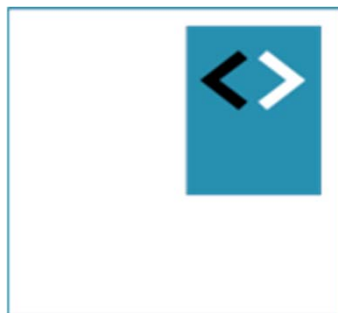




# Gemeente Haarlem

## *Angular - Maatwerk*



Peter Kassenaar  
[info@kassenaar.com](mailto:info@kassenaar.com)



# Routing

Taking your routes to the next level



# Basic Routing

Navigating to different pages in your app



# Official Documentation

- Always check the official documentation
- AI is behind on fast moving techniques!

The screenshot shows the Angular Routing documentation page at [angular.dev/guide/routing](https://angular.dev/guide/routing). The page has a left sidebar with navigation links: Routing, Overview, Common routing tasks, Routing in single-page applications, Creating custom route matches, Router reference, Docs, Tutorials, Playground, and Reference. The main content area is titled "Angular Routing" and includes an introduction, a diagram of routing, and a section "Learn about Angular routing" with links to "Common routing tasks" and "Routing SPA tutorial".

angular.dev/guide/routing

Routing

Overview

Common routing tasks

Routing in single-page applications

Creating custom route matches

Router reference

Docs

Tutorials

Playground

Reference

In-depth Guides > Routing

## Angular Routing

Routing helps you change what the user sees in a single-page app.

In a single-page app, you change what the user sees by showing or hiding portions of the display that correspond to particular components, rather than going out to the server to get a new page.

As users perform application tasks, they need to move between the different views that you have defined.

To handle the navigation from one view to the next, you use the Angular `Router`. The `Router` enables navigation by interpreting a browser URL as an instruction to change the view.

### Learn about Angular routing

#### Common routing tasks

Learn how to implement many of the common tasks associated with Angular routing.


#### Routing SPA tutorial

A tutorial that covers patterns associated with Angular routing.

<https://angular.dev/guide/routing>










# Classical Routing articles (< Angular 17)

Sign in Get started


Dev GeniusWRITE FOR USARCHIVEABOUT US | OFFICIAL STORE

## Advanced Router Configuration In Angular


Aakash Garg Follow 





Oct 19, 2020 · 6 min read




Related

Make a Generic Angular Table. Represent New Data in Easiest Way.

Sharing Data between Angular Components. Data sharing is an essential...

Advanced Angular Structural Directive to Render Long Lists

Setting up Enterprise Angular Applications (AngularInDepth)

Code Image From Unsplash by Markus Spiske

<https://blog.devgenius.io/advanced-router-configuration-in-angular-d22c6dc420be>



ArticlesGuidesBooksWorkshopsMembershipMore 

Search articles...

AccessibilityCSSJavaScriptReactVueRound-UpsUXDesignWeb DesignFigmaWallpapersGuidesBusinessCareer

Ahmed Bouchefra / NOV 28, 2018 / [19 comments](#)

# A Complete Guide To Routing In Angular

 14 min read

 [Apps](#), [PWA](#), [Native](#), [Angular](#), [Service Workers](#)

 Share on [Twitter](#), [LinkedIn](#)

**QUICK SUMMARY** ↔ Throughout this tutorial, Ahmed Bouchefra introduces Angular Router and how you can use it to create client-side apps and Single Page Apps with routing and navigation.

In case you're still not quite familiar with Angular 7, I'd like to bring you closer to everything this impressive front-end framework has to offer. I'll walk you through an Angular demo app that shows different concepts related to the Router, such as:

- The router outlet,



ABOUT THE AUTHOR

Ahmed is a technical author and web developer living in Morocco with a Master's degree in software development. He authors technical content about ... [More about Ahmed](#) ↔

