Departamento de Ciencias de la Computación

Ingeniería de Software

Aplicaciones Distribuidas

Tarea Grupal
Microservicios y Angular 17

NRC: 14930

Autores:

Díaz Bautista Adriana Lizbeth
Masacela Atupaña Brandon Raúl
Pila Torres Johanna Jicela
Pilataxi Chisaguano Daniela del Pilar

"La vida debe ser comprendida hacia atrás, pero debe ser vivida hacia delante."

Søren Kierkegaard

Sangolquí, domingo 20 de julio de 2024

ÍNDICE

ÍNDICE DE FIGURAS

1. Objetivos

1.1. Objetivo General

Desarrollar un sistema de matrículas y gestión de estudiantes utilizando microservicios para el backend y Angular para el frontend, asegurando funcionalidad completa y eficiente a través de un proceso documentado y probado.

1.2. Objetivos Específicos

- Diseñar e implementar microservicios para la gestión de estudiantes y cursos, permitiendo operaciones CRUD y matriculación.
- Desarrollar una interfaz de usuario intuitiva utilizando Angular que permita a los usuarios gestionar estudiantes, cursos y matrículas de manera efectiva.
- Planificar y ejecutar pruebas exhaustivas, incluyendo pruebas unitarias e integración para los microservicios backend con JUnit, pruebas unitarias para los componentes y servicios Angular con Jasmine y Karma, y pruebas end-to-end (E2E) para la aplicación Angular.

2. Introducción

Al combinar microservicios y Angular nos permite ofrecer una arquitectura flexible y escalable. Los microservicios permiten dividir la aplicación en pequeños servicios independientes que pueden desarrollarse, desplegarse y gestionarse de forma autónoma. Esto facilita el mantenimiento y la evolución del sistema, permitiendo añadir nuevas funcionalidades sin afectar a los servicios existentes. Por otro lado, Angular proporciona una plataforma robusta para el desarrollo del frontend, ofreciendo una experiencia de usuario rica y dinámica.

2.1. Microservicios

Son módulos ligeros que pueden servir como componentes básicos de las aplicaciones complejas basadas en la nube. Aunque los microservicios individuales pueden operar de forma independiente, tienen un acople suelto en una interfaz unificada.

La arquitectura de microservicios se considera un reemplazo moderno y flexible del modelo de desarrollo más tradicional de la arquitectura monolítica. Hacen que las aplicaciones sean más fáciles de escalar y más rápidas de desarrollar. Esto permite la innovación y acelera el tiempo de comercialización de las nuevas características. (*¿Qué Son los Microservicios? | AWS*, s. f.)

2.2. Angular

> Es un framework de JavaScript de código abierto escrito en TypeScript. Su objetivo principal es desarrollar aplicaciones de una sola página. Google se encarga del mantenimiento y constantes actualizaciones de mejoras para este framework. Se basa en una colección de bibliotecas bien integradas que cubren una amplia variedad de características, que incluyen enrutamiento, administración de formularios, comunicación cliente-servidor y más. (Gonçalves, 2024)

El desarrollo de este Sistema de Matrículas y Gestión de Estudiantes con microservicios y Angular nos permite desarrollar un sistema modular y escalable, proporcionando una solución eficiente y flexible aprovechando las ventajas de los microservicios y la versatilidad de Angular en el desarrollo de aplicaciones web.

3. Desarrollo

3.1. Msvc Curso

El microservicio ***Curso*** es responsable de gestionar la información de los cursos dentro del sistema. Soporta la creación, actualización, eliminación y visualización de los cursos.

3.1.1. Estructura del microservicio

msvc-cursos

- .mvn — Archivos de configuración de Maven
- src
  - main
    - java
      - com.espe.msvc_cursos
        - *clients* — para comunicación con otros microservicios
        - *controllers* — Controladores que manejan las solicitudes HTTP
        - *models* — Modelos de datos
          - *entity* — Entidades JPA
        - *repositories* — para acceso a la base de datos
        - *services* — contiene la lógica de negocio
    - resources — archivos de configuración y recursos

    - test ⟶ Pruebas unitarias y de integración
- pom.xml ⟶ Archivo de configuración de Maven

La estructura del microservicio ***Curso*** está diseñada para seguir principios de organización y separación de responsabilidades, lo que facilita el desarrollo, mantenimiento y escalabilidad del sistema.

3.1.2. Componentes del Microservicio *"Curso"*

1. Models

El modelo *Curso* representa un curso dentro del sistema. Se encuentra en el paquete *com.espe.msvc_cursos.models.entity*.

```
package com.espe.msvc_cursos.models.entity;

import com.espe.msvc_cursos.models.Usuario;
import jakarta.persistence.*;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotEmpty;

import java.util.ArrayList;
import java.util.List;

@Entity
@Table (name = "cursos")
public class Curso {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @NotBlank
    @NotEmpty
    private String nombre;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch =
FetchType.LAZY)
    @JoinColumn(name = "curso_id")
    private List<CursoUsuario> cursoUsuarios;

    @Transient
    private List<Usuario> usuarios;

    public Curso() {
        cursoUsuarios = new ArrayList<>();
        usuarios = new ArrayList<>();
    }

// Getters y setters
```

