**The Angular Router Module**

The Router is a separate module in Angular. It is in its own library package, *@angular/router*. The Router Module provides the necessary service providers and directives for navigating through application views.

Using Angular Router you can

* Navigate to a specific view by typing a URL in the address bar
* Pass optional parameters to the View
* Bind the clickable elements to the View and load the view when the user performs application tasks
* Handles back and forward buttons of the browser
* Allows you to dynamically load the view
* Protect the routes from unauthorized users using Guards

## Components of Angular Router Module

### Router

### Router

The Angular Router is an object that enables navigation from one component to the next component as users perform application tasks like clicking on menus links, buttons or clicking on back/forward button on the browser. We can access the router object and use its methods like navigate() or navigateByUrl(), to navigate to a route

### Route

Route tells the Angular Router which view to display when a user clicks a link or pastes a URL into the browser address bar. Every Route consists of a path and a component it is mapped to. The Router object parses and builds the final URL using the Route

### Routes

Routes is an array of Route objects our application supports

RouterOutlet

The outerOutlet is a directive (<router-outlet>) that serves as a placeholder, where the Router should display the view

RouterLink

The RouterLink is a directive that binds the HTML element to a Route. Clicking on the HTML element, which is bound to a RouterLink, will result in navigation to the Route. The RouterLink may contain parameters to be passed to the route’s component.

RouterLinkActive

RouterLinkActive is a directive for adding or removing classes from an HTML element that is bound to a RouterLink. Using this directive, we can toggle CSS classes for active RouterLinks based on the current RouterState

ActivatedRoute

The ActivatedRoute is an object that represents the currently activated route associated with the loaded Component.

RouterState

The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.

RouteLink Parameters array

The Parameters or arguments to the Route. It is an array which you can bind to RouterLink directive or pass it as an argument to the Router.navigate method.

How to configure Angular Router

To Configure the Router in Angular, you need to follow these steps

Set the <base href>

Define routes for the view

Register the Router Service with Routes

Map HTML Element actions to Route

Choose where you want to display the view

Set the <base href>

The HTML <base> element specifies the base URL to use for all relative URLs contained within a document.

The Angular Router uses the HTML5 style of Routing (or PathLocationStrategy) as the default option. The router makes use of the browser’s history API for navigation and URL interaction.



To make HTML5 routing to work, we need to set up the “**base href”** in the DOM. This is done in app’s index.html file immediately after the head tag.

### Define the routes

Next, create an array of route objects. Each route maps path (URL Segment) to the component



Where

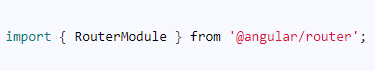
**path:** The URL path segment of the route. We will use this value to refer to this route elsewhere in the app

**component:** The component to be loaded.

This route tells angular to render ProductComponent when the user navigate to the URL “/product”

**Register the Routes**

Import the Router Module from @angular/router library in the root module of the application



Then, install the routes using the RouterModule.forRoot method, passing the routes as the argument in the imports array



#### Map Action to Routes

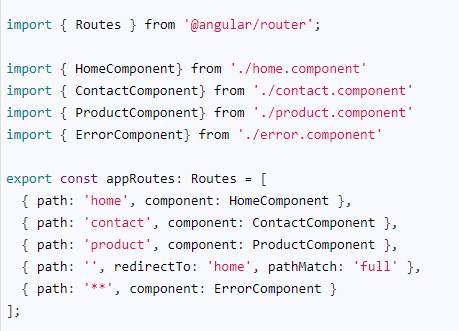


**Choose where you want to display**

Finally, we need to tell the angular where to display the view. This is done using the RouterOutlet directive as shown. We will add the following directive to the root component.



Note the **<base href=”/”>** right after the head tag. This makes the browser know where is the root of our application is and helps it to construct the URL’s



#### Default Route

The fourth route is



The path is empty, indicates the default route. The default route is redirected to the home path using the RedirectTo argument. This route means that, when you navigate to the root of your application /, you are redirected to the home path (/home), which in turn displays the HomeComponent.

Note, that we have pathMatch argument set to ‘full’. The pathMatch tells the Router how to match the URL.

When it is set to full, the path is matched to the entire URL

Every route ends in an empty space for ex: /contact/’’. If pathMatch is not set to full then the router will apply the redirect, which results in the error.

#### Wild Card Route

|  |
| --- |
| The next route is wildcard route |



The “\*\*” matches every URL. The Router will display the ErrorComponent.

#### Order matters: First one wins

Note that the order of the route is important. The Routes are matched in the order they are defined. The Router always returns the first matching route (first-match-wins strategy)

Since the wildcard route (\*\*) matches every URL and should be placed last.

Now, we have set up our routes. Now we will add these routes to our application.



Note that we are using the **forRoot method**.

**the forRoot method** is used, when you want to provide the service and also want to configure the service at the same time

The **routermodule.forroot** method returns the Router Service configured with the routes passed in the argument and also registers the Router service. It also registers the other providers that the routing module requires.

When the application is bootstrapped, the Router service looks at the current browser URL and performs the initial navigation.

When the user changes the URL either clicking on a link in the page or by entering a URL in the address bar, the router looks for a corresponding Route from the Routes array and renders the associated component.

## Location Strategies in Angular Router

Being a Single Page Application, the Angular applications should not send the URL to the server and should not reload the page, every time user requests for a new page.

The URLs are strictly local in Angular Apps. The [Angular router](https://www.tektutorialshub.com/angular/angular-routing-navigation/) navigates to the new component and renders its template and updates the history and URL for the view. All this happens locally in the browser.

There are two ways, by which Angular achieves this. These are called Location Strategies.

The Location Strategy defines how our URL/Request is resolved. It also determines how your URL will look like

Angular supports two Location Strategies:

1. **HashLocationStrategy**  
   Where URL looks like http://localhost:4200/#/product
2. **PathLocationStrategy**  
   Where URL looks like http://localhost:4200/product

Before going further lets first understand what is client-side routing is

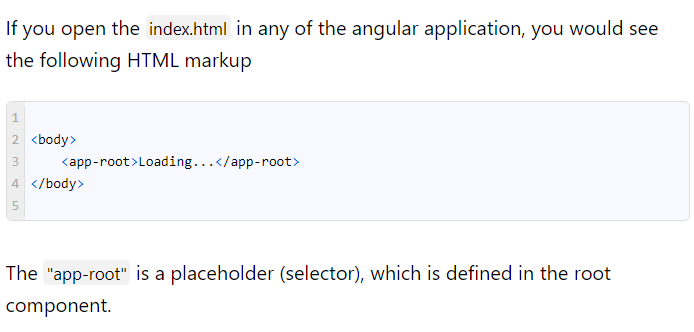
### Client-Side Routing

In a Multi-page web application, Every time the application needs to a display a page it has to send a request to the webserver. You can do that by either typing the URL in the address bar, clicking on the Menu link/ button. Every such action results in browser sending a new request to the Web server

But, the Angular Applications are single-page applications or SPA.

