



UNIVERSIDAD AUTÓNOMA DE SAN LUIS POTOSÍ

FACULTAD DE INGENIERÍA

INGENIERÍA EN SISTEMAS INTELIGENTES

APLICACIONES WEB ESCALABLES

SEMESTRE 2022-2023/I

RODRÍGUEZ GONZÁLEZ FERNANDO DE JESÚS

MANUAL DE PROGRAMADOR – DIARIO DE SERIES

ING. ESTRADA VELAZQUEZ FRANCISCO EVERARDO

28 DE NOVIEMBRE DE 2022

INTRODUCCIÓN

Para la elaboración de este sistema de “Diario de series” realizado para la materia de Aplicaciones Web Escalables se utilizaron distintos recursos orientados al desarrollo de sistemas web de forma escalable, es decir, separando la parte del cliente y el servidor. Se usaron frameworks, librerías y otros recursos que serán explicados más adelante, para que en caso de que se desee retomar este proyecto para darle seguimiento o añadirle más funcionalidades que en un futuro pudieran ser importantes, se pueda hacer y seguir con esta misma estructura.

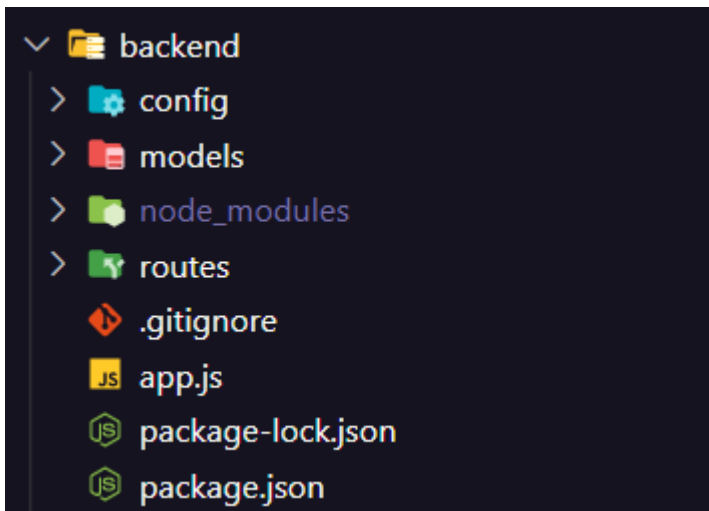
BACKEND

En la parte del backend se optó por utilizar NodeJS, el entorno de ejecución de JavaScript basado en una arquitectura no bloqueante y manejada por eventos para ayudar a los desarrolladores a crear aplicaciones distribuidas robustas. Es bastante rápido en comparación a otras tecnologías similares y esta pensando para la web, nos permite manejar múltiples peticiones en la web. Además, se uso ExpressJS que es una infraestructura de aplicaciones web que usa precisamente NodeJS que proporciona un conjunto solido de características para las aplicaciones web y móviles.

DESARROLLO Y CÓDIGO

Dentro de la carpeta raíz del proyecto, podremos encontrar la carpeta dedicada al Backend.

Dentro de la misma, encontraremos lo siguiente:



CONFIG

Nos encontraremos con dos archivos: config.env y db.js.

En el primero vendrá especificado el puerto por el cual se estará ejecutando el proyecto, así como la URL para la conexión a la Base de Datos de Mongo.

```
1 PORT=3000
2 MONGO_URI = mongodb://localhost:27017/proyecto
```

En el segundo nos encontraremos con el código necesario para hacer la conexión con la base de datos de MongoDB. Tendremos que importar Mongoose que es una librería de node que nos hace mucho más fácil la conexión con la DB. Dentro de este, se declara una función asíncrona para conectarse a la DB.

```
const mongoose = require('mongoose');

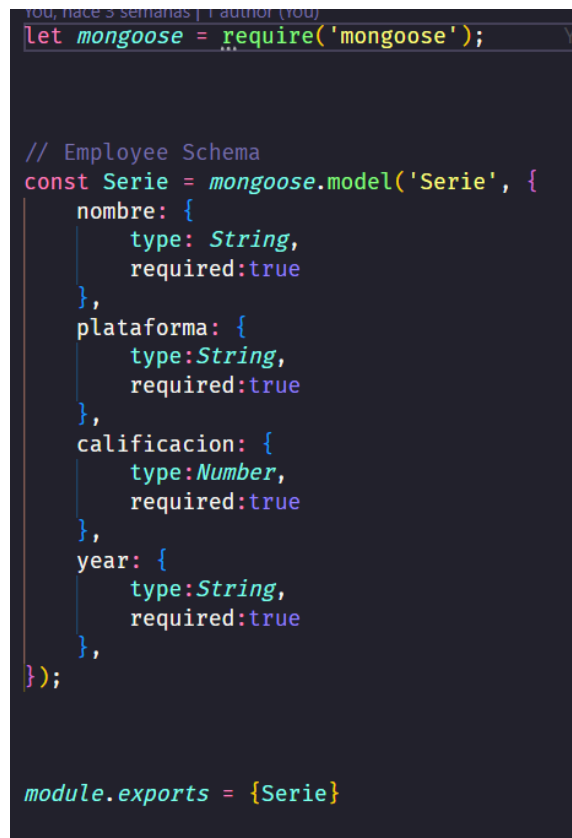
const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
      useFindAndModify: false
    });
    console.log(`Mongo DB Connected: ${conn.connection.host}`);
  } catch(err) {
    console.log(err);
    process.exit(1);
  }
}

module.exports = connectDB;
```

MODELS

Dentro de esta carpeta solo nos encontramos con el archivo serie.js.