El modelo *CursoUsuario,* Representa la relación entre cursos y usuarios, permitiendo asociar un usuario con un curso específico. Cada instancia de *CursoUsuario* corresponde a una inscripción de un usuario en un curso.

```
package com.espe.msvc_cursos.models.entity;

import jakarta.persistence.*;
```

```
@Entity
@Table(name = "curso_usuario")
public class CursoUsuario {
@Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "usuario_id", unique = true)
    private Long usuarioId;

    //Métodos get y set
}
```

El modelo *Usuario* sirve como un modelo de datos que representa a un usuario en el sistema de cursos.

```
package com.espe.msvc_cursos.models;

public class Usuario {
    private Long id;

    private String nombre;

    private String email;

    private String password;

   //Métodos get y set
}
```

2. **Controllers**

El ***CursoController*** maneja las solicitudes HTTP relacionadas con la gestión de cursos. Proporciona endpoints para crear, actualizar, eliminar y recuperar información de los cursos.

```
package com.espe.msvc_cursos.controllers;

import com.espe.msvc_cursos.models.Usuario;
import com.espe.msvc_cursos.models.entity.Curso;
import com.espe.msvc_cursos.servicies.CursoService;
import feign.FeignException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.Collections;
import java.util.List;
import java.util.Optional;

@RestController
@CrossOrigin(origins = "http://localhost:4200")
public class CursoController {
    @Autowired
    private CursoService service;

    // Endpoints CRUD para Curso
}
```

3. **Services**

   La interfaz *CursoService* define la lógica de negocio para la gestión de cursos

```
package com.espe.msvc_cursos.servicies;

import com.espe.msvc_cursos.models.Usuario;
import com.espe.msvc_cursos.models.entity.Curso;
import com.fasterxml.jackson.annotation.OptBoolean;

import java.util.List;
import java.util.Optional;

public interface CursoService {

  // Métodos de lógica de negocio para Curso

}
```

   La clase *CursoServiceImpl* proporciona la implementación de estos métodos.

```
package com.espe.msvc_cursos.servicies;

import com.espe.msvc_cursos.clients.UsuarioClientRest;
import com.espe.msvc_cursos.models.Usuario;
import com.espe.msvc_cursos.models.entity.Curso;
import com.espe.msvc_cursos.models.entity.CursoUsuario;
import com.espe.msvc_cursos.repositories.CursoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class CursoServiceImpl implements CursoService{
    @Autowired
    private CursoRepository repository;

    // Implementación de los métodos de lógica de negocio

}
```

4. **Repositories**

   La interfaz *CursoRepository* extiende el repositorio de Spring Data JPA, proporcionando operaciones CRUD para la entidad ***Curso***.

```
package com.espe.msvc_cursos.repositories;

import com.espe.msvc_cursos.models.entity.Curso;
import org.springframework.data.repository.CrudRepository;

public interface CursoRepository extends CrudRepository<Curso,Long> {
}
```

5. **Clients**

La clase *UsuarioClientRest* se utiliza para comunicarse con otros microservicios. Puede llamar a los endpoints de otros microservicios para recuperar o enviar datos.

```
package com.espe.msvc_cursos.clients;

import com.espe.msvc_cursos.models.Usuario;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.*;

@CrossOrigin
@FeignClient(name = "msvc-usuarios", url = "localhost:8001")
public interface UsuarioClientRest {

    //Metodos de comunicación

    @GetMapping("/usuario/BuscarUsuarioPorID/{id}")
    Usuario detalle(@PathVariable long id);

    @PostMapping("/usuario/CrearUsuario")
    Usuario crear(@RequestBody Usuario usuario);
}
```

3.2. Msvc Usuario

El microservicio ***Usuario*** es responsable de gestionar la información de los estudiantes dentro del sistema. Soporta la creación, actualización, eliminación y visualización de registros de estudiantes. Este microservicio está desarrollado en Java y Spring Boot, y se comunica con otros microservicios del sistema para facilitar la matriculación de estudiantes en cursos.

3.2.1. Estructura del servicio

En referencia al microservicio de Cursos, se construye el microservicio del Usuario con una estructura similar a la del microservicio `Curso`, organizada en las siguientes capas:

- **Modelos** (models): Define las entidades JPA que representan los datos de los usuarios.
- **Controladores** (controllers): Manejan las solicitudes HTTP y exponen los endpoints de la API para las operaciones CRUD de usuarios.
- **Servicios** (services): Contienen la lógica de negocio específica para la gestión de usuarios.
- **Repositorios** (repositories): Gestionan el acceso a la base de datos utilizando Spring Data JPA

Diferencias Clave

1. Funcionalidad
    - Se centra en la gestión de usuarios, incluyendo operaciones como creación, actualización, eliminación y visualización de usuarios.
2. Entidades y Modelos
    - La entidad principal es Usuario, que representa a un usuario del sistema.
3. Interacción entre microservicios
    - Funciona de manera más autónoma en la gestión de usuarios.

Componentes del microservicio *Usuario*

Models

- **Usuario**: Representa un usuario del sistema, con propiedades como id, nombre y email.

