Desarrollando un CRUD en ANGULAR

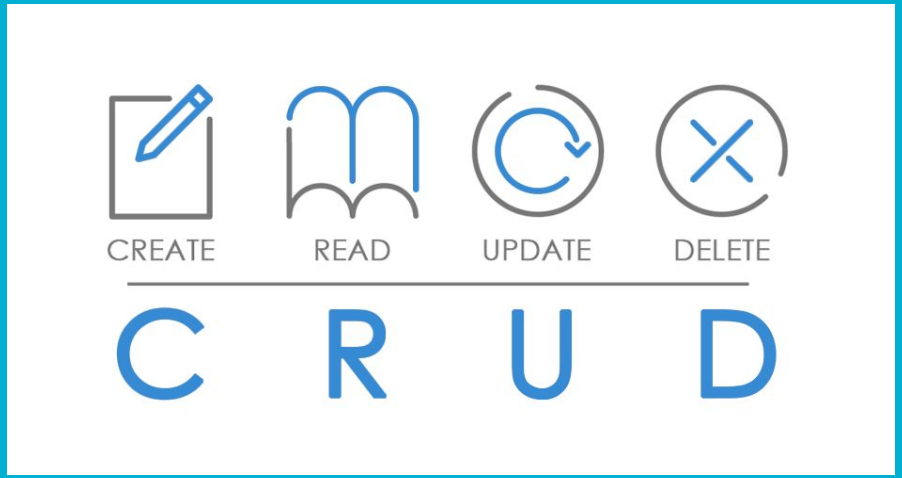
—

Mct. Esteban Calabria

Enunciado

El proyecto tiene como objetivo principal el desarrollo de una aplicación web utilizando Angular que implemente un sistema de gestión de datos (CRUD) de contactos.

En esta parte vamos a implementar la funcionalidad de login



Implementar un sistema de login seguro usando JWT

Frontend: Angular

Backend: Node.js con Express

Autenticación: JSON Web Tokens (JWT)

Incorporar Login en el backend

Paso 1

Configurar el entorno Node.js y Express

Instalar dependencias necesarias (express, jsonwebtoken, cors)

Crear el servidor Express básico

Configurar middleware (cors, express.json)

Definir una clave secreta para JWT

Implementar la ruta de login:

- Recibir credenciales
- Validar credenciales (contra base de datos)
- Generar token JWT si las credenciales son válidas
- Enviar token al cliente

Implementar middleware de autenticación:

- Verificar presencia del token en las solicitudes
- Validar el token
- Adjuntar información del usuario a la solicitud

Proteger rutas que requieren autenticación

JWT (Jason Web Token)

- Es un estándar abierto (RFC 7519) para crear tokens de acceso
- Permite la transmisión segura de información entre partes como un objeto JSON
- Se compone de tres partes: Header, Payload, y Signature
- El Header contiene el tipo de token y el algoritmo de cifrado
- El Payload contiene las declaraciones (claims) sobre la entidad y datos adicionales
- La Signature asegura que el token no ha sido alterado
- JWT es stateless, lo que significa que el servidor no necesita almacenar información de sesión
- Se usa comúnmente para autenticación y autorización en aplicaciones web y APIs
- Proporciona una manera segura de autenticar usuarios sin necesidad de sesiones en el servidor

Status HTTP a utilizar

httpstatus

200 OK: Solicitud exitosa. Se usa para respuestas exitosas de login.

201 Created: Recurso creado exitosamente. Útil al registrar nuevos usuarios.

400 Bad Request: Solicitud inválida. Se usa cuando faltan datos o son incorrectos.

401 Unauthorized: Fallo de autenticación. Se usa cuando las credenciales son inválidas.

403 Forbidden: Acceso denegado. El usuario está autenticado pero no tiene permisos.

404 Not Found: Recurso no encontrado. Útil cuando se busca un usuario que no existe.

500 Internal Server Error: Error en el servidor. Se usa para errores inesperados.

Backend - Configuración Inicial

Librerias

- **express**: Framework web para Node.js
- **jwt**: Librería para manejar JSON Web Tokens
- **cors**: Middleware para habilitar CORS (Cross-Origin Resource Sharing)

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

const jwt = require('jsonwebtoken');

const cors = require('cors');

const app = express();

app.use(cors());

app.use(express.json());

const SECRET_KEY = 'your_secret_key';

// ... (código del servidor)

app.listen(3000, () => console.log('Escuchando puerto 3000'));
```

Backend - Implementación del Login

- if (email === ... && password === ...): Verifica las credenciales (simplificado)
- jwt.sign({ email }, SECRET_KEY, ...): Crea un token JWT
- res.json({ token }): Envía el token al cliente si la autenticación es exitosa
- res.status(401)...: Envía error si la autenticación falla

```
app.post('/login', (req, res) => {

  const { email, password } = req.body;

    // Aquí iría la lógica de verificación con la base de datos

  if (email === "user@example.com" && password === "password") {

    const token = jwt.sign({ email }, SECRET_KEY, { expiresIn: '1h' });

    res.json({ token });

  } else {

    res.status(401).json({ message: 'Autenticación fallida' });

  }

});
```

Incorporar Login en el Frontend

Paso 1

- Implementar el servicio de autenticación (AuthService)
- Crear el componente de login
- Implementar la lógica de manejo de formularios y envío de credenciales
- Crear guards para proteger rutas que requieren autenticación
- Configurar el routing de la aplicación, incluyendo rutas protegidas

Frontend - Servicio de Autenticación