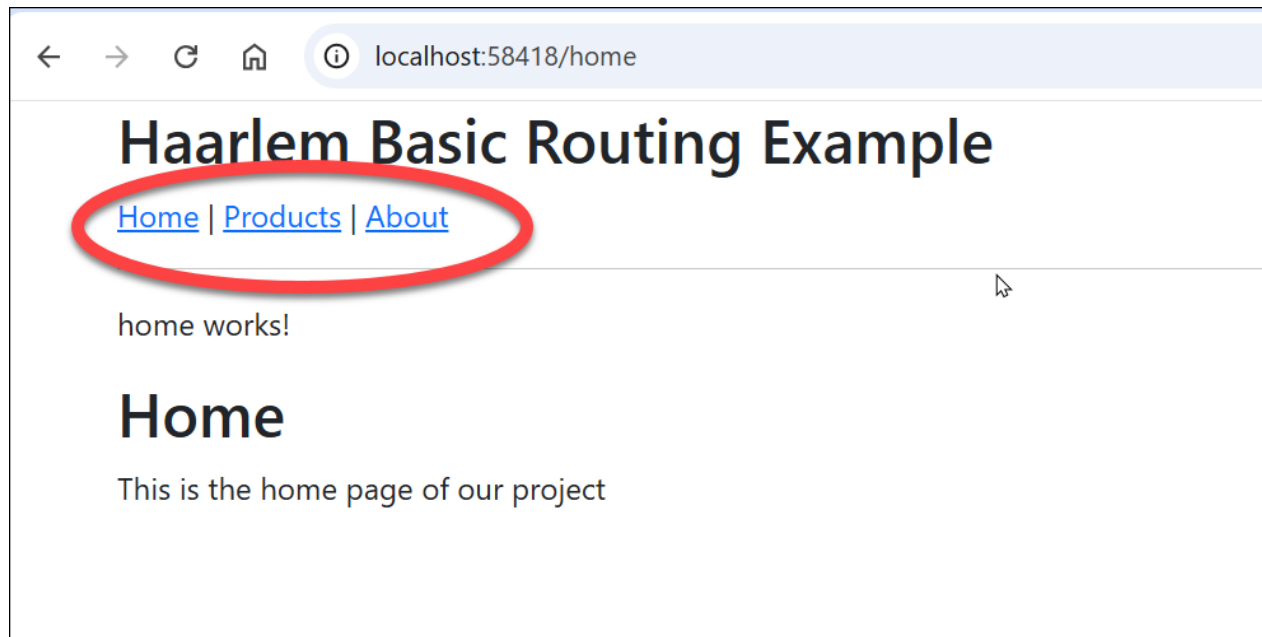
Email Newsletter

<https://www.smashingmagazine.com/2018/11/a-complete-guide-to-routing-in-angular/>



# Setting up basic routing:

- Three steps:
  - 1. Create **Main navigation**, use routerLink attribute!
  - 2. Update **routing table**
  - 3. **Create components** to be routed to.
- Simple example:
  - `../routing-apps/100-basic-routing`





# 1. Create Main navigation

- Can be in its own component of course, or just some `<a routerLink>`-tags
- The `<router-outlet />` tag is where the routes are projected

```
<h2>Haarlem Basic Routing Example</h2>  
  <a routerLink="home">Home</a> |  
  <a routerLink="products">Products</a> |  
  <a routerLink="about">About</a>  
  <hr>  
<router-outlet />
```





## 2. Update/create routing table

- Use lazy loading when possible. Best practice!
- Use `loadComponent()` for that.
  - Previously: `loadChildren()`, which is still available for classic modules.

```
// app.routes.ts
import {Routes} from '@angular/router';

export const routes: Routes = [
  {
    path: 'home',
    loadComponent: () => import('./components/home/home.component')
      .then(m => m.HomeComponent)
  },
  ...
  {
    // No match? Redirect to home page.
    path: '**',
    redirectTo: 'home'
  }
];
```



# More on lazy loading

The screenshot shows the Angular Dev Guide page for lazy loading. The browser address bar displays `angular.dev/guide/routing/common-router-tasks#lazy-loading`. On the left, a sidebar menu under the heading "Routing" lists: "Overview", "Common routing tasks" (highlighted with a purple bar), "Routing in single-page applications", "Creating custom route matches", and "Router reference". The main content area is titled "Lazy loading" and contains the following text: "You can configure your routes to lazy load modules, which means that Angular only loads modules as needed, rather than loading all modules when the application launches. Additionally, preload parts of your application in the background to improve the user experience." Below this, it states: "Any route can lazily load its routed, standalone component by using `loadComponent:`". A code block titled "Lazy loading a standalone component" shows the following TypeScript code:

```
const routes: Routes = [
  {
    path: 'lazy',
    loadComponent: () => import('./lazy.component').then(c => c.LazyComponent)
  }
];
```