Controllers

- **UsuarioController**: Define los endpoints para las operaciones CRUD de usuarios.

Services

- **UsuarioService**: Interfaz que define los métodos de negocio para la gestión de usuarios.
- **UsuarioServiceImpl**: Implementación de UsuarioService, donde se desarrolla la lógica de negocio.

Repositories

- **UsuarioRepository**: Extiende *JpaRepository* para proporcionar operaciones CRUD sobre la entidad *Usuario*.

Decisiones de diseño

- Arquitectura de microservicios: Permite el despliegue y escalamiento independiente del servicio de usuarios.
- Uso de Spring Boot: Facilita el desarrollo rápido y el despliegue de aplicaciones
- Spring Data JPA: Proporciona una forma eficiente de interactuar con la base de datos, reduciendo el código necesario para las operaciones CRUD.

Desafíos y soluciones

- Comunicación entre servicios: Gestionar la comunicación entre microservicios puede ser complejo. Se eligió Feign Client para simplificar este proceso, proporcionando un enfoque declarativo para definir clientes HTTP.

- Consistencia de la Base de datos: Asegurar la consistencia de los datos entre microservicios puede ser desafiante. Implementar una gestión adecuada de transacciones y utilizar patrones como SAGA puede ayudar a abordar este problema.

3.3. Curso - Usuario

Este proyecto es un Sistema de Matrículas y Estudiantes desarrollado con Angular y microservicios. Permite la gestión de estudiantes y cursos, así como la matrícula de estudiantes en cursos disponibles.

Especificaciones Funcionales

- **Gestión de Estudiantes**: Creación, actualización, eliminación y visualización de estudiantes.
- **Gestión de Cursos**: Creación, actualización, eliminación y visualización de cursos.
- **Matrícula de Estudiantes**: Los estudiantes pueden matricularse en los cursos disponibles.

3.3.1. Estructura

La estructura del proyecto en Angular es la siguiente:

```
src/
  app/
    asignar-usuario-dialog/
      asignar-usuario-dialog.component.css
      asignar-usuario-dialog.component.html
      asignar-usuario-dialog.component.ts
    curso-form/
      curso-form.component.css
      curso-form.component.html
      curso-form.component.ts
    curso-list/
    curso-usuario/
      curso-usuario.component.css
      curso-usuario.component.html
      curso-usuario.component.ts
    home-page/
      home-page.component.css
      home-page.component.html
      home-page.component.ts
    model/
      curso-usuario.interface.ts
      curso.interface.ts
      usuario.interface.ts
    registrar-usuari-curso/
      registrar-usuari-curso.component.css
      registrar-usuari-curso.component.html
      registrar-usuari-curso.component.spec.ts
      registrar-usuari-curso.component.ts
    services/
```

```
    curso-usuario.service.ts
    curso.service.ts
    usuario.service.ts
  usuario-form/
    custom-validators.ts
    usuario-form.component.css
    usuario-form.component.html
    usuario-form.component.ts
  usuario-list/
    usuario-list.component.css
    usuario-list.component.html
    usuario-list.component.ts
  app.component.css
  app.component.html
  app.component.spec.ts
  app.component.ts
  app.config.server.ts
  app.config.ts
  app.routes.ts
assets/
```

Tener una estructura modular en tu proyecto Angular permite una organización clara y facilita la escalabilidad y el mantenimiento del código. Separar las funcionalidades en componentes específicos (como gestión de estudiantes, cursos y matrículas) y servicios dedicados centraliza la lógica de negocio, promueve la reutilización de código y mejora la legibilidad. Además, una estructura bien definida agiliza el desarrollo colaborativo, ya que cada desarrollador puede trabajar en módulos independientes sin interferir con otras partes del proyecto.

3.3.2. Configuración del Proyecto

1. **Inicialización del Proyecto Angular**: Se creó un nuevo proyecto Angular utilizando Angular CLI

```
ng new CursoUsuario
```

2. **Instalación de Dependencias**: Se instalaron las dependencias necesarias como Angular Material, RxJS, y otros módulos necesarios.

```
ng add @angular/material
```

3.3.3. Desarrollo de Componentes

1. **Componentes de Gestión de Estudiantes**:
    a. *usuario-form.component:* Formulario para crear y actualizar estudiantes.

b. *usuario-list.component:* Lista de estudiantes con opciones para editar y eliminar.

2. **Componentes de Gestión de Cursos**:
   a. *curso-form.component:* Formulario para crear y actualizar cursos.
   b. *curso-list.component:* Lista de cursos con opciones para editar y eliminar.
3. **Componentes de Matrícula**:
   a. *registrar-usuari-curso.component:* Formulario para registrar estudiantes en cursos.

3.3.4. Servicio

1. **Servicios para Estudiantes** (usuario.service.ts):
   Métodos para realizar CRUD (Crear, Leer, Actualizar, Eliminar) operaciones con la API backend.
2. **Servicios para Cursos** (curso.service.ts):
   Métodos para manejar los datos de cursos a través de la API backend.
3. **Servicios para Matrícula** (curso-usuario.service.ts):
   Métodos para manejar la matrícula de estudiantes en cursos.