All the components are displayed on a single page

In a Typical Single Page Application, when the Web application is loaded it loads the single HTML page. Whenever the user interacts with the page, only a part of the page is dynamically updated.



Angular generates and loads the view associated with the root component inside the "app-root". Any subsequent components are also loaded dynamically inside the "app-root" selector

Angular does all this behind the scenes.

In such a scenario, we are not required to change the URL. But that brings a few cons

* You won’t be able to refresh the page
* You won’t be able to go to a particular view by typing the URL
* Sharing the URL with someone is not possible
* The Back button will not work as you cannot go back to the previous page
* SEO is not possible

That is where the client-side routing comes into the picture

The Client-side routing simply mimics server-side routing by running the process in the browser. It changes the URL in the browser address bar and updates the browser history, without actually sending the request to the server

### How Client-Side Routing works

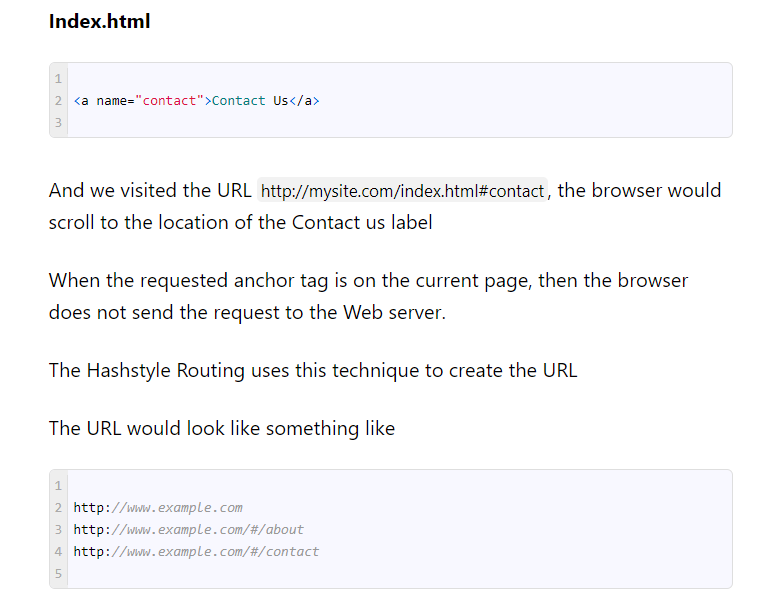
The Client-side routing is handled in two ways

1. Hash style Routing
2. HTML 5 Routing

## Hash Style Routing

The Hash style routing using the anchor tags technique to achieve client-side routing.

The anchor tags, when used along with the # allows us to jump to a place, within the web page.



In all the above examples, only the URL sent to the server is http://www.example.com the URL’s "#/about" and #/contact is never sent to the server

## HTML 5 routing

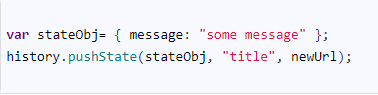
The introduction of HTML5, now allows browsers to programmatically alter the browser’s history through the history object.

Using [history.pushState()](https://developer.mozilla.org/en-US/docs/Web/API/History_API) method, we can now programmatically add the browser history entries and change the location without triggering a server page request.

The [history.pushState](https://www.sitepoint.com/javascript-history-pushstate/) method accepts the following three parameters.

1. **State object:** A state object is a JavaScript object which is associated with the new history entry created by pushState()
2. **Title:** This is an optional title for the state
3. **URL:** The new history entry’s URL. The browser won’t jump to that page.

For example:



Using history.pushState the method, The browser creates new history entries that change the displayed URL without the need for a new request.

Example

When you request for http://www.example.com the server sends the index.html

Now, When you click on ProductList link, Angular use’s the history.pushState method to push the state and change the URL to http://www.example.com/ProductList

Now, when you click on the specific Product, we again the use history method to push the state and change the URL to http://www.example.com/product/1

Here, when you click the back button, the browser will retrieve the http://www.example.com/ProductList from history and displays it.

But there are cons to this approach

1. Not all browsers support HTML 5
2. The older browser does not support HTML5. So if you want to support older browser, you have to stick to the hash style routing
3. The server support is needed for HTML5 based routing.

#### Why Server Support Needed for HTML 5 routing

Now, consider the above example

What would happen, when you type the URL http://www.example.com/ProductList and hit the refresh button.

The browser will send the request to the webserver. Since the page ProductList does not exist, it will return the 404 (page not found) error.

This problem could be solved, if we are able to redirect all the request to the index.html

It means that when you ask from http://www.example.com/ProductList, the Web server must redirect it to index.html and return the request. Then in the Front-end Angular will read the URL and dynamically load the ProductListComponent.

To make HTML5 routing work you need to send the instruction to the webserver to serve /index.html for any incoming request, no matter what the path is.

## Location Strategy

As mentioned earlier, Angular implements both Hashstyle & HTML 5 Routing. HashLocationstrategy implements the Hashstyle routing & Pathlocationstrategy implements the HTML5 style routing

## PathLocationStrategy Vs HashLocationStrategy

### PathLocationStrategy

**Pros:**

* Produces a clear URL like http://example.com/foo
* Supports Server-Side Rendering

Server-side Rendering is a technique that renders critical pages on the server that can greatly improve perceived responsiveness when the app first loads

Cons:

* Older browser does not support
* Server Support needed for this to work

### HashLocationStrategy

**Pros:**

* Supported by all browsers

**Cons:**

* Produces a URL like http://example.com/#foo
* Will not Support Server-Side Rendering

## PathLocationStrategy

The PathLocationStrategy is the default strategy in Angular application.

