The browser is a familiar model of application navigation:

* Enter a URL in the address bar and the browser navigates to a corresponding page.
* Click links on the page and the browser navigates to a new page.
* Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

# The Basics

## <base href>

Most routing applications should add a <base> element to the index.html as the first child in the <head> tag to tell the router how to compose navigation URLs.

If the app folder is the application root, as it is for the sample application, set the href value exactly as shown here.

<base href="/">

## Importing the Module

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core. It is in its own library package, @angular/router. Import what you need from it as you would from any other Angular package.

## Configuration

A routed Angular application has one singleton instance of the Router service. When the browser's URL changes, that router looks for a corresponding Route from which it can determine the component to display.

A router has no routes until you configure it.

The routing can be configured using **RouterModule.forRoot()** method.

### Properties:

path: “” this will be the url which will be entered in browser.

component: name of the component this is not a string property, make sure to import the component.

redirectTo:”” optional property in case you want to redirect to some existing path.

pathMatch:”” used with redirectTo property

# Router Outlet

Given this configuration, when the browser URL for this application becomes /path, the router matches that URL to the route path /path and displays the Component after a RouterOutlet that you've placed in the host view's HTML.

<router-outlet></router-outlet>

# Router Links

Now you have routes configured and a place to render them, but how do you navigate? The URL could arrive directly from the browser address bar. But most of the time you navigate as a result of some user action such as the click of an anchor tag.

<a routerLink="/path" routerLinkActive="active">Link</a>

The **[RouterLink](https://angular.io/api/router/RouterLink)** directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the routerLink (a "one-time" binding).

Had the navigation path been more dynamic, you could have bound to a template expression that returned an array of route link parameters (the *link parameters array*). The router resolves that array into a complete URL.

The **[RouterLinkActive](https://angular.io/api/router/RouterLinkActive)** directive on each anchor tag helps visually distinguish the anchor for the currently selected "active" route. The router adds the active CSS class to the element when the associated ***RouterLink*** becomes active. You can add this directive to the anchor or to its parent element.

# Router state

After the end of each successful navigation lifecycle, the router builds a tree of **[ActivatedRoute](https://angular.io/api/router/ActivatedRoute)** objects that make up the current state of the router. You can access the current **[RouterState](https://angular.io/api/router/RouterState)** from anywhere in the application using the [**Router**](https://angular.io/api/router/Router) service and the **routerState** property.

Each **[ActivatedRoute](https://angular.io/api/router/ActivatedRoute)** in the **[RouterState](https://angular.io/api/router/RouterState)** provides methods to traverse up and down the route tree to get information from parent, child and sibling routes.

# Router Event;

During each navigation, the [Router](https://angular.io/api/router/Router) emits navigation events through the Router.events property. These events range from when the navigation starts and ends to many points in between. The full list of navigation events is displayed in the table below.

|  |  |
| --- | --- |
| Router Event | Description |
| NavigationStart | An event triggered when navigation starts. |
| RoutesRecognized | An event triggered when the Router parses the URL and the routes are recognized. |
| RouteConfigLoadStart | An event triggered before the Router lazy loads a route configuration. |
| RouteConfigLoadEnd | An event triggered after a route has been lazy loaded. |
| NavigationEnd | An event triggered when navigation ends successfully. |
| NavigationCancel | An event triggered when navigation is canceled. This is due to a Route Guard returning false during navigation. |
| NavigationError | An event triggered when navigation fails due to an unexpected error. |

These events are logged to the console when the **enableTracing** option is enabled also. Since the events are provided as an Observable, you can filter() for events of interest and subscribe() to them to make decisions based on the sequence of events in the navigation process.

# ActivatedRoute: the one-stop-shop for route information

The route path and parameters are available through an injected router service called the ActivatedRoute. It has a great deal of useful information including:

|  |  |
| --- | --- |
| Property | Description |
| url | An Observable of the route path(s), represented as an array of strings for each part of the route path. |
| data | An Observable that contains the data object provided for the route. Also contains any resolved values from the [resolve guard](https://angular.io/guide/router#resolve-guard). |
| paramMap | An Observable that contains a [map](https://angular.io/api/router/ParamMap) of the required and [optional parameters](https://angular.io/guide/router#optional-route-parameters) specific to the route. The map supports retrieving single and multiple values from the same parameter. |
| queryParamMap | An Observable that contains a [map](https://angular.io/api/router/ParamMap) of the [query parameters](https://angular.io/guide/router#query-parameters) available to all routes. The map supports retrieving single and multiple values from the query parameter. |
| fragment | An Observable of the URL [fragment](https://angular.io/guide/router#fragment) available to all routes. |
| outlet | The name of the [RouterOutlet](https://angular.io/api/router/RouterOutlet) used to render the route. For an unnamed outlet, the outlet name is *primary*. |
| routeConfig | The route configuration used for the route that contains the origin path. |
| parent | The route's parent [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) when this route is a [child route](https://angular.io/guide/router#child-routing-component). |
| firstChild | Contains the first [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) in the list of this route's child routes. |
| children | Contains all the [child routes](https://angular.io/guide/router#child-routing-component) activated under the current route. |

Two older properties are still available. They are less capable than their replacements, discouraged, and may be deprecated in a future Angular version.