```
@Injectable({  providedIn: 'root')
export class AuthService {
  constructor(private http: HttpClient, private router: Router) {}
  login(email: string, password: string) {
    return this.http.post<any>('http://localhost:3000/login', { email, password })
      .pipe(
        tap(res => {
          localStorage.setItem('token', res.token);
          this.router.navigate(['/']);
        })
      );
  }
```

Frontend - Servicio de Autenticación (Explicacion)

- @Injectable({ providedIn: 'root' }): Decorator que permite inyectar el servicio en toda la aplicación
- constructor(private http: HttpClient, private router: Router): Inyecta dependencias necesarias
- login(email: string, password: string): Método para realizar el login
- http.post<any>('http://localhost:3000/login', { email, password }): Envía credenciales al backend
- .pipe(tap(...)): Permite ejecutar efectos secundarios en el flujo de datos
- localStorage.setItem('token', res.token): Almacena el token JWT en el almacenamiento local
- this.router.navigate(['/']): Redirige al usuario a la página principal tras un login exitoso

Frontend - Servicio de Autenticación

- logout(): Método para cerrar sesión
- localStorage.removeItem('token'): Elimina el token JWT del almacenamiento local
- this.router.navigate(['/login']): Redirige al usuario a la página de login
- isAuthenticated(): Método para verificar si el usuario está autenticado
- const token = localStorage.getItem('token'): Obtiene el token del almacenamiento local
- return !!token: Devuelve true si existe un token, false en caso contrario

```
  logout() {

    localStorage.removeItem('token');

    this.router.navigate(['/login']);

  }


  isAuthenticated(): boolean {

    const token = localStorage.getItem('token');

    return !!token;

  }
}
```

Frontend - Componente de Login

```
@Component({
  selector: 'app-login',
  template: `
    <form (ngSubmit)="onSubmit()">
      <input [(ngModel)]="email" name="email" type="email" required>
      <input [(ngModel)]="password" name="password" type="password" required>
      <button type="submit">Login</button>
    </form>
  `
})
```

Frontend - Componente de Login

```
export class LoginComponent {
  email: string;
  password: string;

  constructor(private authService: AuthService) {}

  onSubmit() {
    this.authService.login(this.email, this.password).subscribe(
      () => console.log('Login exitoso'),
      error => console.error('Error en login', error)
    );
  }
}
```

Frontend - Componente de Login (Explicacion)

- @Component({...}): Decorator que define un componente Angular
- selector: 'app-login': Selector CSS para usar este componente
- template: ...: Template inline del componente
- <form (ngSubmit)="onSubmit()">: Formulario que llama a onSubmit() al ser enviado
- [(ngModel)]="email": Binding bidireccional para el campo de email
- [(ngModel)]="password": Binding bidireccional para el campo de contraseña
- <button type="submit">: Botón para enviar el formulario

Frontend - Componente de Login (Explicacion)

- email: string; password: string;:
  - Propiedades para almacenar las credenciales
- constructor(private authService: AuthService):
  - Inyecta el servicio de autenticación
- onSubmit():
  - Método llamado al enviar el formulario
- this.authService.login(this.email, this.password):
  - Llama al método login del servicio
- .subscribe(...): Se suscribe al observable retornado por login()
  - () => console.log('Login exitoso'): Callback ejecutado en caso de éxito
  - error => console.error('Error en login', error): Callback ejecutado en caso de error

Frontend - Guard de Autenticación

```
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(): boolean {
    if (!this.authService.isAuthenticated()) {
      this.router.navigate(['/login']);
      return false;
    }
    return true;
  }
}
```

Frontend - Guard de Autenticación (Explicacion)

- implements CanActivate: Implementa la interfaz para proteger rutas
- constructor(private authService: AuthService, private router: Router): Inyecta dependencias
- canActivate(): boolean: Método que decide si se permite el acceso a una ruta
- if (!this.authService.isAuthenticated()): Verifica si el usuario no está autenticado
- this.router.navigate(['/login']): Redirige al login si no está autenticado
- return false: Bloquea el acceso a la ruta si no está autenticado
- return true: Permite el acceso a la ruta si está autenticado

Frontend - Configuración de Rutas

```
const routes: Routes = [
  { path: '', component: HomeComponent, canActivate: [AuthGuard] },
  { path: 'login', component: LoginComponent },
  // ... otras rutas
];
```

Frontend - Interceptor HTTP

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = localStorage.getItem('token');
    if (token) {
      const cloned = req.clone({
        headers: req.headers.set('Authorization', `Bearer ${token}`)
      });
      return next.handle(cloned);
    }
    return next.handle(req);
  }
}
```

Frontend - Interceptor HTTP (Explicacion)

- implements HttpInterceptor: Implementa la interfaz para interceptar peticiones HTTP
- intercept(req: HttpRequest<any>, next: HttpHandler): Método para interceptar peticiones
- const token = localStorage.getItem('token'): Obtiene el token JWT del almacenamiento local
- if (token) { ... }: Verifica si existe un token
- req.clone({ ... }): Clona la petición original para modificarla
- headers: req.headers.set('Authorization', Bearer ${token}): Añade el token al header
- return next.handle(cloned): Envía la petición modificada
- return next.handle(req): Envía la petición original si no hay token

Frontend - Interceptor HTTP

```
// En app.module.ts
providers: [
  { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
]
```

Esta configuración asegura que el AuthInterceptor se aplique a todas las peticiones HTTP

El interceptor añadirá automáticamente el token JWT a todas las peticiones salientes

Sigamos Trabajando...