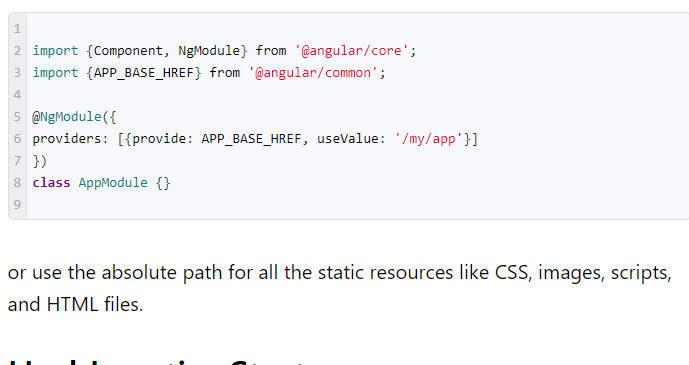
To Configure the strategy, we need to add <base href> in the <head> section of root page (index.html) of our application

|  |  |
| --- | --- |
| 1  2  3 | <base href="/"> |

The Browser uses this element to construct the relative URLs for static resources (images, CSS, scripts) contained in the document.

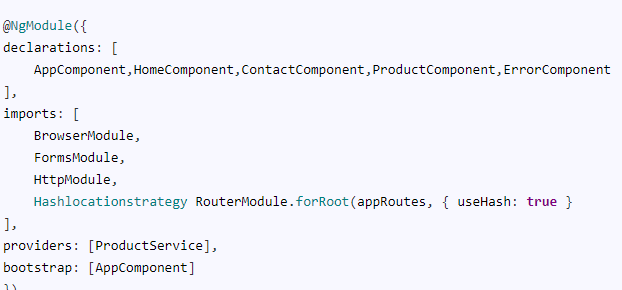
If you do not have access to <head> Section of the index.html, then you can follow either of the two steps

Add the APP\_BASE\_HREF value as shown in the[provider’s](https://www.tektutorialshub.com/angular/angular-providers/) section of the root module



## HashLocationStrategy

You can use the HashLocationStrategy by providing the useHash: true in an object as the second argument of the RouterModule.forRoot in the AppModule.



## Which Location Strategy to Use

We recommend you to use the HTML 5 style (PathLocationStrategy ) as your location strategy.

Because

* It produces clean and SEO Friendly URLs that are easier for users to understand and remember.
* You can take advantage of the server-side rendering, which will make our application load faster, by rendering the pages in the server first before delivering it the client

Use hash location strategy only if you have to support the older browsers.

### Summary

Angular supports two different location strategies or Routing strategy in Angular. One is PathlocationStrategy and the other one is HashLocationStrategy. The HashLocationStrategy use the Hash style routing, while PathlocationStrategy uses the HTML 5 Routing.

# **Passing Parameters to Route**

## What are Route Parameters

The Route parameters are a dynamic part of the Route and essential in determining the route.

## How to Pass parameters to Angular Route



The name id, id1 & id2 are placeholders for parameters.

### Defining the Navigation



### Retrieve the parameter in the component

Finally, our component needs to extract the route parameter from the URL

This is done via the **ActivatedRoute** service from angular/router module to get the parameter value

## ActviatedRoute

The [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) is a service, which keeps track of the currently activated route associated with the loaded Component.



### ParamMap

**ParamMap**

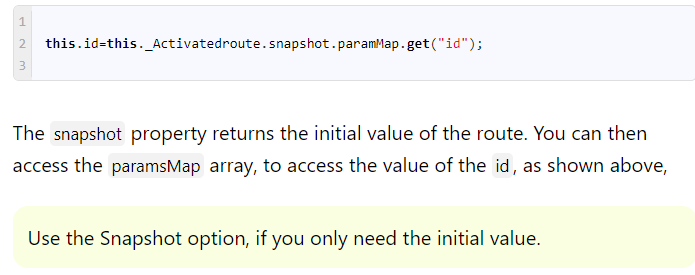
The Angular adds the map all the route parameters in the ParamMap object, which can be accessed from the ActivatedRoute service

The [ParamMap](https://angular.io/api/router/ParamMap) makes it easier to work with parameters. We can use get or getAll methods to retrieve the value of the parameters in the component. Use the has method to check if a certain parameter exists.

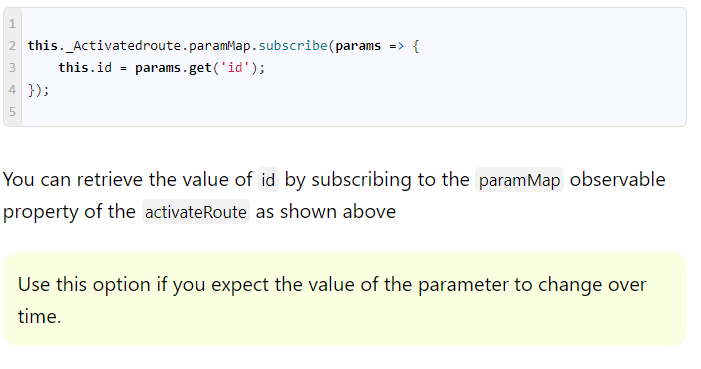
There are two ways in which you can use the ActivatedRoute to get the parameter value from the ParamMap object.

1. Using Snapshot
2. Using observable

### Using Snapshot



### Using Observable



**Why use observable**

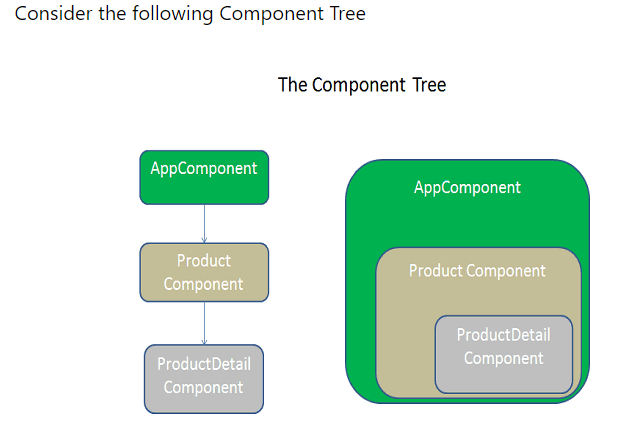
We usually retrieve the value of the parameter in the ngOninit life cycle hook, when the component initialised.

When the user navigates to the component again, the Angular does not create the new component but reuses the existing instance. In such circumstances, the ngOnInit method of the component is not called again. Hence you need a way to get the value of the parameter.

By subscribing to the observable paramMap property, you will retrieve the latest value of the parameter and update the component accordingly.