En él, encontraremos la declaración del modelo que se utilizara, en este caso Serie, ahí se declara el esquema del modelo, sus campos, tipos y si es requerido o no.



```
let mongoose = require('mongoose');

// Employee Schema
const Serie = mongoose.model('Serie', {
  nombre: {
    type: String,
    required: true
  },
  plataforma: {
    type: String,
    required: true
  },
  calificacion: {
    type: Number,
    required: true
  },
  year: {
    type: String,
    required: true
  },
});

module.exports = {Serie}
```

NODE MODULES

Carpeta necesaria para el almacenamiento de dependencias para el funcionamiento del proyecto. Dentro del repositorio no la encontraremos y bastara con ejecutar el comando “npm i” para hacer la instalación de dependencias y crear esta carpeta.

ROUTES

Esta el archivo index.js. Ahí se exporta express que ya se explico antes de que se trata. Después de ello hacemos la importación del router que viene con express.

```
const express = require('express');
const router = express.Router();
const ObjectId = require('mongoose').Types.ObjectId;

const { Serie } = require('../models/serie');
```

Después, para cada ruta que queramos se hace una función, especificando el tipo de petición, la url desde la que se podrá acceder a esta función y ya después el proceso correspondiente.

Dentro de este proyecto nos encontramos con las funciones básicas de una API REST como lo es el index, show, store, update y delete.

Dentro de index y show que es mostrar tanto el listado de series como una sola serie podemos ver que son funciones find, una con parámetros y una sin parámetros para que según el id pueda ser localizada y así mostrarla al usuario.

```
// Get All Employees
router.get('/api/series', (req, res) => {
  Serie.find({}, (err, data) => {
    if(!err) {
      res.send(data);
    } else {
      console.log(err);
    }
  });
});

// Get Single Employee (First Way)

router.get('/api/serie/:id', (req, res) => {
  Serie.findById(req.params.id, (err, data) => {
    if(!err) {
      res.send(data);
    } else {
      console.log(err);
    }
  });
});
```

```

router.post('/api/serie/add', (req, res) => {
  const emp = new Serie({
    nombre: req.body.nombre,
    plataforma: req.body.plataforma,
    calificacion: req.body.calificacion,
    year: req.body.year,
  });
  emp.save((err, data) => {
    if(!err) {
      // res.send(data);
      res.status(200).json({code: 200, message: 'Serie añadido exitosamente', addSerie: data})
    } else {
      console.log(err);
    }
  });
});

// Update Employee
router.put('/api/serie/update/:id', (req, res) => {

  const ser = {
    nombre: req.body.nombre,
    plataforma: req.body.plataforma,
    calificacion: req.body.calificacion,
    year: req.body.year,
  };
  Serie.findByIdAndUpdate(req.params.id, { $set: ser }, { new: true }, (err, data) => {
    if(!err) {
      res.status(200).json({code: 200, message: 'Serie Updated Successfully', updateSerie: data})
    } else {
      console.log(err);
    }
  });
});

```

En tanto el store es una petición post en donde recibimos un objeto de tipo serie declarado en la carpeta models en series.js, ahí hacemos un save en la BD. En update tenemos una petición put donde recibimos también un objeto de tipo serie, pero además un id para buscar el elemento y así poder actualizarlo.

En el archivo app.js ya dentro de la carpeta raíz de backend nos encontramos con lo necesario para poner a correr nuestro servidor, hacemos las respectivas importaciones como express, el body-parser el cual sirve para transformar el cuerpo de cualquier petición a tipo JSON para así hacerla más entendible y poder guardarla fácilmente. También importamos cors para los permisos de cors y no tener problemas al momento de estar haciendo peticiones y enviar o recibir respuestas. Además dotenv para el manejo del archivo de configuración del servidor.

```

const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const dotenv = require('dotenv');

```

Después de ello, hacemos la conexión a la BD mediante la función antes declarada y declaramos las rutas con el archivo que las contiene. Finalmente tenemos el listen de la app en el puerto 3000.

```

const connectDB = require('./config/db');
// Load Config
dotenv.config({path: './config/config.env'})

connectDB();

// Routes
app.use('/', require('./routes/index'));
app.use('/api/series', require('./routes/index'));
app.use('/api/serie/add', require('./routes/index'));
app.use('/api/serie/:id', require('./routes/index'));
app.use('/api/serie/update/:id', require('./routes/index'));
app.use('/api/serie/:id', require('./routes/index'));

app.listen(3000);

