Guards in Angular

In Angular, guards are used to control access to certain routes, providing a way to protect parts of your application and enforce authentication, authorization, or other custom rules. Angular provides several types of guards, including:

1. **CanActivate:** It decides whether a route can be activated. It is typically used to check if a user is authenticated before allowing access to a route.
2. **CanActivateChild:** Similar to `CanActivate`, but specifically for child routes. It is used to check if a user is allowed to access child routes.
3. **CanDeactivate:** It decides whether a route can be deactivated. It is used to confirm with the user before leaving a route (e.g., when a form is dirty).
4. **CanLoad:** It decides if a module can be loaded lazily. It is used to check if a user is authorized to load a feature module.
5. **Resolve:** It performs route data retrieval before activating a route. It allows you to resolve data dependencies before the route is activated.

Example Usage:

Let's go through a simple example of using `CanActivate` to protect a route:

1. **Create a Guard:**

```
ng generate guard auth
```

This command will generate a file named `auth.guard.ts`. Open this file and implement the `CanActivate` interface:

```
// auth.guard.ts
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree }
from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> |
Promise<boolean | UrlTree> | boolean | UrlTree {
    // Your authentication logic goes here
    // If the user is authenticated, return true; otherwise, redirect to the
login page
    const isAuthenticated = true; // Replace with your actual authentication
check
    return isAuthenticated;
  }
}
```

2. **Register the Guard:**

Open your `app-routing.module.ts` file and apply the `CanActivate` guard to the route you want to protect:

```
// app-routing.module.ts
import { AuthGuard } from './auth.guard';

const routes: Routes = [
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [AuthGuard],
  },
  // Other routes...
];
```

In this example, the `/protected` route is protected by the `AuthGuard`, and it will only be accessible if the `canActivate` method in the guard returns `true`.

3. **Use the Guard in a Component:**

You can also use the guard in a component to conditionally display content or redirect the user:

```
// protected.component.ts
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-protected',
  template: `
    <div *ngIf="isAuthenticated; else login">
      <h2>Protected Content</h2>
      <!-- Your protected content goes here -->
    </div>
    <ng-template #login>
      <p>Please login to access this content.</p>
      <button (click)="redirectToLogin()">Login</button>
    </ng-template>
  `,
})
export class ProtectedComponent {
  isAuthenticated = true; // Replace with your actual authentication check

  constructor(private router: Router) {}

  redirectToLogin() {
    this.router.navigate(['/login']);
  }
}
```

This example demonstrates the use of the `CanActivate` guard to protect a route and conditionally display content based on the user's authentication status. You can adapt this pattern for other types of guards based on your application's needs.

Example

Below is an example of a simple login page with username and password using a guard, and the user state is stored in `localStorage`. We'll create an authentication service, a guard, and components for login and protected content.

Step 1: Create an Authentication Service

```
// auth.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class AuthService {
  private readonly storageKey = 'user';

  login(username: string, password: string): boolean {
    // Replace this with your actual authentication logic
    const isAuthenticated = username === 'user' && password === 'password';

    if (isAuthenticated) {
      // Store user state in localStorage
      localStorage.setItem(this.storageKey, JSON.stringify({ username }));
    }

    return isAuthenticated;
  }
  logout(): void {
    // Remove user state from localStorage
    localStorage.removeItem(this.storageKey);
  }

  getUser(): { username: string } | null {
    // Retrieve user state from localStorage
    const user = localStorage.getItem(this.storageKey);
    return user ? JSON.parse(user) : null;
  }

  isAuthenticated(): boolean {
    // Check if user is authenticated based on stored state
    return !!this.getUser();
  }
}
```

Step 2: Create a Guard

```
// auth.guard.ts
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { AuthService } from './auth.service';

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

  canActivate(): boolean {
    if (this.authService.isAuthenticated()) {
      return true;
    } else {
      // Redirect to login page if not authenticated
      this.router.navigate(['/login']);
      return false;
    }
  }
}
```

Step 3: Create Login and Protected Components

```
// login.component.ts
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { AuthService } from './auth.service';
```

```
@Component({
  selector: 'app-login',
  template: `
    <div>
      <h2>Login</h2>
      <label for="username">Username:</label>
      <input type="text" id="username" [(ngModel)]="username" />
      <br />
      <label for="password">Password:</label>
      <input type="password" id="password" [(ngModel)]="password" />
      <br />
      <button (click)="login()">Login</button>
    </div>
  `,
})
export class LoginComponent {
  username = '';
  password = '';

  constructor(private authService: AuthService, private router: Router) {}

  login(): void {
    const isAuthenticated = this.authService.login(this.username,
this.password);

    if (isAuthenticated) {
      this.router.navigate(['/protected']);
    } else {
      // Handle login failure
      alert('Invalid username or password');
    }
  }
}
```

```
// protected.component.ts
import { Component } from '@angular/core';
import { AuthService } from './auth.service';

@Component({
  selector: 'app-protected',
  template: `
    <div *ngIf="user">
      <h2>Protected Content</h2>
      <p>Welcome, {{ user.username }}!</p>
      <button (click)="logout()">Logout</button>
    </div>
    <div *ngIf="!user">
      <p>You are not authenticated. Please login.</p>
    </div>
  `,
})
export class ProtectedComponent {
  user = this.authService.getUser();

  constructor(private authService: AuthService) {}

  logout(): void {
    this.authService.logout();
    location.reload(); // Reload the page to reflect the updated user state
  }
}
```

Step 4: Update App Routing

```
// app-routing.module.ts
import { AuthGuard } from './auth.guard';
import { LoginComponent } from './login.component';
import { ProtectedComponent } from './protected.component';

const routes: Routes = [
  { path: 'login', component: LoginComponent },
  { path: 'protected', component: ProtectedComponent, canActivate: [AuthGuard]
},
  { path: '**', redirectTo: '/login' },
];
```

Step 5: Update AppModule

Ensure you have FormsModule and ReactiveFormsModule imported in your `app.module.ts`:

```
// app.module.ts
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [/* your components */],
  imports: [
    // other imports
    FormsModule,
    ReactiveFormsModule,
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Now, you can navigate to `/login` to enter the username and password, and upon successful login, you'll be redirected to `/protected`. The `AuthGuard` ensures that only authenticated users can access the protected route.

Note: In a real-world application, you would replace the simple username/password check with proper authentication mechanisms, such as token-based authentication. Additionally, handling login failures and user feedback should be improved based on your application's needs.