# **Child Routes / Nested Routes**

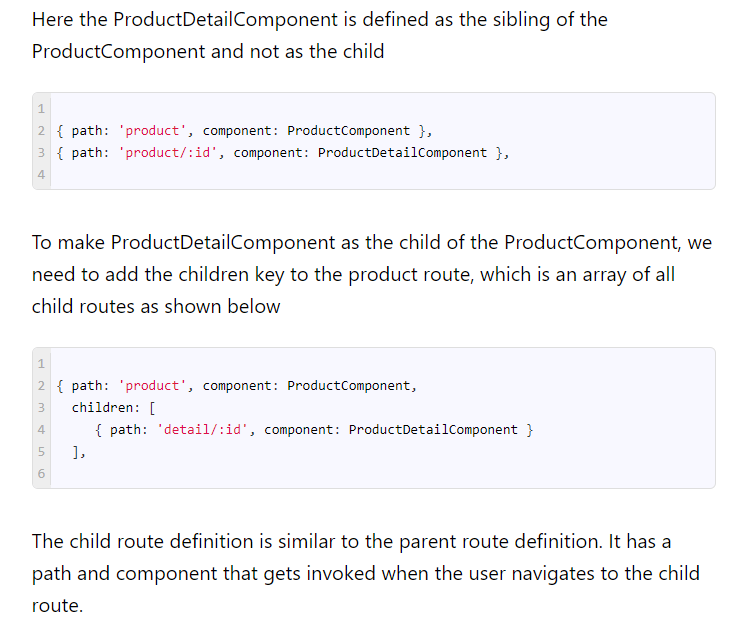
The Angular 2 applications are based on the idea of Components. The Components follows a Tree structure, where we have a root component at the top. We can then add child components forming loosely coupled components resembling a Tree



Our routes would be /Product and /Product/Details/:Id

### Define the Routes

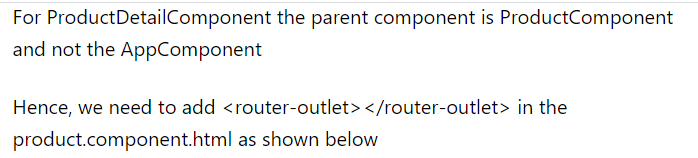
Open the app.routing.ts file

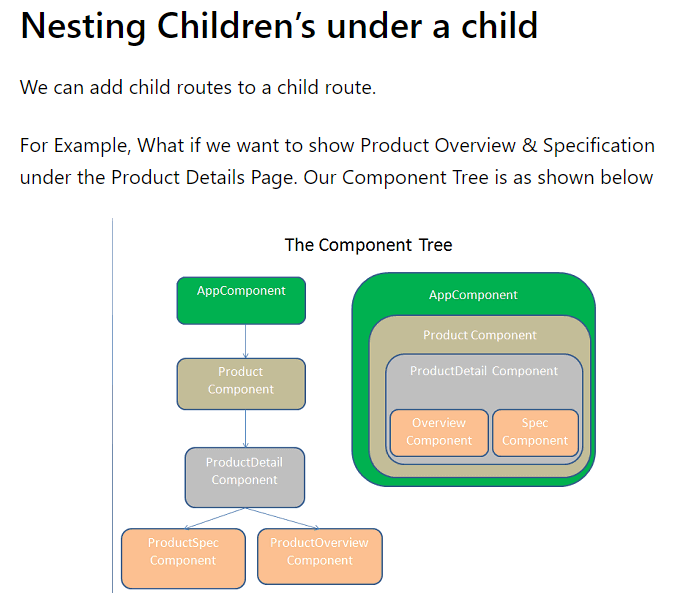


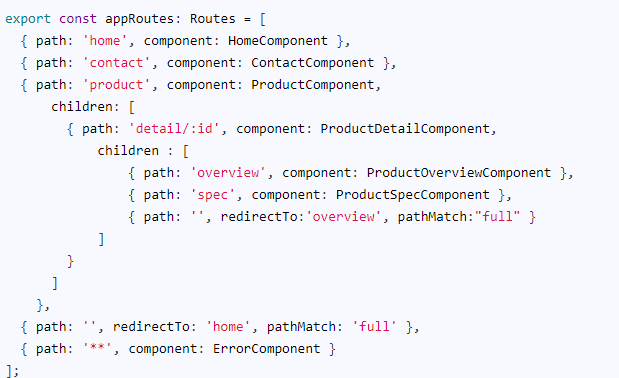
When the user navigates to the “/product/detail/id”, the router will start to look for a match in the routes array

It starts off the first URL segment that is ‘product’ and finds the match in the path ‘product’ and instantiates the ProductComponent and displays it in the <router-outlet> directive of its parent component ( which is AppComponent)

The router then takes the remainder of the URL segment ‘detail/id’ and continues to search for the child routes of Product route. It will match it with the path ‘detail/:id’ and instantiates the ProductDetailComponent and renders it in the <router-outlet> directive present in the ProductComponent







The Url would become ‘/product/detail/:id/overview’ and ‘/product/detail/:id/spec’

The last route is an empty path which is redirected to ‘Overview’ route. Note that **pathMatch** is set to ‘full’

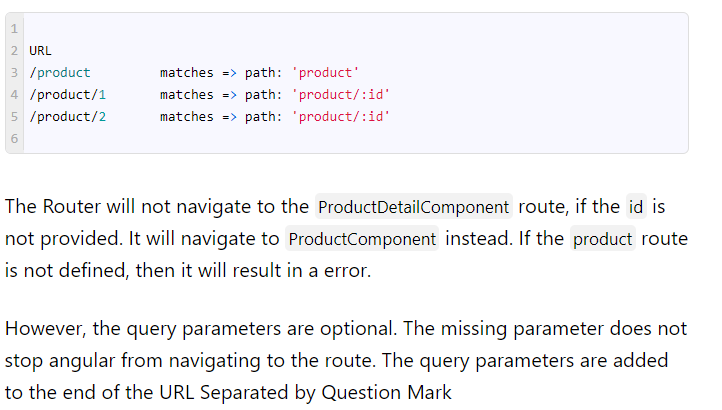
# **Passing Optional (Query) Parameters to a route**

## What are query parameters

Query parameters are optional parameters that you pass to a route

### Difference between Query parameter and Route parameter

The [route parameters](https://www.tektutorialshub.com/angular/angular-passing-parameters-to-route/) are required and is used by [Angular Router](https://www.tektutorialshub.com/angular/angular-routing-navigation/) to determine the route. They are part of the route definition.



**Route Parameters or Query Parameters?**

* Use route parameter when the value is required
* Use query parameter, when the value is optional.

**How to use Query Parameters**

The Query parameters are not part of the route. Hence you do not define them in the routes array like route parameters. You can add them using the routerlink directive or via router.navigate method.