3.3.5. Validaciones y Enrutamiento

1. **Validaciones Personalizadas** (custom-validators.ts):
   Se implementaron validaciones personalizadas para asegurar la integridad de los datos introducidos en los formularios.
2. **Configuración de Enrutamiento** (app.routes.ts)
   Se configuraron las rutas para navegar entre los diferentes componentes de la aplicación.

3.3.6. Diseño

Estructura de Componentes

- Se decidió estructurar los componentes de manera modular para facilitar la escalabilidad y el mantenimiento del código.
- Cada funcionalidad principal (gestión de estudiantes, gestión de cursos, y matrícula) tiene su propio conjunto de componentes.

```
import { Usuario } from './../model/usuario.interface';
import { CommonModule } from '@angular/common';
import { Component, inject, OnInit } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { CustomValidators } from './custom-validators';
```

```
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { RouterModule, Router, ActivatedRoute } from '@angular/router';
import { UsuarioService } from '../services/usuario.service';

@Component({
  selector: 'app-usuario-form',
  standalone: true,
  imports: [CommonModule, FormsModule, ReactiveFormsModule,
RouterModule],
  templateUrl: './usuario-form.component.html',
  styleUrl: './usuario-form.component.css'
})
export default class UsuarioFormComponent implements OnInit{
  private usuarioService = inject(UsuarioService);
  private router = inject(Router);
  private route = inject(ActivatedRoute);
  registerForm: FormGroup;
  usuarioExiste? : Usuario;

  constructor(private fb: FormBuilder) {
    this.registerForm = this.fb.group({
      nombre: ['', [Validators.required, Validators.minLength(5)]],
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required]],
      repeatPass: ['', [Validators.required]]
    }, { validators: CustomValidators.passwordsMatch('password',
'repeatPass') });
  }

  ngOnInit(): void {
    const id = this.route.snapshot.paramMap.get('id');
    // Implementación del método
  }

  submit() {
    // Implementación del método
  }

  save() {
  //Implementación del método
    }
```

Servicios

- Se implementaron servicios dedicados para cada entidad (estudiantes, cursos, matrícula) para manejar la lógica de negocio y la comunicación con la API backend.
- Los servicios permiten reutilizar el código y centralizar la lógica de negocio.

```
import { HttpClient } from '@angular/common/http';
import { inject, Injectable } from '@angular/core';
import { Usuario } from '../model/usuario.interface';

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

  private http = inject(HttpClient);

  list() {
```

```
    return
this.http.get<Usuario[]>('http://localhost:8001/usuario/Mostrar
Usuarios');
  }

  get(id: number) {
    return
this.http.get<Usuario>(`http://localhost:8001/usuario/BuscarUsu
arioPorID/${id}`);
  }

  create(usuario : Usuario) {
    return
this.http.post<Usuario>('http://localhost:8001/usuario/CrearUsu
ario', usuario);
  }

  update(id: number, usuario: Usuario) {
    return
this.http.put<Usuario>(`http://localhost:8001/usuario/Modificar
Usuario/${id}`, usuario);
  }

  delete(id: number) {
    return
this.http.delete<void>(`http://localhost:8001/usuario/EliminarU
suario/${id}`);
  }

}
```

Validaciones Personalizadas

- Se crearon validaciones personalizadas para los formularios para asegurar que los datos introducidos por los usuarios sean válidos y consistentes.

```
// src/app/custom-validators.ts
import { AbstractControl, ValidationErrors, ValidatorFn } from
'@angular/forms';

export class CustomValidators {
  static passwordsMatch(passwordField: string,
repeatPasswordField: string): ValidatorFn {
    return (control: AbstractControl): ValidationErrors | null
=> {
      const password = control.get(passwordField)?.value;
      const repeatPassword =
control.get(repeatPasswordField)?.value;

      if (password !== repeatPassword) {
        return { passwordsMismatch: true };
      }
      return null;
    };
  }
}
```

CursoUsuario es una parte crucial del sistema de gestión, permitiendo la visualización y gestión de la relación entre cursos y usuarios. La modularidad y la separación de responsabilidades entre componentes y servicios aseguran un código limpio, mantenible y fácil de escalar.

3.4. FrontEnd en Angular

El frontend de este proyecto está desarrollado utilizando Angular, un framework de desarrollo web popular que permite la creación de aplicaciones web dinámicas y robustas. La aplicación facilita la gestión de estudiantes y cursos, así como la matrícula de estudiantes en cursos, utilizando componentes modulares y servicios para interactuar con un backend basado en microservicios.

Especificaciones Funcionales

- **Gestión de Estudiantes**: Creación, actualización, eliminación y visualización de estudiantes.
- **Gestión de Cursos**: Creación, actualización, eliminación y visualización de cursos.
- **Matrícula de Estudiantes**: Los estudiantes pueden matricularse en los cursos disponibles.

