Routing:

* Till this point, we haven’t used routing in its true sense. We mimicked routing by using the \*ngIf directive. This, although it successfully changed the view, it did not update the URL path in the browser.
* Routing can be applied to an app in a few steps:

1. Import the Routes type and RouterModule from @angular/router.
2. Define a constant of Routes type which takes an array of objects.
3. Each object contains the *path-name* and the *component* to load. The landing page or the home page has path as ‘’.
4. Add the RouterModule to the imports section of @NgModule. We also call the forRoot(<ROUTES>) function available in the RouterModule, where <ROUTES> is the constant that we declared above. The function call is done while add to the imports section itself. We also export the same RouterModule.
5. To actually change the route, we use the router-outlet directive in the location where we want to render the component in the HTML view.
6. Set the href attribute of the respective anchor tags to the absolute or relative paths.

* After adding the paths in the respective anchor tags, we observe that the app reloads every time we navigate. This isn’t how a single-page application should work.
* To fix this, we use the routerLink directive in place of the href attribute. It holds the same value as mentioned above. Additionally, we can set the value using property binding - this can be useful when we have to bind non-string values to the path or if we have to deal with deep links.
* The routerLink directive, with property binding, can take values in a string format (single-quotations) or an array. The first entry in the array will be the main path (containing the ‘/’). The other entries will be just the name or the relevant value.
* We can add custom class based on the active link by using the routerLinkActive directive. It can be applied to the wrapping element or to the link itself. This class is applied to the tag and all its active children as well.
* If we want to add the active class to a specific link, we can use the routerLinkActiveOptions directive, which takes an object as the value. We set the property exact as true to achieve this.
* Sometimes, we do not want to navigate directly, but want to perform some calculation and then route to the relevant page. This can be done by accessing the Router in the component. The Router class can be injected into a component TS file via the constructor.
* After injection, we can use the router instance to navigate to a different page via the navigate([<PATH>]) function. It takes a string array as an argument. Each entry corresponds to either the absolute or relative path to the respective page.
* To navigate to a relative path, we must add some options to the navigate function. This is because the navigate function does not know whether the current path is the root or not.
* Hence we add a second argument to the function, which is an object. We set the relativeTo property in this object and it takes a Route as the value. The current route can be injected via the constructor using the ActivatedRoute instance.

Dynamic Routes:

* There will be cases where we have to route to a path based on some parameters or dynamic data in the URL. This is known as dynamic routing. We can achieve the same in Angular as well. The dynamic route is specified in the app module along with the routes.
* The path is written as <PATH>/:<PARAM>. The dynamic value can be accessed using the ActivatedRoute instance. This will contain a lot of meta-data regarding the current route. It also gives us the dynamic parameters we had entered. This is retrieved via the snapshot.<PARAM>.
* Snapshots work when the component is first initialized. For subsequent changes in the URL, I.e, if we call the same component again just by changing the URL, the URL params will get updated but the data in the component will not change.
* To facilitate this requirement, we use the params *observable* (more on this later). An *Observable* is used to execute asynchronous tasks and keep the system unblocked. We get a callback function which returns the output once the asynchronous work is done.

Angular clean-up:

* Angular usually automatically cleans up our subscriptions and instances. We can also do this using the ngOnDestroy life-cycle method.
* Doing this is a good idea so that we don’t forget how the insides of Angular work.

Query parameters, fragments and data:

* The same way we can add dynamic paths to the URL, we can also add query parameters. We do this in two ways - using queryParams feature of the routerLink directive or the queryParams property in the second argument of the navigate method.
* A Fragment can also added in the same way as mentioned above. Only one fragment is allowed at a time. The only difference is that queryParams takes an object as value and fragment takes string as value.
* Retrieving query parameters and fragments works the same way as dynamic params - either using the snapshot or by using the respective observable.
* We also pass data between route using the data property of the route. This property takes an object as value. The data can be accessed in the same way as above. This can be especially useful if we have a component we want to reuse but need to change some context based on where it called from.

Adding child routes:

* As our app grows, there are going to be many deep links and routing between components.
* To route to a child component, we add the router-outlet directive in the desired location. This will now be the outlet for all routing activities from the current component and its children, unless more outlets are specified within.
* The routes mentioned in the module are optimized for this. All the routes that have a common starting point are pushed into the children array of the parent route.
* A master-detail view can benefit with this kind of routing.
* Query parameters are usually lost when navigating from one component to another. To preserve it, we can add the queryParamsHandling property to the second argument of the navigate function.
* It takes one of 2 values - preserve or merge. Preserve, as the name suggests, preserves the current query parameters even while routing to a child component. Merge, in the other hand, merges the child route query parameters with the current route query parameters.