**Passing Query Parameters**

|  |  |
| --- | --- |
|  |  |
| Use the queryParams directive to add the query parameter. Use this directive along with the routerlink directive as shown below      You can also navigate programmatically using the navigate method of the Router service as shown below   Reading Query Parameters Reading the Query parameters is similar to reading the Router Parameter. There are two ways by which you can retrieve the query parameters.  Note that queryParams is deprecated. It is replaced by the queryParamMap. Using queryParamsMap observable The queryParamsMap is a Observable that contains a [map](https://angular.io/api/router/ParamMap) of the query parameters available to the current route. We can use this to retrieve values from the query parameter. The queryParamsMap is accessible via ActivatedRoute   Using snapshot.queryParamMap property  queryParamsHandling The query parameter is lost when the user navigates to another route.  For Example, if user navigates to the product page with route /product?pageNum=2 then he navigates to the product detail page, the angular removes the query parameter from the url. This is the default behaviour  You can change this behavior by configuring the queryParamsHandling strategy. This Configuration strategy determines how the angular router handles query parameters, when user navigates away from the current route. It has three options queryParamsHandling : null This is default option. The angular removes the query parameter from the URL, when navigating to the next..  this.router.navigate(['product'], { queryParams: { pageNum: this.pageNo + 1 }, queryParamsHandling :null}   );  <a [routerLink]="['product']" [queryParams]="{ pageNum:2 }">Page 2</a> queryParamsHandling : preserve The Angular preserves or carry forwards the query parameter of the current route to next navigation. Any query parameters of the next route are discarded.  this.router.navigate(['product'], { queryParams: { pageNum: this.pageNo + 1 }, queryParamsHandling :"preserve"}   );  <a [routerLink]="['product']" [queryParams]="{ pageNum:2 }" queryParamsHandling="preserve">Page 2</a> queryParamsHandling : merge The Angular merges the query parameters from the current route with that of next route before navigating to the next route.  this.router.navigate(['product'], { queryParams: { pageNum: this.pageNo + 1 }, queryParamsHandling :"merge"}   );  <a [routerLink]="['product']" [queryParams]="{ pageNum:2 }" queryParamsHandling="merge">Page 2</a> **RouterLinkActive in Angular** The RouterLinkActive is a directive for adding or removing classes from an HTML element that is bound to a RouterLink. Using this directive, we can toggle CSS classes for active Router Links based on the current RouterState. The main use case of this directive is to highlight which route is currently active.    The Angular does this by watching the URL. Whenever the Url matches with the URL of the routerLink directive, it applies the classes defined in the RouterLinkActive directive. When it does not match it will be removed from the element. Multiple classes  Child Routes When the child route is active, then all the parent routes are also marked as active and routerLinkActive is applied to URL tree cascading all the way to the top.  For Example    When the URL is either /product/PC or /product/mobile, the RouterLinkactive class (i.e. class1 class2) is added to the /product element also as it is the parent of these child routes. Exact matching You can stop that from happening by, passing the exact: true to the RouterLinkactive using the routerLinkActiveOptions as shown below  <a routerLink="/product" routerLinkActive="class1 class1" [routerLinkActiveOptions]="{exact:  true}">Product</a>  <a routerLink="/product/PC" routerLinkActive="class1 class2">PC</a>  <a routerLink="/product/mobile" routerLinkActive="class1 class2">Mobile</a>  **Matching**  The routerActiveLink follows the following criteria before returning **true**  **without exact: true**   * A subset of the [queryParams](https://www.tektutorialshub.com/angular/angular-passing-optional-query-parameters-to-route/) is matched. * The URL is a subtree of the URL tree. * Matrix params are ignored   **with exact: true**   * A [queryParams](https://www.tektutorialshub.com/angular/angular-passing-optional-query-parameters-to-route/) must match exactly * The URL must match exactly * Matrix params are ignored   **Adding classes to ancestors**     **Angular Router Events**The [Angular Routers](https://www.tektutorialshub.com/angular/angular-routing-navigation/) triggers several events starting with when the Navigation starts ( NavigationStart ) and also when the Navigation end ( NavigationEnd ) successfully. It is triggered when the navigation is canceled either by the user ( NavigationCancel ) or due to an error in the navigation ( NavigationError). The Events trigger when the lazy loaded modules are about to load and when they finish loading. They trigger before and after the guards like [canActivate](https://www.tektutorialshub.com/angular/angular-canactivate-guard-example/), [canActivateChild](https://www.tektutorialshub.com/angular/angular-canactivatechild-example/). Events fire before and after the Angular runs the [Route Resolvers](https://www.tektutorialshub.com/angular/angular-resolve-guard/).How to Listen to Router Events First, we need to import the event, which we want to listen to.  And finally in you can listen to events by subscribing to the router.events observable.  The router.events is an Observable that gets triggered when the route changes. We receive NavigationEvent as the parameter in the callback. We check the NavigationEvent instance to check the Type of event fired. Summary The Router events allow us to watch for the router state changes and run some custom logic. One of the use case scenarios is to show the loading indicator when the user navigates from one route to another. Navigating between Angular routes You can navigate between routes in Angular 2 in two ways   1. Using RouterLink Directive 2. Via Code |  |

### Navigating Using Code

You can also navigate imperatively by using the code. This is done using the router service, which provides navigate and navigatebyUrl methods via which you can perform route changes.

#### router.navigate

Use this method, if you want to Navigate to a route using the link parameters array. The first argument to the navigate method is link parameters array, which is similar to what we provide while defining the routerlink directive

***Navigate Method always uses the absolute path unless you provide a starting point.***

navigate.navigateByUrl

Use this method if you want to navigate to a URL by using the absolute path. The first argument is a string containing the complete URL.

To use both these methods, we need to inject router service into our component as shown below



### Link Parameters array

LINK Parameters array is an array of strings, which you must specify as argument to either to routerlink directive or navigate method for navigation to work

We need to specify the path of the route and route parameters that go into the route URL.



## Relative and Absolute Paths in Routes

The Angular routes resemble directory-like tree structure.

Hence, We can use directory like syntaxes like add / (root node) , ./ (current node) or ../ (Parent node) in the link parameters array

The First segment of the link parameters array can be prepended with “**/**“, “**./**“, or “**../**“

If the First segment of the route starts with “**/**“, then the path is considered to be Absolute path

If the First segment begins with “**./”** or it does not begin with a slash, then the path is considered to be the relative path.