3.4.1. Componentes

1. Componente para asignar un usuario a un curso

```
import { Component } from '@angular/core';
import { MatDialogModule, MatDialogRef } from
'@angular/material/dialog';
import { MatFormFieldModule } from
'@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { ReactiveFormsModule, FormBuilder, FormGroup,
Validators } from '@angular/forms';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-asignar-usuario-dialog',
  templateUrl: './asignar-usuario-dialog.component.html',
  standalone: true,
  imports: [
    CommonModule,
    ReactiveFormsModule,
    MatDialogModule,
    MatFormFieldModule,
    MatInputModule
  ]
})
export class AsignarUsuarioDialogComponent {
}
```

2. Componente para el formulario de creación y edición de cursos

```
import { Component, OnInit, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
```

```
import { CursoService } from '../services/curso.service';
import { RouterModule, Router, ActivatedRoute } from
'@angular/router';
import { Usuario } from '../model/usuario.interface';
import { FormBuilder, FormGroup, Validators } from
'@angular/forms';
import { FormsModule, ReactiveFormsModule } from
'@angular/forms';
import { Curso } from '../model/curso.interface';

@Component({
  selector: 'app-curso-form',
  templateUrl: './curso-form.component.html',
  standalone: true,
  imports: [CommonModule, FormsModule, ReactiveFormsModule,
RouterModule],
  styleUrls: ['./curso-form.component.css']
})
export default class CursoFormComponent implements OnInit {
}
```

3. Componente para visualizar los cursos de un usuario específico

```
import { Component, inject, OnInit } from '@angular/core';
import { RouterLink, RouterModule } from '@angular/router';
import { Curso1 } from '../model/curso-usuario.interface';
import { CursoUsuarioService } from
'../services/curso-usuario.service';
import { UsuarioService } from '../services/usuario.service';
import { Usuario } from '../model/usuario.interface';
import { catchError, forkJoin, of } from 'rxjs';
import { Curso } from '../model/curso.interface';


@Component({
  selector: 'app-curso-usuario',
  templateUrl: './curso-usuario.component.html',
  standalone: true,
  imports: [RouterModule, RouterLink],
  styleUrls: ['./curso-usuario.component.css']
})
export default class CursoUsuarioComponent implements OnInit {
}
```

4. Componente para la página de inicio de la aplicación

```
import { Component } from '@angular/core';
import { RouterModule } from '@angular/router';

@Component({
  selector: 'app-home-page',
  standalone: true,
  templateUrl: './home-page.component.html',
  styleUrls: ['./home-page.component.css'],
  imports: [RouterModule]
})
export default class HomePageComponent{

  constructor() { }

}
```

5. Componente para registrar un usuario en un curso

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-registrar-usuari-curso',
  templateUrl: './registrar-usuari-curso.component.html',
  styleUrls: ['./registrar-usuari-curso.component.css']
})
export class RegistrarUsuariCursoComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

6. Componente para el formulario de creación y edición de usuarios

```
import { Usuario } from './../model/usuario.interface';
import { CommonModule } from '@angular/common';
import { Component, inject, OnInit } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from
'@angular/forms';
import { CustomValidators } from './custom-validators';
import { FormBuilder, FormGroup, Validators } from
'@angular/forms';
import { RouterModule, Router, ActivatedRoute } from
'@angular/router';
import { UsuarioService } from '../services/usuario.service';

@Component({
  selector: 'app-usuario-form',
  standalone: true,
  imports: [CommonModule, FormsModule, ReactiveFormsModule,
RouterModule],
  templateUrl: './usuario-form.component.html',
  styleUrl: './usuario-form.component.css'
})
export default class UsuarioFormComponent implements OnInit{
}
```

7. Componente para la visualización de la lista de usuarios

```
import { Usuario } from './../model/usuario.interface';
import { RouterLink, RouterModule } from '@angular/router';
import { UsuarioService } from './../services/usuario.service';
import { Component, inject, OnInit } from '@angular/core';

@Component({
  selector: 'app-usuario-list',
  standalone: true,
  imports: [RouterModule, RouterLink],
  templateUrl: './usuario-list.component.html',
  styleUrl: './usuario-list.component.css'
})
export default class UsuarioListComponent implements OnInit {
}
```

8. Enrutamiento

```
import { HttpClient } from '@angular/common/http';
import { inject, Injectable } from '@angular/core';
import { Usuario } from '../model/usuario.interface';

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

  private http = inject(HttpClient);

  list() {
    return
this.http.get<Usuario[]>('http://localhost:8001/usuario/Mostrar
Usuarios');
  }

  get(id: number) {
    return
this.http.get<Usuario>(`http://localhost:8001/usuario/BuscarUsu
arioPorID/${id}`);
  }

  create(usuario : Usuario) {
    return
this.http.post<Usuario>('http://localhost:8001/usuario/CrearUsu
ario', usuario);
  }

  update(id: number, usuario: Usuario) {
    return
this.http.put<Usuario>(`http://localhost:8001/usuario/Modificar
Usuario/${id}`, usuario);
  }

  delete(id: number) {
    return
this.http.delete<void>(`http://localhost:8001/usuario/EliminarU
suario/${id}`);
  }

}
```

La estructura, los componentes, servicios, y enrutamiento del frontend de tu aplicación Angular para la gestión de matrículas y estudiantes. Puedes expandir cada sección con más detalles específicos de tu implementación y agregar ejemplos de uso, capturas de pantalla, y descripciones adicionales si es necesario.

