**The RouterOutlet Placeholder**

In this lesson, we'll learn how to use the RouterOutlet element to load a component for a route.

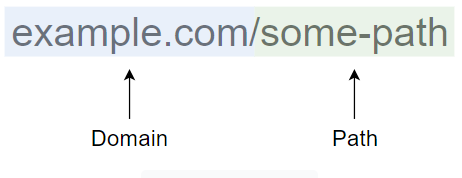
**We'll cover the following**

* [Understanding routing](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4945622186065920#Understanding-routing)
* [Routing module](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4945622186065920#Routing-module)
* [Adding a new route](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4945622186065920#Adding-a-new-route)
* [RouterOutlet](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4945622186065920#RouterOutlet)

We’re able to load the <app-home-main> component in the app’s template. However, it’s not what we’re aiming for. We want to load it if the client is requesting the homepage. The Angular Router can help us with this. First, we’ll need to make some modifications to our modules.

**Understanding routing**

Routing is the idea of tying components to a path in the URL. URLs are mainly comprised of a domain and a path.



URL breakdown

Other things make up a URL, but I’m keeping it simple to explain how routes work. The path portion of a URL is where we request specific resources from a server. When the server serves our application, Angular will grab the path from the URL. It will match the path against the routes registered in our application.

Angular will not magically know which path should render which component. It’s our responsibility to tell Angular what to render when a specific URL is visited. This process is what’s known as routing.

It’s not necessary to add routing to an application, but there are a lot of benefits to doing so. Some benefits are:

* Separating the content into different areas
* Protecting areas of an app based on a specific set of rules
* Easier to maintain

Imagine navigating around an app. For example, let’s say you’re on an e-commerce site. Some things you might do is click on products, view your cart, or checkout. Each of these actions would lead you to different pages. If we built an app like this, there would be a lot of problems if we didn’t dedicate URLs to each page.

* You can’t refresh the page and keep your location within the app.
* You are unable to bookmark a page and revisit it.
* You are unable to share a page with other people.

By adding routes, these issues are non-existent. A URL can point to a specific resource in our application. Angular comes with a routing feature to help us build custom URLs in our application.

**Routing module**

We created a module in the previous lesson with the --routing option in the command. It generated two modules, one of them with -routing in its name. Modules with -routing in their name are called routing modules.

Routing modules are responsible for defining new routes and configuring the router. Every routing module is tied to another module. Angular recommends routing module to have a parallel name to its companion module using the suffix -routing.

For example, if we have a module called home, it’s routing module should be called home-routing. The CLI takes care of adding the suffix for us.

**Adding a new route**

Routes must be defined in routing modules. We’ll define our first route in the home-routing.module.ts file. In this file, there’s a variable called routes. It’s set to an array. Let’s modify it to the following:

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

import { HomeMainComponent } from './home-main/home-main.component';

const routes: Routes = [

  { path: '', component: HomeMainComponent }

];

@NgModule({

  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule]

})

export class HomeRoutingModule { }

In the routes array, we’re adding in an object with two properties. The path property represents the path in the URL. An empty string is the same as passing in a forward slash. We’re creating a route for the homepage, so an empty string suits our case.

If the path in the address bar matches the path in this route, Angular will load the component in the component option. In this case, we’re telling it to load the HomeMainComponent.

We’ll need to update the HomeModule next. We’ll remove the HomeMainComponent from the exports option.

import { CommonModule } from '@angular/common';

import { HomeRoutingModule } from './home-routing.module';

import { HomeMainComponent } from './home-main/home-main.component';

@NgModule({

  declarations: [HomeMainComponent],

  imports: [

    CommonModule,

    HomeRoutingModule

  ],

  exports: []

})

export class HomeModule { }

We’re not exporting the component anymore because the Router will take care of loading the component for us if the route is visited.

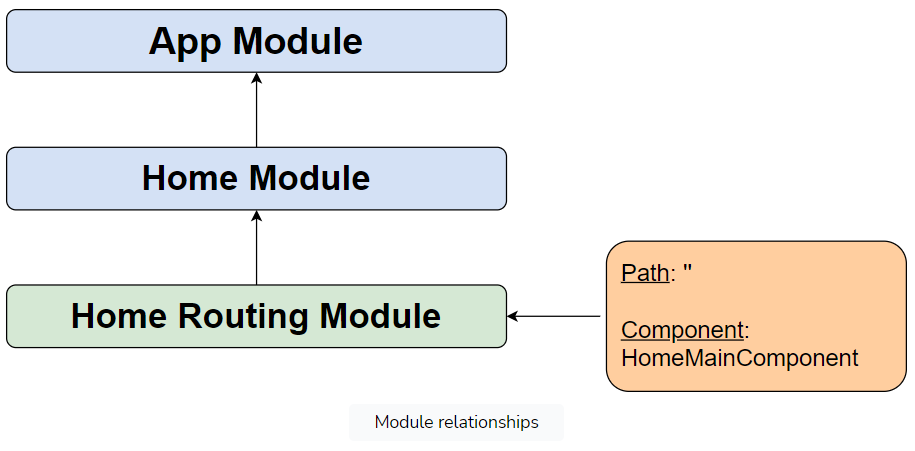
**RouterOutlet**

The Angular Router provides us with a directive that will act as a placeholder for the component. In the app.component.html template file, we’ll replace the contents with the following:

<router-outlet></router-outlet>

The <router-outlet> is a placeholder for the component. The router will replace this placeholder with the component associated with the current browser’s URL. By using this placeholder, we don’t have to export the component. Instead, we can export the module.