```

FRONTEND

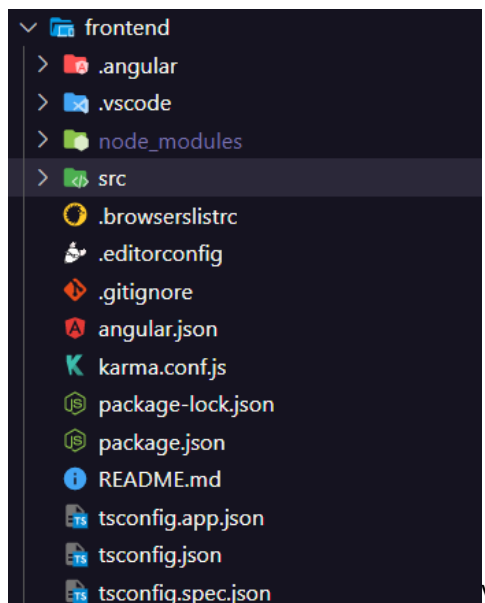
FRAMEWORK UTILIZADO

Para la parte del frontend, se decidió usar el framework Angular para así estar trabando con el stack MEAN. Además, el framework es fácil de usar y tiene una curva de aprendizaje muy pequeña para quienes van empezando en este tipo de frameworks. Angular es una plataforma de desarrollo, construida sobre TypeScript. Es un framework basado en componentes para crear aplicaciones web escalables. Una colección de bibliotecas bien integradas que cubren una amplia variedad de características, que incluyen enrutamiento, administración de formularios, comunicación cliente-servidor y más. Un conjunto de herramientas para desarrolladores que permiten desarrollar, compilar, probar y actualizar el código fuente de la aplicación.

Entre las dependencias utilizadas tenemos lo que es DRACULA UI, una librería de css para estilos de tipo dark mode, inspirado en el famoso tema de VSCode Dracula. Además de Bootstrap e Icons Bootstrap. Material Data Tables para el listado. Angular routing para el ruteo de componentes.

DESARROLLO Y CÓDIGO

Dentro de la carpeta raíz de frontend nos encontramos la siguiente estructura:



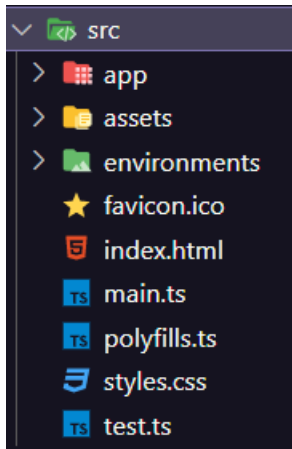
Las carpetas de .angular y .vscode son carpetas de configuración para el editor de texto y algunas configuraciones de angular. La carpeta de node_modules como ya se menciono antes, contiene todas las dependencias necesarias para el funcionamiento del proyecto y basta con hacer “npm i” para instalarlas y crear esta carpeta.

ARCHIVOS

Los archivos localizados debajon de la carpeta de src son archivos de configuración a los cuales no se les necesito modificar nada para el funcionamiento del proyecto.

SRC

La carpeta src es la más importante de nuestro proyecto y tiene la siguiente estructura:



ENVIROMENTS

Contiene archivos y variables de configuración para links de API, API KEYS entre otras cosas necesarias para el modo producción del proyecto.

Los archivos index.html, main.ts, polyfills.ts, test.ts y favicon.ico no se modifico gran cosa por lo que recomiendo mantenerlos así.

Al archivo styles.css se le agrego la importación de los iconos de Bootstrap para que estos pudieran ser usados globalmente en el proyecto. Además de declararon algunos otros estilos globales:

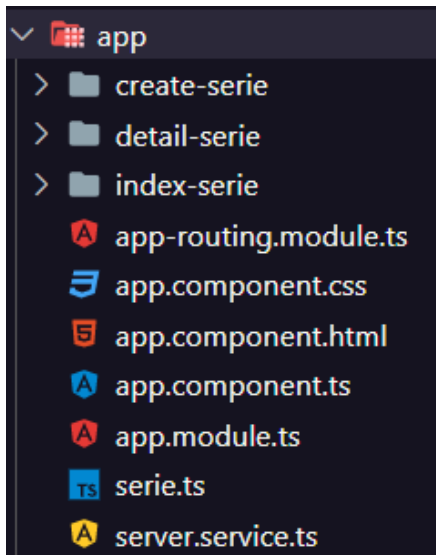
```
@import "~bootstrap-icons/font/bootstrap-icons.css";

body {
  background-color: #282A36 !important;
}
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

.custom-modalbox > mat-dialog-container {
  background-color: #282A36 !important;
}
```


Carpeta APP

Esta carpeta contiene todos los archivos del código fuente necesarios para la ejecución del proyecto y tiene la siguiente estructura:



App-routing.module.ts

Contiene las rutas del proyecto, junto con sus urls y los componentes que se deben mostrar cuando se ingrese a esa ruta:

```
import { Injectable, NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { CreateSerieComponent } from './create-serie/create-serie.component';
import { DetailSerieComponent } from './detail-serie/detail-serie.component';
import { IndexSerieComponent } from './index-serie/index-serie.component';

const routes: Routes = [
  {
    path: 'series',
    component: IndexSerieComponent
  },
  {
    path: 'series/serie',
    component: CreateSerieComponent
  },
  {
    path: 'series/serie/:id',
    component: CreateSerieComponent
  },
  {
    path: 'series/:id',
    component: DetailSerieComponent
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Como se puede ver, se deben importar los componentes a rutear, así como en el module hacer la respectiva importación y exportación.

App.component.html

Contiene el html base que se mostrara siempre como puede ser el menú o la barra de navegación.