<https://angular.dev/guide/routing/common-router-tasks#lazy-loading>

**NOTE: Lazy loaded component MUST be a standalone component!**



# Add routing to app configuration

```
// app.config.ts
import {ApplicationConfig, provideZoneChangeDetection} from '@angular/core';
import {provideRouter} from '@angular/router';

import {routes} from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({eventCoalescing: true}),
    provideRouter(routes) // <== Make sure to add router to config
  ]
};
```



### 3. Create Components

- Create a [standalone] component for every route.
- When navigating via router, the `selector` may actually be omitted.

```
<p>home works!</p>
```

```
<h2>Home</h2>
```

```
<p>This is the home p
```

```
<p>about works!</p>
```

```
<h2>About us</h2>
```

```
<p>
```

```
  Blablaablabalablablablabl
```

```
</p>
```

```
<p>products works!</p>
```

```
<h2>All Products</h2>
```

```
<ul>
```

```
  <li>Product 1</li>
```

```
  <li>Product 2</li>
```

```
  <li>Product 3</li>
```

```
  <li>Product 4</li>
```

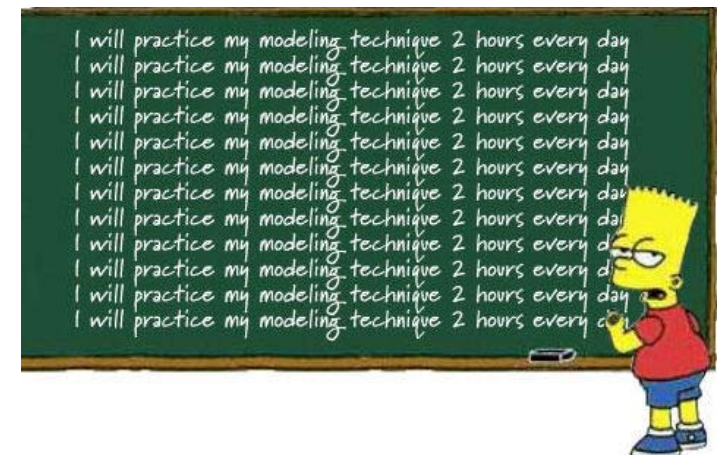
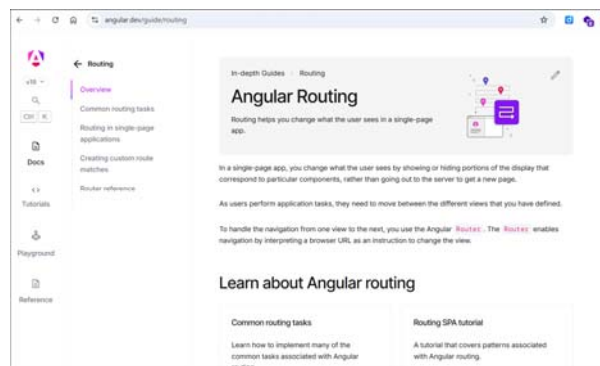
```
  <li>Product 5</li>
```

```
</ul>
```



# Workshop

- Create a small application, using basic routing with lazy loading, as described in the previous slides
- Need inspiration? See example [../100-basic-routing](#)
- Read the documentation on Routing on angular.dev





