# Chapter 09: Lazy Loading & Route Guards

## ****What are Child Routes?**** Child routes (or nested routes) allow you to define routes inside a parent route. They are useful for building layouts where certain components (like headers or sidebars) remain constant, while only a part of the view updates.

### What is Lazy Loading?

**Lazy loading** means loading components or routes only when they are needed — not at the start. This reduces the initial bundle size and improves performance.

### ✅ When to Use It

* Large applications with multiple features
* Feature areas like dashboard, admin panel, reports, etc.
* Anything that’s not immediately required on startup

### Setup Lazy Loading in Angular 17+

Angular 17+ uses **loadComponent** or **loadChildren** with **standalone components** — no NgModules required!

Great! Let see how to define and configure **child routes** with **lazy loaded** component:

* Lazy loading studedit component
* Components like studlist, studadd, and studedit
* Dedicated routing file: stud.route.ts
* Integration with main app.routes.ts

**Folder Structure**

src/

app/

stud/

studlist/

studlist.component.ts

studadd/

studadd.component.ts

studedit/

studedit.component.ts

stud.route.ts

app.routes.ts

app.component.ts

**Step 1: Create Standalone Components**

Each component should be created as **standalone**:

ng g component stud/studlist --standalone

ng g component stud/studadd --standalone

ng g component stud/studedit --standalone

**Step 2: Define Routes in stud.route.ts**

**We have to create this file manually in stud folder**

// src/app/stud/stud.route.ts

import { Routes } from '@angular/router';

import { StudlistComponent } from './studlist/studlist.component';

import { StudaddComponent } from './studadd/studadd.component';

export const studRoutes: Routes = [

{ path: '', component: StudlistComponent },

{ path: 'add', component: StudaddComponent },

{ path: 'edit/:id', loadComponent: () =>

import('./studedit/studedit.component').then(m => .StudeditComponent)}

];

* '' is the default child path (/stud)
* edit/:id uses lazy loading with loadComponent, **it will not be imported**

**Step 3: Add Child Routes to app.routes.ts**

// src/app/app.routes.ts

import { Routes } from '@angular/router';

import { studRoutes } from './stud/stud.route';

export const appRoutes: Routes = [

{ path: 'stud', children: studRoutes }

];

**Step 4: Add RouterLinks in StudlistComponent**

<!-- src/app/stud/studlist/studlist.component.html -->

<h2>Student List</h2>

<ul>

<li><a [routerLink]="['/stud/add']">Add Student</a></li>

<li><a [routerLink]="['/stud/edit', 1]">Edit Student #1</a></li>

</ul>

## What is loadChildren?

loadChildren is used in Angular routing to **lazy-load an entire set of routes** (usually for a feature or section like staff, admin, products, etc.).

### Key Benefits:

* Improves **initial load performance**
* Downloads only the routes and components **when needed**

### How it works with Standalone Components

In Angular 14+, loadChildren supports **lazy-loading a route definition (array of Routes)** from a .ts file.

#### ✅ Example

{

path: 'staff',

loadChildren: () =>

import('./staff/staff.routes').then(m => m.staffRoutes)

}

* This tells Angular:
  + When the user navigates to /staff…
  + Load the staff.routes.ts file dynamically.
  + Use the exported staffRoutes array for routing.

## What is a Shared Layout Component?

### Definition:

A **shared layout component** acts as a **visual wrapper** or **parent layout** for a group of child routes.  
It typically contains:

* A **common layout** (e.g., nav, sidebar, header)
* A <router-outlet> to render child components

### Example

// staff.routes.ts

{

path: '',

loadComponent: () =>

import('./layout/layout.component').then(m => m.LayoutComponent),

children: [

{

path: '',

loadComponent: () =>

import('./dashboard/dashboard.component').then(m => m.DashboardComponent)

},

{

path: 'members',

loadComponent: () =>

import('./members/members.component').then(m => m.MembersComponent)

}

]

}

This means:

1. When someone visits /staff or /staff/members, Angular:
   * Loads layout.component.ts first.
   * Inserts the matched child component (dashboard, members, etc.) into **<router-outlet> inside the layout**.

### Layout Component Template Example:

<!-- layout.component.html -->

<h2>Staff Portal</h2>

<nav>

<a routerLink="/staff">Dashboard</a> |

<a routerLink="/staff/members">Members</a>

</nav>

<hr />

<router-outlet></router-outlet>

* The <router-outlet> is **where Angular renders matched child routes**
* The nav, header, etc., are always shown as **shared UI**

**Route Guards**

**What is a Route Guard?**

In Angular, **Route Guards** are interfaces that allow or prevent navigation to or from a route based on logic defined by the developer. They are part of Angular's **RouterModule** and help implement features like authentication, authorization, unsaved form protection, and conditional navigation.