```
<nav class="navbar navbar-dark navbar-expand-lg drac-bg-black">
  <div class="container-fluid">
    <a class="navbar-brand drac-text-white" href="#">Diario de Series</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent">
      <span class="navbar-toggler-icon drac-text-white"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="drac-tabs-white nav justify-content-center">
        <li class="drac-tab" [routerLinkActive]='["drac-tab-active"]' [routerLinkActiveOptions]='{ exact: true }">
          <a class="drac-tab-link drac-text" routerLink="/">Inicio</a>
        </li>
        <li class="drac-tab" [routerLinkActive]='["drac-tab-active"]' [routerLinkActiveOptions]='{ exact: true }">
          <a class="drac-tab-link drac-text" routerLink="series">Series</a>
        </li>
      </ul>
    </div>
  </div>
</nav>

<router-outlet></router-outlet>
```

Como se ve, al final tiene las etiquetas de router-outlet que es ahí donde se incrustara el componente según la ruta a la que se este accediendo.

App.module.ts

Es uno de los archivos más importantes, pues ahí es donde se va declarando todo lo necesario para el funcionamiento del proyecto, como componentes, librerías, módulos, entre otras cosas, se deben hacer las declaraciones, importaciones y cosas que sean necesarias para que se reflejen en los componentes que se vayan creando y poder usarlos. Digamos que es un archivo de configuración.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NoopAnimationsModule } from '@angular/platform-browser/animations';
import { MatTableModule } from '@angular/material/table';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatPaginatorModule } from '@angular/material/paginator';
import { MatInputModule } from '@angular/material/input';
import { MatSelectModule } from '@angular/material/select';
import { MatIconModule } from '@angular/material/icon';
import { MatSortModule } from '@angular/material/sort';
import { MatDialogModule } from '@angular/material/dialog';
import { CreateSerieComponent } from './create-serie/create-serie.component';
import { IndexSerieComponent, DialogAnimationsExampleDialog } from './index-serie/index-serie.component';
import { DetailSerieComponent } from './detail-serie/detail-serie.component';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
    CreateSerieComponent,
    IndexSerieComponent,
    DetailSerieComponent,
    DialogAnimationsExampleDialog
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    AppRoutingModule,
    NoopAnimationsModule,
    MatTableModule,
    MatFormFieldModule,
    MatPaginatorModule,
```

Serie.ts

Tiene el modelo de la BD a utilizar dentro del proyecto, es muy parecido al declarado en el backend.

```
export interface Serie {  
  id:string;  
  nombre:string;  
  calificacion:number;  
  plataforma:string;  
  year:string;  
}
```

Server.service.ts

Es el archivo por el cual se estarán haciendo todas las peticiones a nuestro servidor backend, esta generalizado a tener las cuatro funciones más importantes como lo son GET, POST, PUT y DELETE.

Tiene declarado una URL_API que será constante para todas las peticiones y en cada función se debe recibir la url restante o especifica para hacer la petición.

```
URL_API = 'http://localhost:3000/api';
```

```
getServer(url: string) {  
  return new Promise((resolve, reject) => {  
    this.http.get(this.URL_API + url).subscribe(  
      (res) => {  
        resolve(res);  
      }, (err) => {  
        reject(err);  
      }  
    );  
  });  
}  
  
postServer(url: string, params: any) {  
  return new Promise((resolve, reject) => {  
    this.http.post(this.URL_API + url, params).subscribe(  
      (res) => {  
        resolve(res);  
      }, (err) => {  
        reject(err);  
      }  
    );  
  });  
}
```

```

putServer(url: string, params: any) {
  return new Promise((resolve, reject) => {
    this.http.put(this.URL_API + url, params).subscribe(
      (res) => {
        resolve(res);
      }, (err) => {
        reject(err);
      }
    );
  });
}

deleteServer(url: string) {
  return new Promise((resolve, reject) => {
    this.http.delete(this.URL_API + url).subscribe(
      (res) => {
        resolve(res);
      }, (err) => {
        reject(err);
      }
    );
  });
}

```

COMPONENTES

Nos encontramos con 2 carpetas más, create-serie e index-serie. Ambas carpetas corresponden a componentes creados para el proyecto mediante el comando `ng g c nombre`.

Cada carpeta se compone de tres archivos, uno de html, otro de ts y uno de css.

Dentro de cada archivo se hace lo necesario en cada lenguaje para el funcionamiento del componente.