Re-directing routes:

* Every app has a page that shows up if a non-existing page is requested. In Angular, we can do this via the routing. We can have one route that deals with all erroneous paths. We use a wild-card route - ‘\*\*’.
* We can redirect requests to a specific route from the routes list it self. We have to specify the redirectTo property of the route instead of the component.
* It is important that this route be kept at the bottom of the list of routes in the app module as it redirects all the routes that come to it. If kept at the top of the list, we wont be able to show even available pages.
* Re-direction is a bit tricky. If we have two routes - ‘’ and ‘/recipes’, and we add redirection logic for the first route (the landing page route), then this will match for the second route as well.
* The reason behind this is that Angular matches paths by the prefix. Hence ‘’ and ‘/recipes’ will both execute the redirection. To avoid this, we can use the pathMatch: 'full' property to match the full path.

Handling application routing - the best way:

* Typically, in large projects, we don’t add all the routing logic in the app module itself. We add a new app-routing module which handles all the routing and then injects the routing logic into the main app module.
* We no longer import the RouterModule in the app module. Instead, we import it in the app-routing module and then export it from the same module as well.
* We then import the app-routing module into the main app. We learn more about modules later.

Route Guard:

CanActivate guard

* These are used to execute some logic before or after a route is accessed. This can be very useful when we want to check is whether a user is allowed to access a page or edit some information.
* To do this we add a service in the root folder - the auth-guard.service.ts file. This service then implements the CanActivate interface provided by @angular/router. It forces us to implement the canActivate() in the service.
* This method has 2 arguments - route of type ActivatedRouteSnapshot and state of type RouterStateSnapshot. The method can return 3 types of outputs - an Observable, a Promise or a Boolean value. This way we can execute either synchronous or asynchronous code.
* For the sake of learning, lets create a fake authentication service that returns a promise on whether the user logged in or not. This service is injected into our guard service and used within the canActivate method.
* Finally, our guard service is set on a particular route in our app-routing module. This is done by setting the canActivate property of the route, which takes an array. The guard prevents access to the route if the user is not logged in.
* To enable this guard only to the child routes of a route, we implement another interface named CanActivateChild in the guard service, which has a function named canActivateChild. The structure of this function is the same as canActivate function.
* We then apply this to the relevant parent route. We add canActivateChild property instead of the canActivate property in the route. The value assigned is the same as canActivate property.

CanDeactivate guard

* Another possibility is if we want to warn the user from navigating away from a view, accidentally or on purpose. This is a bit more complicated than the canActivate method.
* We have to start with creating and exporting our own interface which will contain a canDeactivate method. In the same file, we export a guard class implementing the CanDeactivate<OUR\_INTERFACE> generic type interface, which can be imported from @angular/router.
* This interface forces the component, on which this guard is used, to implement our custom method. This interface also has its own canDeactivate method which has 4 arguments - the current component which implements OUR\_INTERFACE, current route of type ActivatedRouteSnapshot, current state of type RouterStateSnapshot and next state of type RouterStateSnapshot. The method can return 3 types of outputs - an Observable, a Promise or a Boolean value.
* This class then returns the canDeactivate method of OUR\_INTERFACE. This way, any logic we apply at the respective component level will be run so that we have full control of what to display to the user, specific to the component.
* Now we add the canDeactivate property to the respective route on which we to apply this. The value is array of guard classes, just like in canActivate. Here we add the exported guard class.
* At the component level, we implement the exported OUR\_INTERFACE and define the canDeactivate method from this interface. Here we can write our custom logic to handle user exit from view.

Resolve guard (To preload dynamic data)

* It is common to execute some logic before or after the route has been activated. We can do this either in the CanActivate guard or in the ngOnInit function of the component.
* Sometimes we need to fetch data from a server or from a backend system. This data-fetch can be either before the route is rendered or after. If we want to get data before the route is rendered, we use a resolve guard.
* This is also a service. Our class will implement the Resolve<DATA\_TYPE> generic type interface imported from @angular/route. This interface has this resolve method which has 2 arguments - route of type ActivatedRouteSnapshot and state of type RouterStateSnapshot.
* The data type can be defined by us. This method can then send an asynchronous / synchronous call and fetch the data we need. For a component to get this data, we must add a resolve property to the route which takes an **object** as value.
* Just as we fetch query parameters and fragments for a route, we can subscribe to changes in data sent to a route and get the dynamically fetched data in the component.

Hash routes for browser support:

* The final step of any project is deployment. We deploy the apps in a server which is then accessed by the clients. Servers usually deal with the URLs that are requested. When using angular, however, it is angular itself handling the Web client URL.
* There are cases where the Server sends a 404 error even if the route exists within the context of Angular. This is common when dealing with older browsers.
* One way to fix this is by setting the useHash property to true. The router module injected into the app takes a second argument, which is an object. In this object, we set the above mentioned property.
* With this, the URL is divided into 2 parts - the part before the hash (to be dealt by the server) and the part after the hash (to be dealt by angular). Only use this in case the app is to be supported by old browsers as well.