.route('/:id')
.get(fileController.get)
router.route('/photo')
.post(multer.single('image'), fileController.create);

PÁGINA EMPLEADOS MUNICIPALES

La pagina de los empleados municipales realiza solicitudes http al servidor back-end con el fin de interactuar con la información que se encuentra alojada en la base de datos. Estas peticiones se encuentran en los archivos de servicios. Todos los archivos de servicios poseen una base similar con funciones asíncronas retornando una promesa, agregando los headers y contenido necesario.

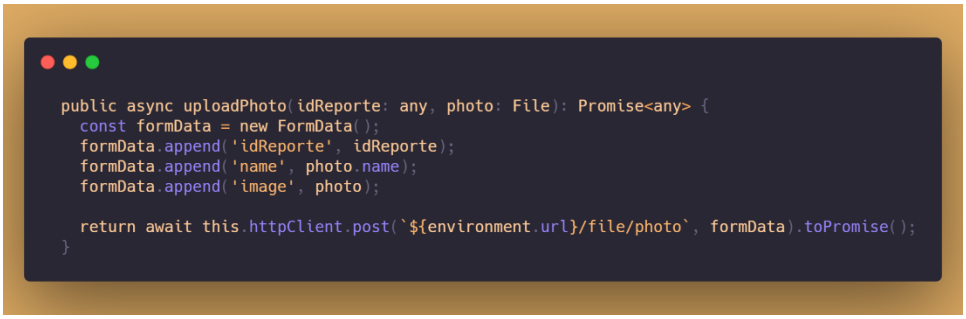
```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';

import { environment } from 'src/environments/environment';

import { Usuario } from 'src/app/models/usuario.model';

@Injectable({
  providedIn: 'root'
})
export class UsuarioService {
  public url: string;

  constructor(private _httpClient: HttpClient) {
    this.url = `${environment.url}/user`;
  }

  public async getRol(usuario: Usuario): Promise<any> {
    return await this._httpClient.get(`${this.url}/rol/${usuario.usuarioID}`)
      .toPromise();
  }

  public async registrarEmpleado(usuario: Usuario): Promise<any> {
    const json = JSON.stringify(usuario);
    const headers = new HttpHeaders().set('Content-Type', 'application/json');

    return await this._httpClient.post(`${this.url}/registro/empleado`, json, { headers })
      .toPromise();
  }
}
```

APLICACIÓN MÓVIL

La aplicación móvil realiza llamadas a solicitudes http, esperando que el servidor alojado en la nube pueda atenderlas y devolver los resultados correspondientes. Estas promesas que se realizan para consumir los datos manejan una estructura similar utilizando comúnmente get y post.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { environment } from 'src/environments/environment';
import { Usuario } from '../models/usuario.model';
import { BehaviorSubject, Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthService {


  public url: string;

  private currentUserSubject: BehaviorSubject<Usuario>;

  constructor(private httpClient: HttpClient) {
    this.url = `${environment.url}/user`
    this.currentUserSubject = new BehaviorSubject<Usuario>(JSON.parse(localStorage.getItem('user') || '{}'));
  }

  public getUsuarioEnSesion(): Usuario {
    return this.currentUserSubject.value
  }

  public setUsuarioEnSesion(user: Usuario) {
    localStorage.setItem('user', JSON.stringify(user))
    this.currentUserSubject.next(user);
  }

  public endSession() {
    localStorage.removeItem('user');
    this.currentUserSubject.next(new Usuario());
  }

  public isSessionActive() {
    return this.getUsuarioEnSesion().usuarioID !== undefined && this.getUsuarioEnSesion().usuarioID !== 0
  }


  public async authenticate(usuario: Usuario): Promise<any> {
    const json = JSON.stringify(usuario);
    const headers = new HttpHeaders().set('Content-Type', 'application/json');

    return await this.httpClient.post(`${this.url}/login`, json, { headers })
      .toPromise();
  }

  public async signUp(usuario: Usuario): Promise<any> {
    const json = JSON.stringify(usuario);
    const headers = new HttpHeaders().set('Content-Type', 'application/json');

    return await this.httpClient.post(`${this.url}/registro`, json, { headers })
      .toPromise();
  }
}
```

Un método el cual se tiene que tener mucho ojo, es la petición la cual se utiliza para almacenar las imágenes, esta tiene una peculiaridad que las imágenes serán enviadas en un tipo de formData.



```
public async uploadPhoto(idReporte: any, photo: File): Promise<any> {
  const formData = new FormData();
  formData.append('idReporte', idReporte);
  formData.append('name', photo.name);
  formData.append('image', photo);

  return await this.httpClient.post(`${environment.url}/file/photo`, formData).toPromise();
}
```

Para finalizar, una configuración importante el cual tenemos que tener siempre presente, es que al momento de tener nuestro proyecto listo para su compilación en Android studio, en el archivo manifest hacer la siguiente configuración para poder permitir el tráfico de peticiones get y post.

```
<application
android:usesCleartextTraffic="true">
</application>
```