And if the First segment begins with “.**./**“, then the path is relative to the parent route. (one level up) **Relative and Absolute Paths in Routes**

And if the First segment begins with “.**./**“, then the path is relative to the parent route. (one level up)

**router.navigate method and relative path**

As mentioned earlier navigate method always uses the absolute path. To make Navigate method work with a relative path, we must let know the router where are we in the route tree.

|  |
| --- |
| This done by setting the relativeTo Property to the ActivatedRoute as shown below |

this.\_router.navigate(['detail'], { queryParams: { pageNum: this.pageNum + 1 }, relativeTo: this.\_Activatedroute } );

### RouterLink directive and relative path

If you were using a RouterLink to navigate instead of the Router service, you’d use the same link parameters array, but you wouldn’t provide the object with the relativeTo property. The ActivatedRoute is implicit in a RouterLink directive.

### Absolute Path Vs Relative Path Which one to Use?

It is recommended to use the Relative path. Using absolute path breaks our code if the parent URL structure changes. The relative path will not change even if the parent path changes

To go to the parent route



## NavigationExtras

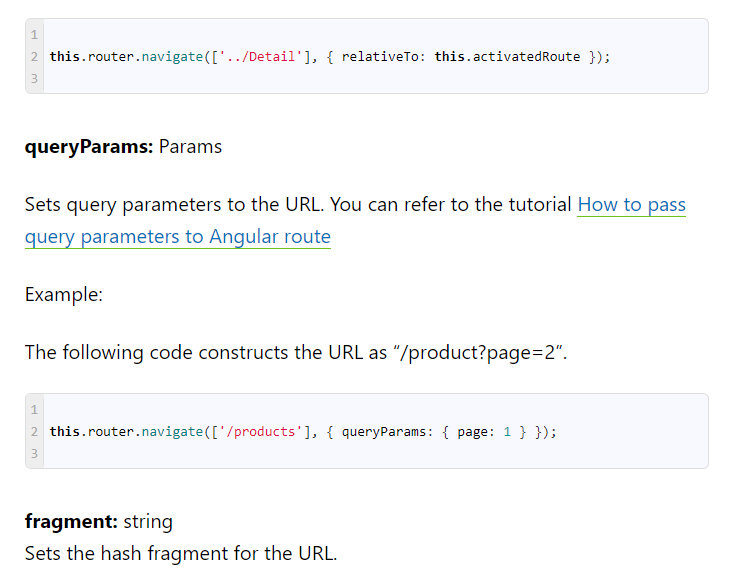
We can provide the extra options to both router.navigate() or router.navigatebyURL() method.

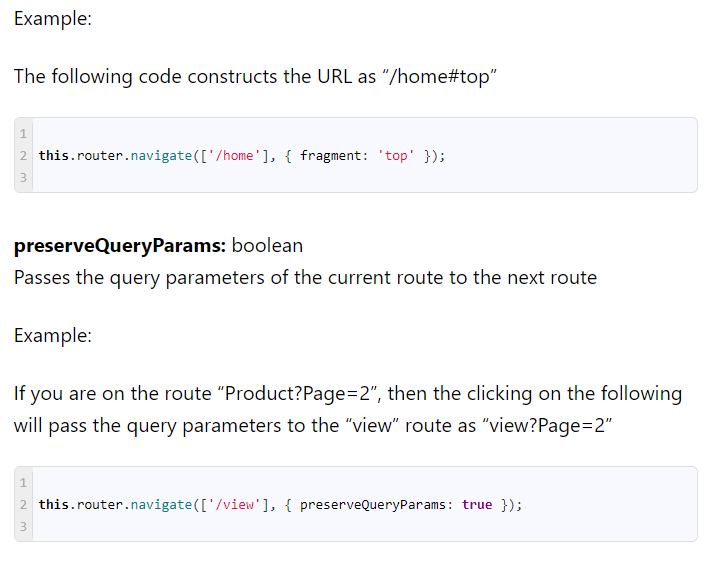
**relativeTo:**ActivatedRoute

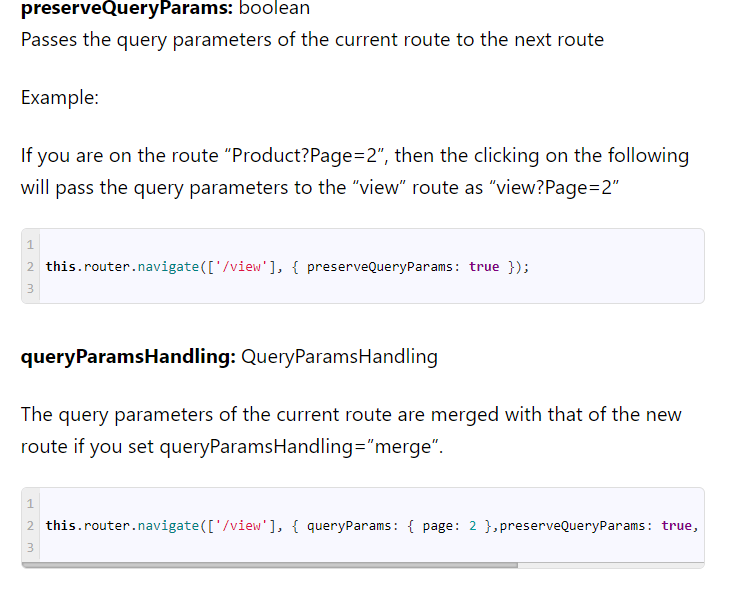
Enables relative navigation from the current ActivatedRoute. This is applicable only to router.navigate() method.

Example:

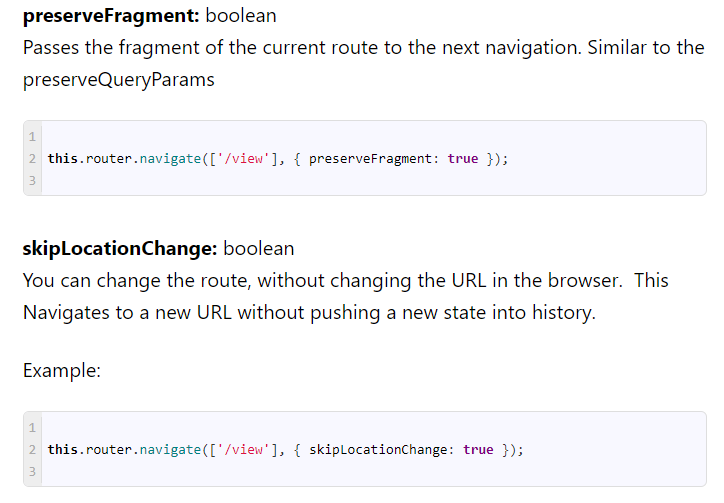
The following Navigates to the Detail route from child route