Create-serie

Dentro del html se hace toda la maquetación de inputs, alertas, grid, selects y opciones para el funcionamiento.

```

<div class="drac-boc drac-card drac-bg-purple drac-p-md">
  <div class="row">
    <h2 class="drac-heading drac-heading-xl drac-text-white">{{title}}</h2>
  </div>

  <div class="row">
    <div class="col-xl-4 col-md-4 col-12">
      <label for="nombre">Nombre</label>
      <input
        placeholder="introduce un valor"
        class="drac-input drac-input-white drac-text-white"
        [(ngModel)]="serie.nombre"
      />
    </div>

    <div class="col-xl-4 col-md-4 col-12">
      <label for="nombre">Plataforma</label>
      <div style="position: relative">
        <select class="drac-select drac-select-white" [(ngModel)]="serie.plataforma">
          <option value="default" disabled="" selected="">Selecciona una plataforma</option>
          <option value="Netflix">Netflix</option>
          <option value="Prime Video">Prime Video</option>
          <option value="HBO Max">HBO Max</option>
          <option value="Star+">Star+</option>
          <option value="Disney+">Disney+</option>
          <option value="Apple TV+">Apple TV+</option>
          <option value="VIX+">VIX+</option>
        </select>
      </div>
    </div>
  </div>

```

Dentro del archivo ts, se declara el componente con todas su extensiones:

```
@Component({
  selector: 'app-create-serie',
  templateUrl: './create-serie.component.html',
  styleUrls: ['./create-serie.component.css']
})
```

Tanto su selector, la referencia al template que en este caso es el archivo HTML y la url de los estilos que es el archivo css.

Ya dentro de la clase o el componente, tenemos primero dos variables, una de titulo y otra de buttonText estas cambian de valor según sea el caso. Ya sea para editar la serie o para crearla, pues este componente cumple con las dos funciones, el componente cambia de tipo cuando se detecta que en la URL viene el id. Si viene la ID se pone a tipo editar y se debe obtener el valor de la serie mediante una petición get al servidor pasando el id como parámetro.

```
const id = this._route.snapshot.paramMap.get('id');
console.log(id);
if(id !== null){
  this._ServerService.getServer('/serie/'+id).then(
    (data:any) => {
      console.log(data);
      this.title = 'Editar serie';
      this.buttonText = 'Guardar serie';
      this.serie = data;
    }, (error:any) => {
      console.log(error);
    }
  )
}
```

El resultado de la petición se asigna a nuestro modelo serie para que así los datos se muestren en los inputs.

Al guardar los datos, otra vez se valida que caso es, si es editar o agregar. Se checa que todos los campos estén llenos y se manda la petición, ya sea post o put. En dado caso de éxito, se muestra la alerta de éxito y se redirige a la lista principal.

```

const id = this._route.snapshot.paramMap.get('id');
if(this.serie.nombre !== '' && this.serie.plataforma !== '' && this.serie.year !== ''){
  if(id===null){
    this._ServerService.postServer('/serie/add',this.serie).then(
      (data:any) => {
        this.alert = true;
        this.textAlert = "Serie añadida correctamente";
        setTimeout(async () => {
          this.alert = false;
        }, 8000);
        setTimeout(() => {
          this._router.navigateByUrl('series');
        }, 2000);
        console.log(data);
        this.title = 'Editar serie';
        this.buttonText = 'Guardar serie';
      }, (error:any) => {
        console.log(error);
      })
  }
}
}else{
  this._ServerService.putServer('/serie/update/'+id,this.serie).then(

```

Index-serie

Esta carpeta es la del componente del listado, aquí, además de los tres archivos típicos de componente (HTML,CSS,TS) se agrega un html más, este corresponde a la alerta de confirmación de eliminación de un registro.

```

<h1 mat-dialog-title class="drac-text-white">Eliminar serie</h1>
<div mat-dialog-content class="drac-text-white">
  ¿Seguro que quieres eliminar esta serie? Ya no la podras recuperar
</div>
<div mat-dialog-actions class="drac-text-white drac-p-sm d/felx justify-content-end">
  <button mat-button mat-dialog-close class=" drac-text-white drac-btn drac-bg-purple drac-mx-auto">Cancelar</button>
  <br>
  <button mat-button mat-dialog-close cdkFocusInitial class=" drac-text-white drac-btn drac-bg-purple drac-mx-auto" (click)="onConfirm()">Eliminar</button>
</div>

```

Pasando al archivo css, tenemos algunos estilos correspondientes a lo que es la tabla, para que ocupe un 100% de su contenedor, estilos para el buscador, además de un estilo para el scroll de la tabla y que si tiene overflow pueda tener el scroll.

Dentro del HTML, tenemos de arriba hacia abajo, la alerta, que es un div y solo se muestra si la variable alert es en true, esto lo hacemos con la directiva de angular *ngIf. Después tenemos la card con el titulo y el input para el buscador.

El botón de añadir serie que tiene la directica (click) para que cuando se presione, este lo redireccione al formulario de añadir serie.

Dentro de la tabla, ponemos la directiva mat-table que sirve para hacer esta tabla dinámica indicando el arreglo de donde se tomaran los datos a mostrar. Para cada columna, se declara un ng-container se coloca el encabezado que queremos que se muestre y el dato a mostrar del objeto mediante la interpolación.

```

<div class="offset-md-1 col-md-10">
  <div class="alert alert-primary" role="alert" *ngIf="alert">
    {{textAlert}}
  </div>
  <div class="drac-boc drac-card drac-bg-purple drac-p-md">
    <div class="row">
      <div class="col-12 col-md-6 col-lg-6">
        <mat-label class="h3">Series que vi o vere</mat-label>
        <input (keyup)="applyFilter($event)" #input placeholder="Buscar" class="drac-input drac-input-white drac-text-white">
      </div>
      <div class="butt offset-xl-3 offset-md-3 mt-md-2 col-12 col-md-3 col-xl-3 d-flex justify-content-end">
        <button class="drac-btn drac-bg-orange drac-mx-sm my-sm-3 my-3 my-md-4 completo" (click)="agregarSerie()">
          Nueva serie
        </button>
      </div>
    </div>
  </div>
</div>