3.4.2. Visualización de Interfaces del aplicativo

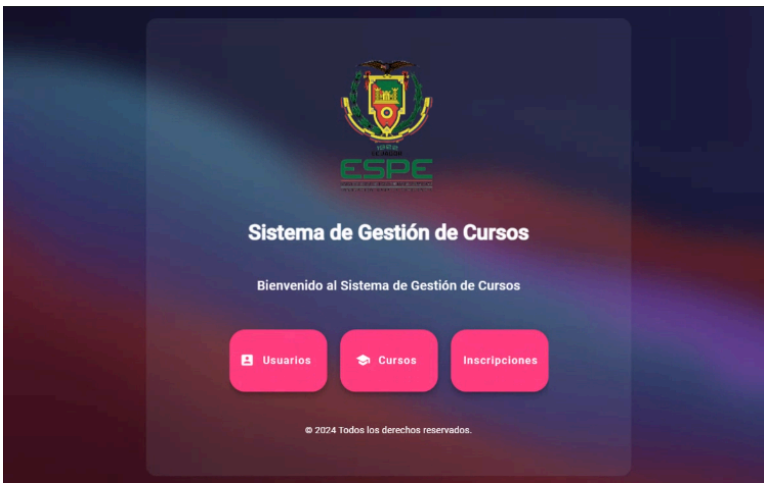

Fig. 1. Interfaz-Pantalla Inicio

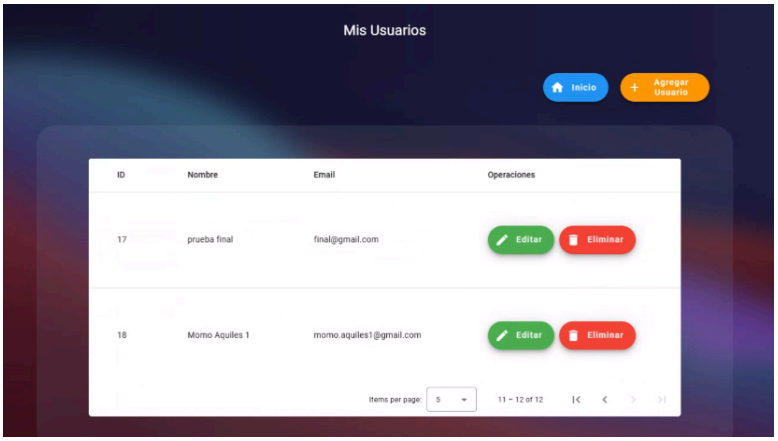

Fig. 2. Interfaz-Pantalla Listado “Mis Usuarios”

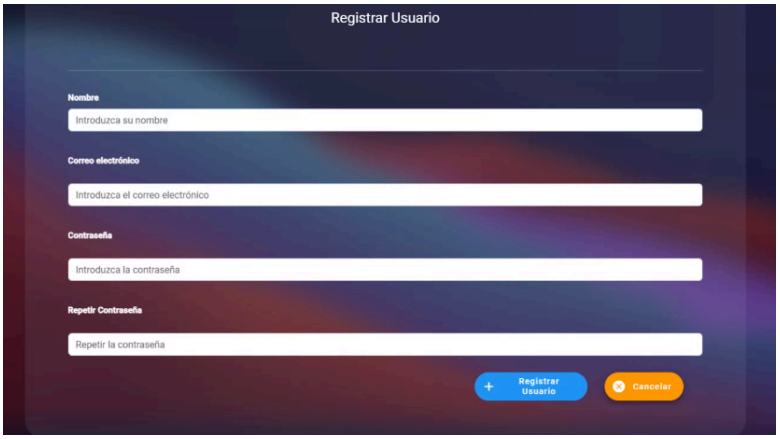

Fig. 3. Interfaz-Pantalla Formulario Registro Usuarios

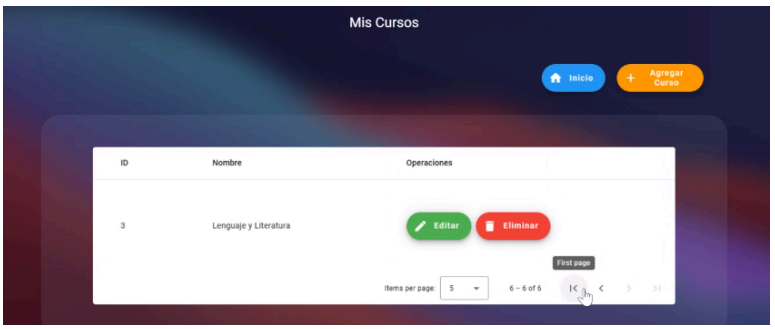

Fig. 4. Interfaz-Pantalla Listado “Mis Cursos”