1. params — An Observable that contains the required and optional parameters specific to the route. Use paramMap instead.
2. queryParams — An Observable that contains the query parameters available to all routes. Use queryParamMap instead.

# *ParamMap* API

The ParamMap API is inspired by the URLSearchParams interface. It provides methods to handle parameter access for both route parameters (paramMap) and query parameters (queryParamMap).

this.route.paramMap .switchMap((params: ParamMap) => this.service.getHero(params.get('id'))) .subscribe((hero: Hero) => this.hero = hero);

|  |  |
| --- | --- |
| **Member** | **Description** |
| has(name) | Returns true if the parameter name is in the map of parameters. |
| get(name) | Returns the parameter name value (a string) if present, or null if the parameter name is not in the map. Returns the *first* element if the parameter value is actually an array of values. |
| getAll(name) | Returns a string array of the parameter name value if found, or an empty array if the parameter name value is not in the map. Use getAll when a single parameter could have multiple values. |
| keys | Returns a string array of all parameter names in the map. |

|  |  |
| --- | --- |
| **Member** | **Description** |
| has(name) | Returns true if the parameter name is in the map of parameters. |
| get(name) | Returns the parameter name value (a string) if present, or null if the parameter name is not in the map. Returns the *first* element if the parameter value is actually an array of values. |
| getAll(name) | Returns a string array of the parameter name value if found, or an empty array if the parameter name value is not in the map. Use getAll when a single parameter could have multiple values. |
| keys | Returns a string array of all parameter names in the map. |

# Optional Params:

this.router.navigate(['/route, { id: id, foo: 'foo' }]);

It should look something like this, depending on where you run it:

localhost:4200/route;id=15;foo=foo

The optional route parameters are not separated by "?" and "&" as they would be in the URL query string. They are separated by semicolons ";" This is matrix URL notation — something you may not have seen before.

# Route guards

At the moment, any user can navigate anywhere in the application anytime. That's not always the right thing to do.

1. Perhaps the user is not authorized to navigate to the target component.
2. Maybe the user must login (authenticate) first.
3. Maybe you should fetch some data before you display the target component.
4. You might want to save pending changes before leaving a component.
5. You might ask the user if it's OK to discard pending changes rather than save them.

You can add guards to the route configuration to handle these scenarios.

A guard's return value controls the router's behavior:

* If it returns true, the navigation process continues.
* If it returns false, the navigation process stops and the user stays put.

The guard can also tell the router to navigate elsewhere, effectively canceling the current navigation.

A routing guard can return an **Observable<boolean>** or a **Promise<boolean>** and the router will wait for the observable to resolve to true or false.

The router supports multiple guard interfaces:

* CanActivate to mediate navigation to a route.
* CanActivateChild to mediate navigation to a child route.
* CanDeactivate to mediate navigation away from the current route.
* Resolve to perform route data retrieval before route activation.
* CanLoad to mediate navigation to a feature module loaded asynchronously.

# *CanActivate*: requiring authentication

Applications often restrict access to a feature area based on who the user is. You could permit access only to authenticated users or to users with a specific role. You might block or limit access until the user's account is activated.

The CanActivate guard is the tool to manage these navigation business rules.

# CanActivateChild*: guarding child routes*

You can also protect child routes with the CanActivateChild guard. The CanActivateChild guard is similar to the CanActivate guard. The key difference is that it runs before any child route is activated.

# *CanDeactivate*: handling unsaved changes

In the real world, you might have to accumulate the users changes. You might have to validate across fields. You might have to validate on the server. You might have to hold changes in a pending state until the user confirms them *as a group* or cancels and reverts all changes.

It's better to pause and let the user decide what to do. If the user cancels, you'll stay put and allow more changes. If the user approves, the app can save.

You still might delay navigation until the save succeeds. If you let the user move to the next screen immediately and the save were to fail (perhaps the data are ruled invalid), you would lose the context of the error.

You can't block while waiting for the server—that's not possible in a browser. You need to stop the navigation while you wait, asynchronously, for the server to return with its answer.

You need the CanDeactivate guard.

# Resolve*: pre-fetching component data*

It's preferable to pre-fetch data from the server so it's ready the moment the route is activated. This also allows you to handle errors before routing to the component.

In summary, you want to delay rendering the routed component until all necessary data have been fetched.

You need a *resolver*.

# Routing Module

The Routing Module has several characteristics:

1. Separates routing concerns from other application concerns.
2. Provides a module to replace or remove when testing the application.
3. Provides a well-known location for routing service providers including guards and resolvers.
4. Does not declare components.

# Shared Module:

Add shared modules to hold common modules /components/directives and pipes.

# Feature Module:

There is a problem with adding everything into AppModule, The root AppModule grows larger with each new application class.

The above issue can be resolved with using **Feature Modules.**

A feature module is a class adorned by the **@NgModule** decorator and its metadata, just like a root module. Feature module metadata have the same properties as the metadata for a root module.

The root module and the feature module share the same execution context. They share the same dependency injector, which means the services in one module are available to all.

The modules have the following significant technical differences:

1. You boot the root module to launch the app; you import a feature module to extend the app.
2. A feature module can expose or hide its implementation from other modules.

A feature module delivers a cohesive set of functionality focused on an application business domain, user workflow, facility (forms, http, routing), or collection of related utilities.

While you can do everything within the root module, feature modules help you partition the app into areas of specific interest and purpose.

A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.