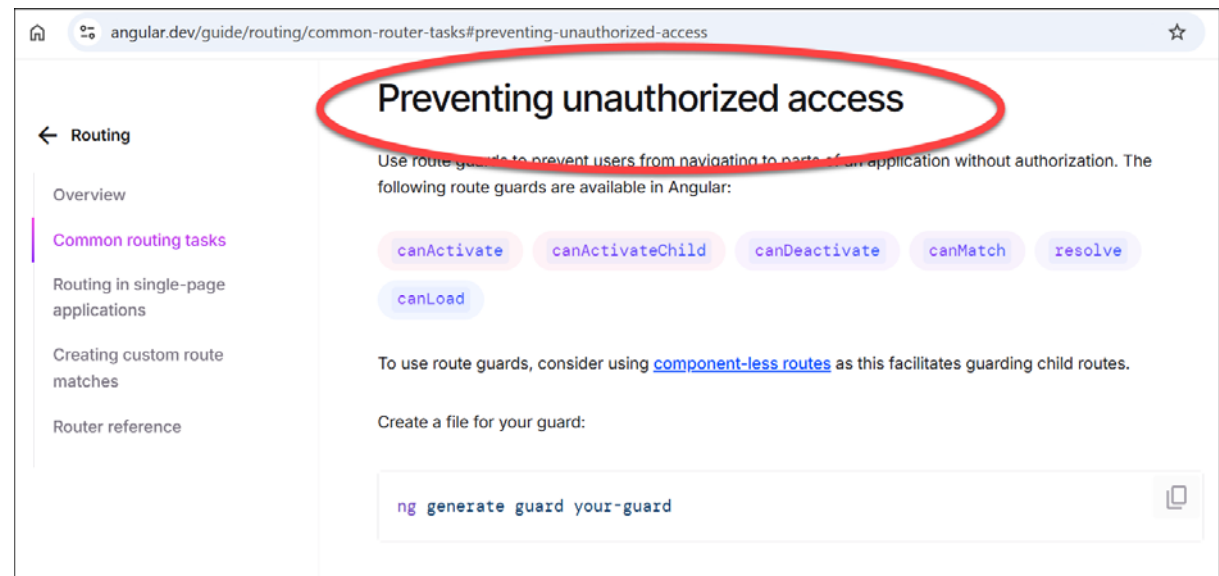
# Routing Guards

Preventing access to parts of your site



# Preventing unauthorized access

- Use **Route Guards** to prevent users from navigating to parts of an application without authorization.
- The following route guards are **available** in Angular:
  - `canActivate`
  - `canActivateChild`
  - `canDeactivate`
  - `canMatch`
  - `resolve`
  - `canLoad`





# 1. Generate a guard file

- Generate a guard file with a custom name
- For instance: `ng generate guard authGuard`
- Add the guard function that you want to use (here: `canActivateFn`)

```
ng g guard authGuard
```

```
? Which type of guard would you like to create?
```

```
> ☒ CanActivate
```

```
☐ CanActivateChild
```

```
☐ CanDeactivate
```

```
☐ CanMatch
```







# 1. Add logic to guard file

- Implement guarding logic to return `true` | `false`

```
// auth.guard.ts
import {ActivatedRouteSnapshot,
        CanActivateFn, RouterStateSnapshot} from '@angular/router';
import {inject} from '@angular/core';
import {AuthService} from '../services/auth.service';

export const authGuard: CanActivateFn = (
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
) => {

  // Your Logic goes here. For now:
  const authService = inject(AuthService)
  return authService.isLoggedIn();
};
```



## 2. Tell the route to use the guard

- **Update** the route configuration to use the guard
- There can be **multiple guards** active on a route
  - ALL guards must return true in order for the guard to succeed!
- For instance:

```
import {authGuard} from './guards/auth.guard';
```

```
{  
  path: 'products',  
  loadComponent: () => import('./components/products/products.component')  
    .then(m => m.ProductsComponent),  
  canActivate: [authGuard],  
},
```

Should the `authGuard` return `false`, the Product component can NOT be loaded



### 3. Redirect on false

- If you want to redirect to (for instance) a Login page if the user is NOT logged in, the Guard must return a `UrlTree`:

```
export const authGuard: CanActivateFn = (route, state) => {  
  
  const authService = inject(AuthService)  
  const router = inject(Router);  
  
  if (authService.isLoggedIn()) {  
    return true;  
  }  
  
  // Redirect to login page and store the attempted URL as a queryParams.  
  return router.createUrlTree(['/login'], {  
    queryParams: { returnUrl: state.url }  
  });  
  
};
```



# When guard returns false:

← → ↻ 🏠 ⓘ localhost:58418/login?returnUrl=%2Fproducts

## Haarlem Basic Routing Example

[Home](#) | [Products](#) | [About](#)

---

login works!