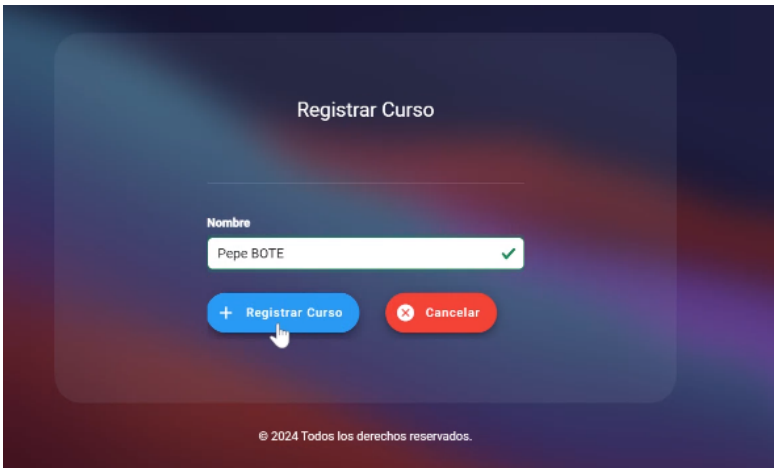

Fig. 5. Interfaz-Pantalla Formulario Registro Cursos

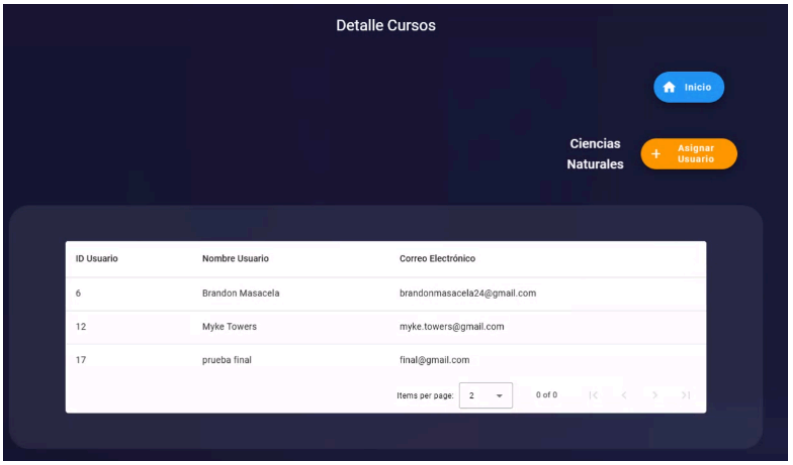

Fig. 6. Interfaz-Pantalla Listado de Inscripciones

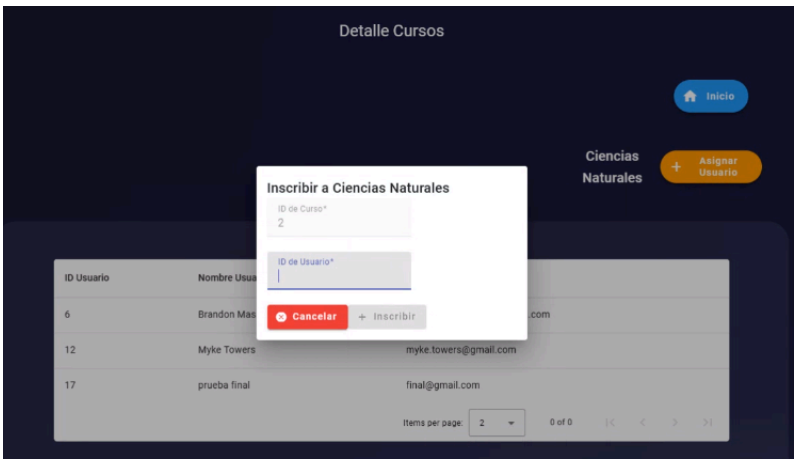

Fig. 7. Interfaz-Pantalla PopUp Nueva Inscripción

4. Diagramas

Al realizar ambos diagramas son esenciales para la planificación, diseño, implementación y mantenimiento de sistemas de software complejos. Proporcionan claridad, mejoran la comunicación y aseguran que el sistema esté bien diseñado y sea eficiente, escalable y mantenible.

4.1. Arquitectura

El diagrama de arquitectura ofrece una representación visual de los distintos componentes que forman un sistema y muestran cómo se comunican e interactúan entre sí.

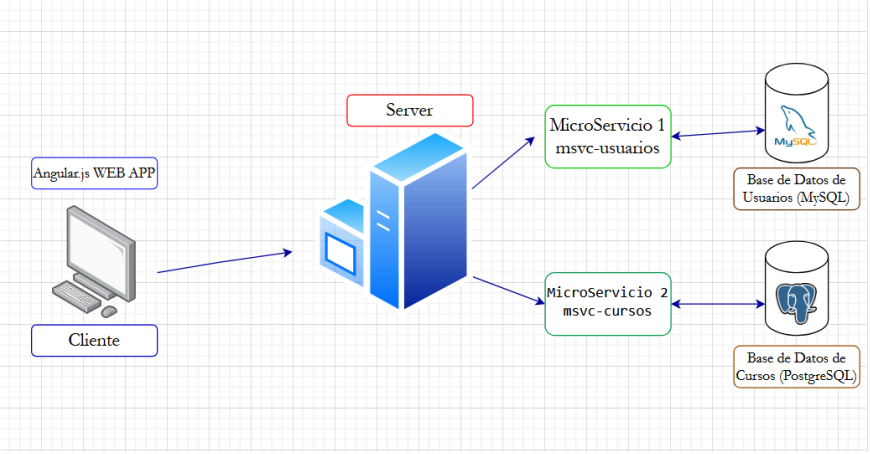

Fig. 8. Diagrama de arquitectura

4.2. Flujo de Datos

El diagrama de flujo de datos traza el flujo de la información para cualquier proceso o sistema. Empleando símbolos, como rectángulos, círculos y flechas, además de etiquetas de texto breves, para mostrar las entradas y salidas de datos, los puntos de almacenamiento y las rutas entre cada destino.