**Types of Route Guards in Angular**

| **Guard Type** | **Interface** | **When It Runs** |
| --- | --- | --- |
| CanActivate | CanActivateFn | Before navigating to a route |
| CanActivateChild | CanActivateChildFn | Before navigating to child routes |
| CanDeactivate | CanDeactivateFn<T> | Before leaving the current route/component |
| CanLoad | CanLoadFn | Before loading lazy-loaded modules |
| CanMatch | CanMatchFn | To check if a route should match the URL segment |

This guide focuses on the first three, which are implemented in your application.

Angular CLI provides a quick and consistent way to scaffold guards.

ng generate guard guards/auth --skip-tests

ng generate guard guards/child --skip-tests

ng generate guard guards/deactivate --skip-tests

**1. CanActivate – Protecting a Route**

**Purpose:** Prevent access to a route unless a condition is met (commonly user authentication).

**Implementation**

**File:** src/app/guards/auth.guard.ts

import { CanActivateFn, Router } from '@angular/router';

import { inject } from '@angular/core';

import { AuthService } from '../auth.service';

export const canActivateGuard: CanActivateFn = () => {

const auth = inject(AuthService);

const router = inject(Router);

if (auth.isLoggedIn()) return true;

alert('Login required');

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

return false;

};

**When to Use**

* Restrict access to private pages (e.g., dashboards, profiles).
* Prevent unauthenticated users from viewing sensitive routes.

**✅ In Routing**

{

path: 'profile',

component: ProfileComponent,

canActivate: [canActivateGuard],

}

**2. CanActivateChild – Guarding Child Routes**

**Purpose:** Similar to CanActivate, but applies to **child routes** under a parent route.

**Implementation**

**File:** src/app/guards/child.guard.ts

import { CanActivateChildFn, Router } from '@angular/router';

import { inject } from '@angular/core';

import { AuthService } from '../auth.service';

export const canActivateChildGuard: CanActivateChildFn = () => {

const auth = inject(AuthService);

const router = inject(Router);

if (auth.isLoggedIn()) return true;

alert('Access denied');

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

return false;

};

**When to Use**

* When a parent route is accessible, but child routes require extra checks.
* Hierarchical access control (e.g., nested modules, admin-only sections).

**✅ In Routing**

{

path: 'profile',

component: ProfileComponent,

canActivate: [canActivateGuard],

canActivateChild: [canActivateChildGuard],

children: [

{ path: 'payment/add', component: AddPaymentComponent },

{ path: 'payment/list', component: ListPaymentComponent },

]

}

**3. CanDeactivate – Confirm Before Leaving**

**Purpose:** Prevent users from leaving a route/component if there are unsaved changes.

**Implementation**

**File:** src/app/guards/deactivate.guard.ts

import { CanDeactivateFn } from '@angular/router';

import { inject } from '@angular/core';

import { FormStateService } from '../form-state.service';

export const canDeactivateGuard: CanDeactivateFn<any> = () => {

const formState = inject(FormStateService);

return !formState.hasUnsavedChanges || confirm('You have unsaved changes. Leave anyway?');

};

**When to Use**

* In forms where unsaved input should be preserved or warned about.
* On components involving user-generated content.

**✅ In Routing**

{

path: 'payment/add',

component: AddPaymentComponent,

canDeactivate: [canDeactivateGuard]

}

**Demonstrating Route Guards in Your App**

**Recommended Folder Structure**

src/app/

│

├── guards/

│ ├── auth.guard.ts (CanActivate)

│ ├── child.guard.ts (CanActivateChild)

│ └── deactivate.guard.ts (CanDeactivate)

│

├── login.component.ts

├── profile.component.ts

├── payment/

│ ├── add-payment.component.ts

│ └── list-payment.component.ts

│

├── auth.service.ts

├── form-state.service.ts (for CanDeactivate)

├── app.routes.ts

├── app.component.ts

└── main.ts

## Code Files

### ✅ AuthService

// src/app/auth.service.ts

import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class AuthService {

private isLoggedInValue = false;

login() { this.isLoggedInValue = true; }

logout() { this.isLoggedInValue = false; }

isLoggedIn() { return this.isLoggedInValue; }

}

### FormStateService (for unsaved form tracking)

// src/app/form-state.service.ts

import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class FormStateService {

hasUnsavedChanges = false;