```

```

<ng-container matColumnDef="name">
  <th mat-header-cell *matHeaderCellDef mat-sort-header class="drac-text drac-text-white"> Serie </th>
  <td mat-cell *matCellDef="let row" class="drac-text drac-text-white"> {{row.nombre}} </td>
</ng-container>

<!-- Name Column -->
<ng-container matColumnDef="plataforma">
  <th mat-header-cell *matHeaderCellDef mat-sort-header class="drac-text drac-text-white"> Plataforma </th>
  <td mat-cell *matCellDef="let row" class="drac-text drac-text-white"> {{row.plataforma}} </td>
</ng-container>

<!-- Fruit Column -->
<ng-container matColumnDef="year">
  <th mat-header-cell *matHeaderCellDef mat-sort-header class="drac-text drac-text-white"> Año </th>
  <td mat-cell *matCellDef="let row" class="drac-text drac-text-white"> {{row.year}} </td>
</ng-container>

<ng-container matColumnDef="calificacion">
  <th mat-header-cell *matHeaderCellDef mat-sort-header class="drac-text drac-text-white"> Calificación </th>
  <td mat-cell *matCellDef="let row" class="drac-text drac-text-white"> {{row.calificacion}} </td>
</ng-container>

```

```

<!-- Row shown when there is no matching data. -->
<tr class="mat-row" *matNoDataRow class="drac-text drac-text-white">
  <td class="mat-cell" colspan="4" class="drac-text drac-text-white">No data matching the filter "{{input.value}}"</td>
</tr>
</table>

</div>
<mat-paginator class="drac-bg-black-secondary drac-text drac-text-white" [pageSizeOptions]="[10, 25, 50]"></mat-paginator>

```

En la parte del archivo TS tenemos las importaciones necesarias para el funcionamiento del componente, como lo es el AfterViewInit, Component, ViewChild, OnInit. Plugins de material como el paginator, sort, la datatable, el MatDialog.

```

import {AfterViewInit, Component, ViewChild, OnInit} from '@angular/core';
import {MatPaginator} from '@angular/material/paginator';
import {MatSort} from '@angular/material/sort';
import {MatIcon} from '@angular/material/icon';
import {MatTableDataSource} from '@angular/material/table';
import {Router} from '@angular/router';
import {ServerService} from '../server.service';
import {Serie} from '../serie';
import {MatDialog, MatDialogRef} from '@angular/material/dialog';

```

Dentro del componente, tenemos lo que es el arreglo de columnas a usar con la tabla, las columnas que declaremos aquí son las que tenemos que usar en la tabla.

Declaramos el arreglo de datos para la MatTableDataSource.

```
displayedColumns: string[] = ['name', 'plataforma', 'year', 'calificacion', 'actions'];
dataSource!: MatTableDataSource<Serie>;
series: any;

@ViewChild(MatPaginator) paginator :any = MatPaginator;
@ViewChild(MatSort) sort: any = MatSort;

alert: boolean = false;
textAlert: string = '';

constructor(private router: Router, private _ServerService: ServerService, public dialog: MatDialog) {
}
```

En la función de ngAfterViewInit, que se ejecuta justo al terminar de cargarse todos los elementos de la pagina, se hace la petición get para obtener todo el listado de series, cuando obtenemos una respuesta positiva, lo asignamos a las variables de series y datasource para que así se puedan desplegar dentro de la tabla. Ponemos la etiqueta del paginador y obtenemos los datos para el mismo.

```
ngAfterViewInit() {
  this._ServerService.getServer('/series').then(
    (data: any) => {
      console.log(data);
      this.series = data;
      this.dataSource = new MatTableDataSource<Serie>(this.series);
      this.paginator._intl.itemsPerPageLabel = 'Series por página';
      this.paginator._intl.getRangeLabel = this.getRangeLabel;
      this.dataSource.paginator = this.paginator;
      this.dataSource.sort = this.sort;
    }, (error: any) => {
      console.log(error);
    }
  );
}
```

Funciones como applyFilter, getRangeLabel son funciones auxiliares para el uso e implementación de la tabla.


```

applyFilter(event: Event) {
  const filterValue = (event.target as HTMLInputElement).value;
  this.dataSource.filter = filterValue.trim().toLowerCase();

  if (this.dataSource.paginator) {
    this.dataSource.paginator.firstPage();
  }
}

getRangeLabel = (page: number, pageSize: number, length: number): any => {
  if (length === 0 || pageSize === 0) {
    return '0 de ' + length;
  }
  length = Math.max(length, 0);
  const startIndex = page * pageSize;
  const endIndex = startIndex < length ?
    Math.min(startIndex + pageSize, length) :
    startIndex + pageSize;
  return startIndex + 1 + ' - ' + endIndex + ' de ' + length;
};