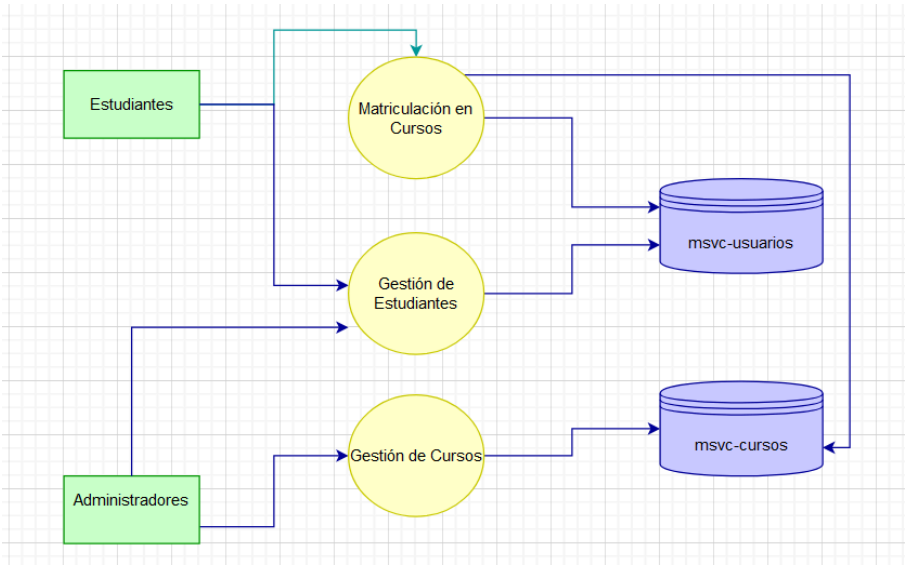

Fig. 9. Diagrama de flujo

5. Conclusiones

El uso de una arquitectura basada en microservicios y Angular ha demostrado ser una solución efectiva y escalable en el desarrollo del Sistema de Matrículas y Gestión de Estudiantes. La adopción de microservicios ha permitido que cada componente del sistema funcione de manera independiente, lo cual facilita significativamente el mantenimiento y la actualización de los servicios sin interrumpir la operación general del sistema. Esta independencia de componentes ha sido crucial para mejorar la modularidad y la capacidad de adaptación del sistema a futuras necesidades y cambios.

Por otro lado, Angular ha proporcionado una plataforma robusta y dinámica para el desarrollo del frontend. Gracias a sus interfaces intuitivas y responsivas, ha sido posible crear una experiencia de usuario fluida y agradable. La capacidad de Angular para manejar datos en tiempo real y su estructura basada en componentes han permitido el desarrollo de una interfaz de usuario coherente y eficiente.

La combinación de estas tecnologías ha resultado en un sistema robusto e intuitivo de utilizar. La separación clara entre el frontend y los servicios backend ha permitido un desarrollo más organizado y eficiente, reduciendo los tiempos de respuesta y mejorando la experiencia general tanto para los usuarios finales como para los

desarrolladores. En resumen, la integración de microservicios y Angular ha demostrado ser una estrategia exitosa para construir aplicaciones web modernas, escalables y fáciles de mantener.

6. Recomendaciones

- **Seguridad:** Implementar medidas de seguridad robustas, como autenticación, autorización y encriptación, para proteger las comunicaciones remotas y prevenir ataques.
  Asegurarse de que todas las comunicaciones entre el frontend y el backend estén encriptadas utilizando protocolos seguros como HTTPS.
- **Gestión de Errores:** Diseñar un sistema de manejo de errores eficaz que contemple la recuperación de fallos y tiempos de espera, para mejorar la resiliencia del sistema.
  Proveer mensajes de error claros y útiles para el usuario final, con recomendaciones para resolver problemas cuando sea posible.
- **Proceso de pruebas:** Verificar que las pruebas realizadas cumplan con las expectativas al momento del desarrollo del sistema.
- **Capacitación y Documentación**: Proporcionar formación adecuada a los desarrolladores sobre el uso de angular con microservicios y mantener una documentación detallada del proceso de implementación para facilitar el mantenimiento y la escalabilidad del sistema.
  Incorporar pruebas de aceptación de usuario para validar que el sistema cumple con los requisitos y expectativas del usuario final. Recoger y analizar feedback para realizar ajustes necesarios.

7. Bibliografía

- Gonçalves, M. J. (2024, 4 junio). *¿Qué es Angular y para qué sirve?* Blog de Hiberus. https://www.hiberus.com/crecemos-contigo/que-es-angular-y-para-que-sirve/
- *¿Qué son los microservicios? | AWS*. (s. f.). Amazon Web Services, Inc. https://aws.amazon.com/es/microservices/