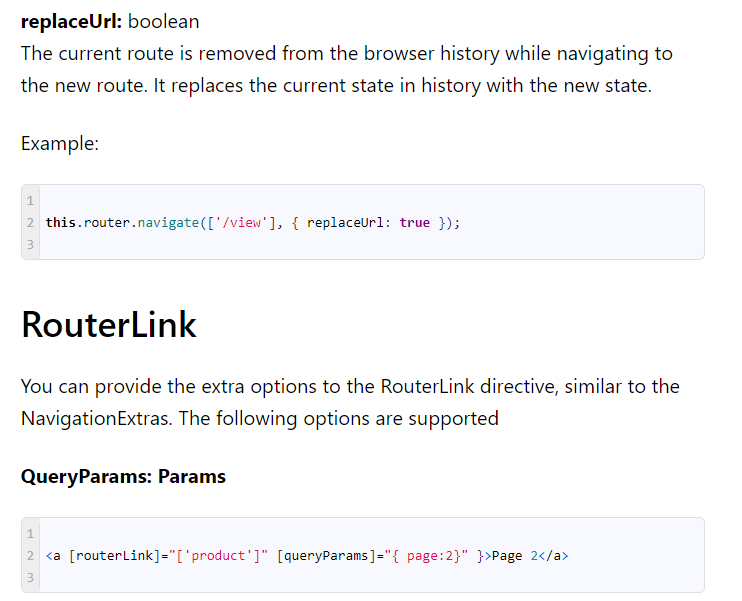


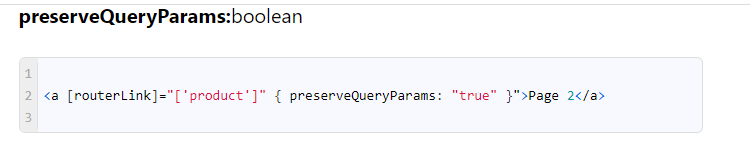




this.router.navigate(['/view'], { queryParams: { page: 2 },preserveQueryParams: true, queryParamsHandling: "merge" });

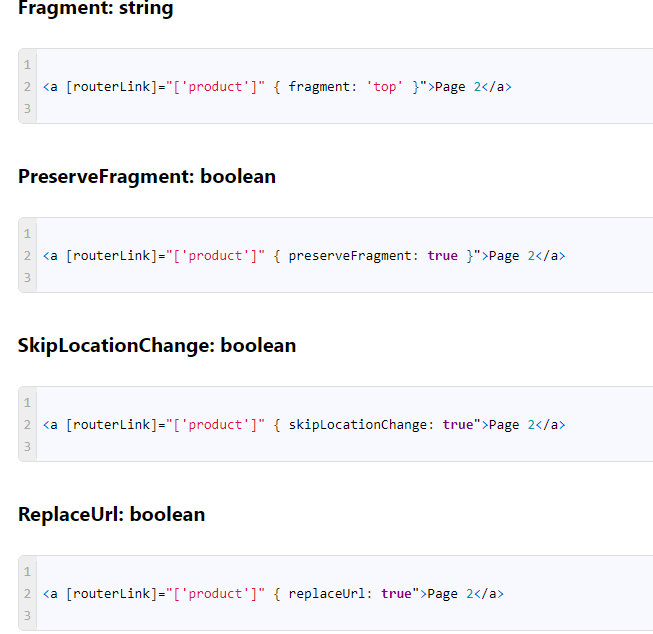






**queryParamsHandling : QueryParamsHandling**

<a [routerLink]="['product']" { queryParams: { page: 2 }, queryParamsHandling: "merge" }">Page 2</a>



### Conclusion

In this tutorial, we looked at various ways you can navigate between Angular routes, using the Angular router.  We learn how to navigate using either RouterLink directive or using router.navigate or router.navigatebyUrl method provided by the router service. We learned how to set up relative and absolute path routing. Finally, we looked at the various options that are provided by these navigation methods.

# **Angular Pass Data to Route: Dynamic/Static**

Angular allows us to pass data to the route. The route data can be either static or dynamic. The static data use the [**Angular route**](https://www.tektutorialshub.com/angular/angular-navigating-between-routes/)**data property**, where you can store arbitrary data associated with this specific route. For to pass dynamic data (or an object), we can make use of the history state object. The Routed Component can then retrieve the dynamic data from the history state object.

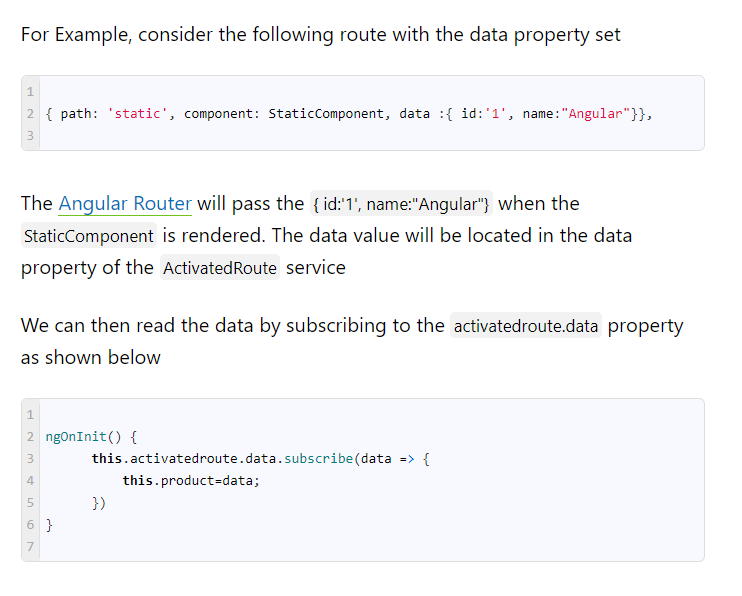
## Various ways of passing route data

The Angular can pass data to Route in several ways.