```

Funciones agregarSerie y editSerie son las que nos redirigen a la vista para agregar y editar series respectivamente, aquí hacemos uso del router para que por medio de la URL dada, nos lleve a esa dirección.

```

agregarSerie(): void{
  this.router.navigateByUrl('/series/serie');
}

editSerie(id:number): void{
  this.router.navigateByUrl('/series/serie/'+id);
}

```

OpenDialog es la función que nos abre la pequeña alerta para confirmar la eliminación de un registro de la tabla de series.

```

openDialog(enterAnimationDuration: string, exitAnimationDuration: string, id:string): void {
  this.dialog.open(DialogAnimationsExampleDialog, {
    width: '350px',
    panelClass: 'custom-modalbox',
  }).afterClosed()
  .subscribe((confirmado: Boolean) => {
    if (confirmado) {
      this._ServerService.deleteServer('/serie/'+id).then(
        (data:any) => {
          console.log(data);
          this._ServerService.getServer('/series').then(
            (data:any) => {
              console.log(data);
              this.series = data;
              this.dataSource = new MatTableDataSource<Serie>(this.series);
              this.paginator._intl.itemsPerPageLabel = 'Series por página';
              this.paginator._intl.getRangeLabel = this.getRangeLabel;
              this.dataSource.paginator = this.paginator;
              this.dataSource.sort = this.sort;
            }, (error: any) => {
              console.log(error);
            }
          );

          this.alert = true;
          this.textAlert = "Serie eliminada correctamente";
          setTimeout(async () => {
            this.alert = false;
          }, 8000);
        }, (error: any) => {
          console.log(error);
        }
      );
    }
  });
}

```

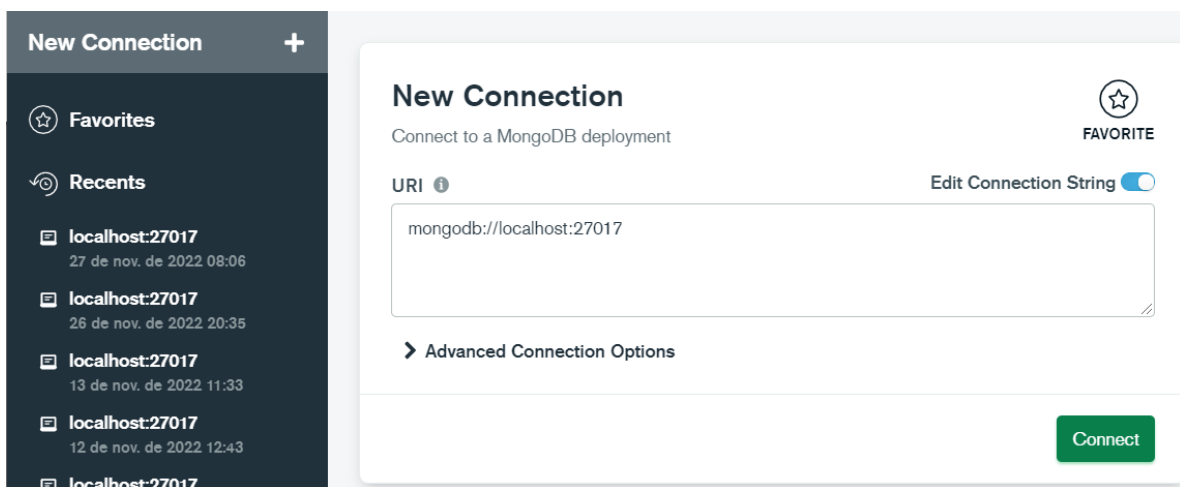
Si se obtiene una respuesta positiva, obtenemos que se hace una petición delete con el ID de la serie para que en el servidor se haga lo necesario para eliminar el registro.

BASE DE DATOS

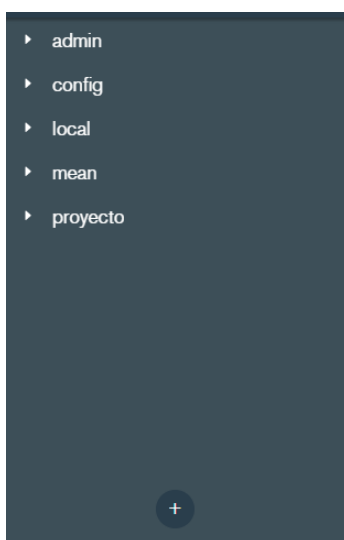
Para la base de datos, se usó MongoDB con su gestor MongoDB Compass, el cual facilita el uso de la misma, otorgándonos una interfaz rápida de usar, muy intuitiva.

IMPLEMENTACIÓN

Entrando primero a MongoDB compass, lo primero que vemos es la pantalla para crear una nueva conexión, esta nos proporciona una URL default para nuestro servidor BD con el puerto 27017. Es importante tener esto en cuenta, pues si recordamos, esta URL se debe colocar en los archivos de configuración del backend y así tener la interacción entre las dos partes.



Solo al dar click en “Connect” nos llevará a nuestra base de datos, en donde de lado izquierdo tenemos un menú con las bases de datos creadas.



Si nosotros queremos agregar una nueva BD es necesario dar click al icono de “+”.