}

### Guards

#### CanActivate

// src/app/guards/auth.guard.ts

import { CanActivateFn, Router } from '@angular/router';

import { inject } from '@angular/core';

import { AuthService } from '../auth.service';

export const canActivateGuard: CanActivateFn = () => {

const auth = inject(AuthService);

const router = inject(Router);

if (auth.isLoggedIn()) return true;

alert('Login required');

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

return false;

};

#### CanActivateChild

// src/app/guards/child.guard.ts

import { CanActivateChildFn, Router } from '@angular/router';

import { inject } from '@angular/core';

import { AuthService } from '../auth.service';

export const canActivateChildGuard: CanActivateChildFn = () => {

const auth = inject(AuthService);

const router = inject(Router);

if (auth.isLoggedIn()) return true;

alert('Access denied');

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

return false;

};

#### CanDeactivate

// src/app/guards/deactivate.guard.ts

import { CanDeactivateFn } from '@angular/router';

import { inject } from '@angular/core';

import { FormStateService } from '../form-state.service';

export const canDeactivateGuard: CanDeactivateFn<any> = () => {

const formState = inject(FormStateService);

return !formState.hasUnsavedChanges || confirm('You have unsaved changes. Leave anyway?');

};

### LoginComponent

// src/app/login.component.ts

import { Component } from '@angular/core';

import { Router } from '@angular/router';

import { AuthService } from './auth.service';

@Component({

selector: 'app-login',

standalone: true,

template: `

<h2>Login</h2>

<button (click)="login()">Login</button>

`,

})

export class LoginComponent {

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

login() {

this.auth.login();

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

}

}

### ProfileComponent (replaces AdminComponent)

// src/app/profile.component.ts

import { Component } from '@angular/core';

import { RouterModule } from '@angular/router';

@Component({

selector: 'app-profile',

standalone: true,

imports: [RouterModule],

template: `

<h2>My Profile</h2>

<nav>

<a routerLink="payment/add">Add Payment</a> |

<a routerLink="payment/list">List Payments</a>

</nav>

<router-outlet></router-outlet>

`,

})

export class ProfileComponent {}

### AddPaymentComponent (uses CanDeactivate)

// src/app/payment/add-payment.component.ts

import { Component } from '@angular/core';

import { FormStateService } from '../form-state.service';

@Component({

selector: 'app-add-payment',

standalone: true,

template: `

<h3>Add Payment</h3>

<input placeholder="Amount" (input)="onInput()" />

<button (click)="submit()">Submit</button>

`,

})

export class AddPaymentComponent {

constructor(public formState: FormStateService) {}

onInput() {

this.formState.hasUnsavedChanges = true;

}

submit() {

alert('Payment submitted!');

this.formState.hasUnsavedChanges = false;

}

}

### ListPaymentComponent

// src/app/payment/list-payment.component.ts

import { Component } from '@angular/core';

@Component({

selector: 'app-list-payment',

standalone: true,

template: `

<h3>Payment List</h3>

<ul>

<li>Payment #1 - ₹500</li>

<li>Payment #2 - ₹1200</li>

</ul>

`,

})

export class ListPaymentComponent {}

### Routing Setup

// src/app/app.routes.ts

import { Routes } from '@angular/router';

import { LoginComponent } from './login.component';

import { ProfileComponent } from './profile.component';

import { AddPaymentComponent } from './payment/add-payment.component';

import { ListPaymentComponent } from './payment/list-payment.component';

import { canActivateGuard } from './guards/auth.guard';

import { canActivateChildGuard } from './guards/child.guard';

import { canDeactivateGuard } from './guards/deactivate.guard';

export const routes: Routes = [

{ path: '', redirectTo: 'login', pathMatch: 'full' },

{ path: 'login', component: LoginComponent },

{

path: 'profile',

component: ProfileComponent,

canActivate: [canActivateGuard],

canActivateChild: [canActivateChildGuard],

children: [

{ path: 'payment/add', component: AddPaymentComponent, canDeactivate: [canDeactivateGuard] },

{ path: 'payment/list', component: ListPaymentComponent },

]

},

{ path: '\*\*', redirectTo: 'login' },

];

### AppComponent

import { Component } from '@angular/core';

import { RouterModule } from '@angular/router';

@Component({

selector: 'app-root',

standalone: true,

imports: [RouterModule],

template: `

<h1>Payment Portal</h1>

<router-outlet></router-outlet>

`,

})

export class AppComponent {}

## ✅ Summary

You now have a fully working standalone Angular 19 app with:

* **Login system**
* **Protected parent + child routes**
* **Unsaved form detection**
* **Standalone components (no modules!)**
* **Three types of guards**: CanActivate, CanActivateChild, CanDeactivate