1. Using URL or [Route Parameter](https://www.tektutorialshub.com/angular/angular-passing-parameters-to-route/)
2. The [Optional Parameter or Query Strings](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/)
3. Using URL Fragment
4. Static data using the data property
5. Dynamic data using state object

## Passing static data to a route

We can configure the static data at the time of defining the route. This is done by using the **Angular route data property** of the [route](https://www.tektutorialshub.com/angular/angular-routing-navigation/#route). The route data property can contain an array of arbitrary string key-value pairs. You can use the static data to store items such as page titles, breadcrumb text, and other read-only, static data



## Passing Dynamic data to a Route

The option to pass the dynamic data or a user-defined object was added in the **Angular Version 7.2** using the state object. The state object is stored in [History API](https://developer.mozilla.org/en-US/docs/Web/API/History)

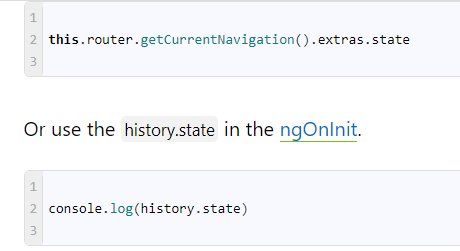
### Providing the State value

The state can be provided in two ways

#### 

### Accessing the state value

The state can be accessed by using the getCurrentNavigation method of the router (works only in the constructor)





# **Angular Guards**

We use the Angular Guards to control, whether the user can navigate to or away from the current route.

## Why Guards

We looked at how to configure our routes and navigate to the different parts of our application in our [Angular Router Tutorial](https://www.tektutorialshub.com/angular/angular-routing-navigation/). Allowing the user to navigate all parts of the application is not a good idea. We need to restrict the user until the user performs specific actions like login. Angular provides the **Route Guards** for this purpose.

One of the common scenario, where we use Route guards is authentication. We want our App to stop the unauthorized user from accessing the protected route. We achieve this by using the **[CanActivate guard](https://www.tektutorialshub.com/angular/angular-canactivate-guard-example/)**, which angular invokes when the user tries to navigate into the protected route. Then we hook into the **[CanActivate guard](https://www.tektutorialshub.com/angular/angular-canactivate-guard-example/)** and use the authentication service to check whether the user is authorized to use the route and if not we can redirect the user to the login page.

## Uses of  Angular Route Guards

* To Confirm the navigational operation
* Asking whether to save before moving away from a view
* Allow access to certain parts of the application to specific users
* Validating the route parameters before navigating to the route
* Fetching some data before you display the component.

## Types of Route Guards

The Angular Router supports Five different guards, which you can use to protect the route

1. CanActivate
2. CanDeactivate
3. Resolve
4. CanLoad
5. CanActivateChild

### CanActivate

This guard decides if a route can be activated (or component gets used). This guard is useful in the circumstance where the user is not authorized to navigate to the target component. Or the user might not be logged into the system

### CanDeactivate

This Guard decides if the user can leave the component (navigate away from the current route). This route is useful in where the user might have some pending changes, which was not saved. The CanDeactivate route allows us to ask user confirmation before leaving the component.  You might ask the user if it’s OK to discard pending changes rather than save them.

### Resolve

This guard delays the activation of the route until some tasks are complete. You can use the guard to pre-fetch the data from the backend API, before activating the route

### CanLoad

The CanLoad Guard prevents the loading of the [Lazy Loaded Module](https://www.tektutorialshub.com/angular/angular-lazy-loading/). We generally use this guard when we do not want to unauthorized user to be able to even see the source code of the module.

This guard works similar to CanActivate guard with one difference. The CanActivate guard prevents a particular route being accessed. The CanLoad prevents entire lazy loaded module from being downloaded, Hence protecting all the routes within that module.

### CanActivateChild

This guard determines whether a child route can be activated. This guard is very similar to CanActivateGuard. We apply this guard to the parent route. The Angular invokes this guard whenever the user tries to navigate to any of its child route. This allows us to check some condition and decide whether to proceed with the navigation or cancel it.

## How to Build Angular Route Guards

Building the Guards are very easy.

1. Build the Guard as Service.
2. Implement the Guard Method in the Service
3. Register the Guard Service in the Root Module
4. Update the Routes to use the guards

### 1. Build the Guard as Service

Building the Guard Service is as simple as building any other [Angular Service](https://www.tektutorialshub.com/angular/angular-services/). You need to import the corresponding guard from the Angular Router Library using the Import statement. For Example to use CanActivate Guard import the CanActivate in the import  the CanActivate in the import statement

|  |  |
| --- | --- |
|  |  |

Next, create the Guard class which implement the selected guard Interface as shown below.

|  |  |
| --- | --- |
|  | @Injectable()  export class ProductGuardService implements CanActivate {} |

You can also inject other services into the Guards using the [Dependency Injection](https://www.tektutorialshub.com/angular/how-dependency-injection-resolution-works-in-angular/)

### 2. Implement the Guard Method

The next step is to create the Guard Method. The name of the Guard method is same as the Guard it implements. For Example to implement the CanActivate guard, create a method CanActivate

|  |  |
| --- | --- |
|  | canActivate(): boolean {      // Check weather the route can be activated;      return true;      // or false if you want to cancel the navigation;  } |

#### Return value from the Guard

The guard method must return either a True or a False value.

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

The above method returns a True value. The Guard can also return an Observable or a Promise which eventually returns a True or false. The Angular will keep the user waiting until the guard returns true or false.

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

### 3. Register the Guard as Service in Module

As mentioned earlier, guards are nothing but services. We need to register them with the Providers array of the [Angular Module](https://www.tektutorialshub.com/angular/angular-modules/) as shown below

|  |  |
| --- | --- |
|  | providers: [ProductService,ProductGuardService] |

The [Angular router](https://www.tektutorialshub.com/angular/angular-navigating-between-routes/) requires the Guards and all other services that guard depends on available during the navigation. Hence the guards must be provided at the module level. This allows the router to access the guards using the [Dependency Injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/).

### 4. Update the Routes

Finally, we need to add the guards to the routes array as shown below

|  |  |
| --- | --- |
|  | { path: 'product', component: ProductComponent, canActivate : [ProductGuardService]  } |

The above code adds the canActivate guard (ProductGuardService) to the  Product route.

When the user navigates to the Product route the Angular calls the canActivate method from the ProductGuardService. If the method returns true then the ProductComponent is rendered.

The syntax for adding other guards are also similar

|  |  |
| --- | --- |
|  |  |

#### Order of execution of route guards

A route can have multiple guards and you can have guards at every level of a routing hierarchy.

CanDeactivate() and CanActivateChild() guards are always checked first. The checking starts from the deepest child route to the top.

CanActivate() guard is checked next and checking starts from the top to the deepest child route.

CanLoad() is invoked next,  If the feature module is to be loaded asynchronously.

Resolve() Guard is invoked last.

The Angular Router cancels the navigation If any of the guards return false.