Se nos desplegara un formulario como el siguiente:

Create Database

Database Name

Collection Name

☐ **Capped Collection**
Fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. [Learn More](#)

☐ **Use Custom Collation**
Collation allows users to specify language-specific rules for string comparison, such as rules for lettercase and accent marks. [Learn More](#)

☐ **Time-Series**
Time-series collections efficiently store sequences of measurements over a period of time. [Learn More](#)

i Before MongoDB can save your new database, a collection name must also be specified at the time of creation. [More Information](#)

Cancel Create Database

Ahí nosotros pondremos el nombre de la base de datos y el nombre de una colección, si no queremos agregar aun ninguna colección, no es necesario hacerlo.

Cuando la creamos, tendremos la siguiente pantalla:

Create collectionView

Sort byCollection Name

series

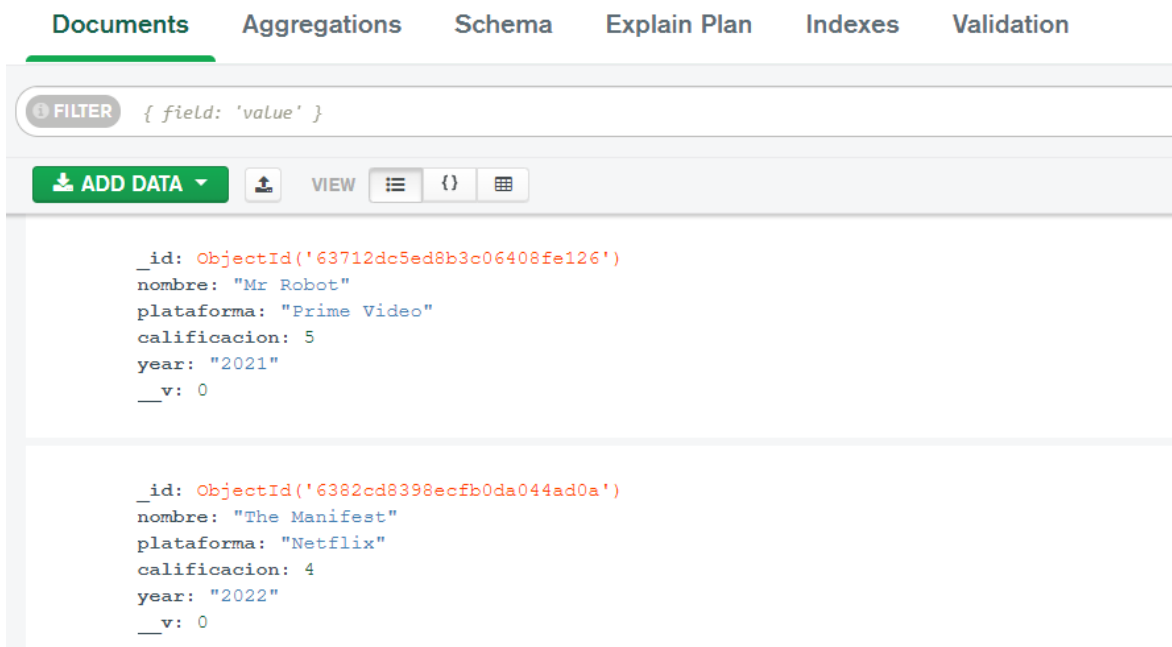
<div>Storage_size:20.48 kB</div>	<div>Documents:2</div>	<div>Avg. document size:113.00 B</div>	<div>Indexes:1</div>	<div>Total index size:36.86 kB</div>
----------------------------------	------------------------	--	----------------------	--------------------------------------

Se nos mostraran las colecciones que tengamos en esa base de datos.

Para ver más detalle de una colección damos click en la que queramos.

Y veremos los documentos en esa colección, los documentos podemos decir que son los registros.

proyecto.series



Documents Aggregations Schema Explain Plan Indexes Validation

FILTER { field: 'value' }

ADD DATA VIEW

```
{
  "_id": ObjectId('63712dc5ed8b3c06408fe126'),
  "nombre": "Mr Robot",
  "plataforma": "Prime Video",
  "calificacion": 5,
  "year": "2021",
  "__v": 0
}
```

```
{
  "_id": ObjectId('6382cd8398ecfb0da044ad0a'),
  "nombre": "The Manifest",
  "plataforma": "Netflix",
  "calificacion": 4,
  "year": "2022",
  "__v": 0
}
```

Con esto, ya tendremos funcionando la base de datos para que se conecte a nuestro backend y nuestro frontend.

CONCLUSIÓN

Realizar este proyecto fue muy beneficioso, puesto que nos permitió experimentar con una nueva forma de hacer sistemas web, conectando tres servidores distintos como lo son el back, el front y la base de datos. El poner a funcionar todo junto fue muy interesante y nos ayuda a ver distintas formas de realizar proyectos.

ENLACES RELEVANTES

<https://angular.io>

<https://material.angular.io/components/table/overview>

<https://material.angular.io/components/paginator/overview>

<https://ui.draculatheme.com>

<https://nodejs.org/en/>

<https://expressjs.com/es/>

<https://www.mongodb.com/try/download/community>

<https://getbootstrap.com>

<https://icons.getbootstrap.com>