### Login component

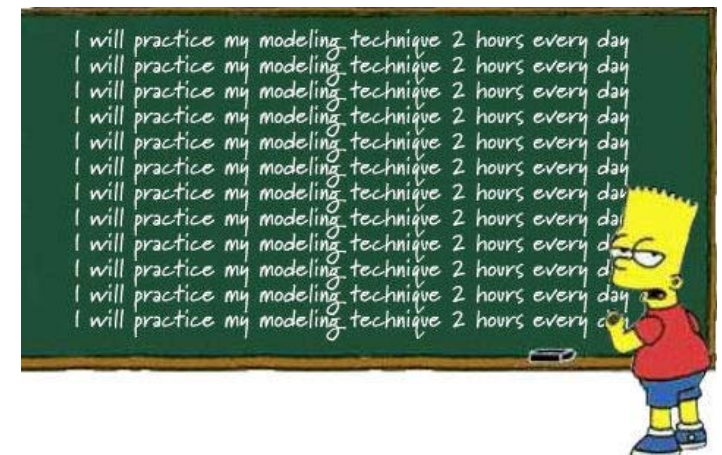
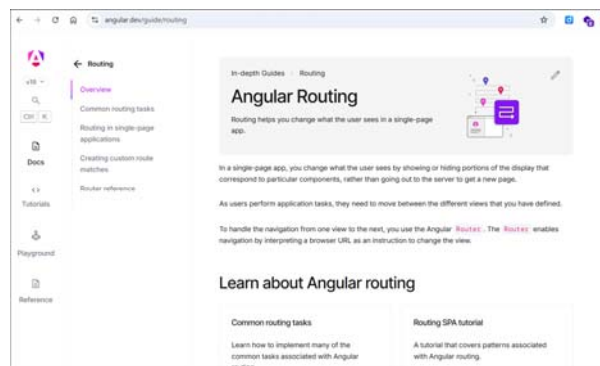
This is the `Login` component. It should be loaded if the `authService` returns `false` for the logged in state!

Write your own logic to handle username/password and logins.



# Workshop

- Continue from the previous application
  - Add a guard to one of your routes. Access to a component should ONLY be granted if the guard returns true
  - Create a redirect/login page if the user is not logged in
- Need inspiration? See example [../100-basic-routing](#)
- Read the documentation on Routing on angular.dev





# Deactivate Guard

We want to prevent users from accidentally navigating away if they have unsaved changes.



# 1. canActivate()

- Prevent users from (accidentally) navigating away.
- Multiple steps involved.
- 1. Create canActivate Guard:

```
// deactivate.guard.ts  
import { CanDeactivateFn } from '@angular/router';  
  
// Define an interface for components that can be deactivated  
export interface CanDeactivateComponent {  
  canActivate: () => boolean | Promise<boolean>;  
}  
  
export const deactivateGuard: CanDeactivateFn<CanDeactivateComponent> = (  
  component) => {  
    return component.canDeactivate();  
  };  
};
```



## 2. Component that implements interface

- Create a component that implements the `CanDeactivateComponent` interface, created in the previous step

```
<h3>Edit Profile</h3>
<form #form="ngForm" (ngSubmit)="onSubmit(form)">
  ...
</form>
```

```
export class ProfileEditComponent implements CanDeactivateComponent {

  isDirty = false;

  canDeactivate(): boolean {
    if (this.isDirty) {
      return window.confirm('You have unsaved changes (...)?');
    }
    return true;
  }
  ...
}
```





### 3. Update routing table

- Create a route for the `profileEdit` component and add the guard.

```
export const routes: Routes = [  
  // ...  
  // A route that prevents users from navigating away.  
  {  
    path: 'profile/edit',  
    loadComponent: () => import('...../profile-edit.component')  
      .then(m => m.ProfileEditComponent),  
    canDeactivate: [deactivateGuard]  
  },  
  ...  
];
```



## 4. Update main navigation

- Create a new entry in the main navigation to point to the edit-route.

This is simple:

```
<div class="container">
  <h2>Haarlem Basic Routing Example</h2>
  ...
  <a routerLink="profile/edit">Edit Profile</a> |
  ...
  <router-outlet/>
</div>
```

# Result



localhost:58418/profile/edit

## Haarlem Basic Routing Example

[Home](#) | [Products](#) | [Edit Profile](#) | [About](#)

### Edit Profile

This form uses the `canDeactivate` guard. If you want to navigate away, while there are unsaved changes, you will be asked to confirm.

Name:

Email:

localhost:58418 says  
You have unsaved changes. Do you really want to leave?

../120-canDeactivate-guard



# Summary

- Now, if you navigate away from the form if there are unsaved changes, you'll have to confirm.
- This pattern is useful for:
  - Forms with unsaved changes
  - File editors
  - Components with ongoing operations that shouldn't be interrupted
- You can make the guard more sophisticated by:
  - Using a custom dialog instead of the browser's confirm
  - Adding async operations (return a Promise )
  - Saving draft data before allowing navigation
  - Checking specific conditions beyond just form state