We’ve updated a lot of files. Let’s look at a diagram to examine things.



The HomeRoutingModule is a routing module where we can define routes. We have one route where we’ll check if the client is requesting the homepage. If they are, we’ll load the HomeMainComponent.

The home routing module needs to be imported in a module. The home module imports it. Then, we import the home module in the app module. In the app component’s template file, we use the router-outlet directive to load the component registered by the path.

Linking the modules together will result in the HomeMainComponent loading on the page. One important point to make is that the HomeMainComponent can’t be used in the app component’s template file because it’s not exported anymore. Instead, the router is loading the component for us with the placeholder.

# Creating More Routes

In this lesson, we'll get some practice with modules and routing.

**We'll cover the following**

* [The solution](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5764752566910976#The-solution)
  + [Creating components](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5764752566910976#Creating-components)
  + [Defining routes](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5764752566910976#Defining-routes)
  + [Registering the modules](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5764752566910976#Registering-the-modules)

Let’s do an exercise. We were able to create a routing module to load the HomeComponent to display the content for the home page. There are two other pages we’ll want to make routes for. They’re the about and contact pages. We’ve already taken the first steps by creating modules for each page. The next step is to create components and route them.

Try doing this exercise on your own. Here are the series of steps you’ll need to take to accomplish this:

1. Create a component for the contact page.
2. Define a new route in the contact routing module.
3. Import the contact module in the app module.
4. Add the contact module to the imports option.
5. Repeat this for the about page.

Good luck!

## The solution

I won’t leave you hanging. Below the solution for the exercise. For this portion, we’ll be tackling both pages at the same time.

### Creating components

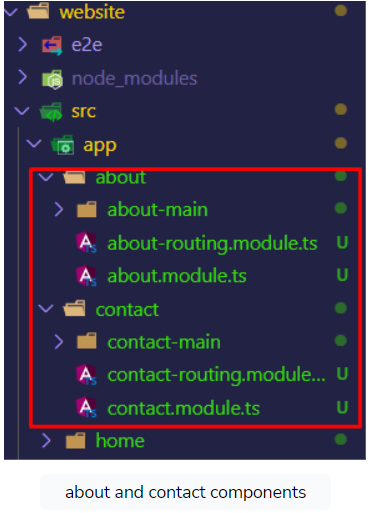
In the command line, run the following commands for the contact component:

ng generate component contact/contact-main

Run this command for the about component:

ng generate component about/about-main

Here’s what should’ve been generated:



### Defining routes

Next, we’ll need to define the routes. In the contact-routing.module.ts file, we’ll update the routes array to the following:

import { ContactMainComponent } from './contact-main/contact-main.component'  
  
const routes: Routes = [  
  { path: 'contact', component: ContactMainComponent }  
];

In the example above, the ContactMainComponent is imported so that we can register it for the contact route. The path property is set to contact. If the client visits /contact, then the router will render the ContactMainComponent.

Afterward, we’ll update the about-routing.module.ts file to the following:

import { AboutMainComponent } from './about-main/about-main.component'  
  
const routes: Routes = [  
  { path: 'about', component: AboutMainComponent }  
];

We do the same thing for the about page. The AboutMainComponent is imported and registered for the about route. If the client visits /about, then the router will render the AboutMainComponent.

### Registering the modules

The routes are ready. We’ll need to import the AboutModule and ContactModule modules into the app module and add them to the imports option.

import { BrowserModule } from '@angular/platform-browser';

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

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { HomeModule } from './home/home.module';

import { AboutModule } from './about/about.module';

import { ContactModule } from './contact/contact.module';

@NgModule({

  declarations: [

    AppComponent,

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    HomeModule,

    AboutModule,

    ContactModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

We’re finished. If we were to run the code, we’d be able to view different pages by changing the path in the URL.

**The RouterLink Directive**

In this lesson, we'll learn how to use the RouterLink directive to navigate around the application.

**We'll cover the following**

* [The history API](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4948243145818112#The-history-API)
* [Base path](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4948243145818112#Base-path)

At the moment, the only way we can navigate around the application is by manually typing the path in the URL. The next thing we’ll tackle is creating some links that we can use to navigate to the other pages.

We installed Bootstrap, which comes with some classes for creating a navigation bar. You can find more information about those classes [here](https://getbootstrap.com/docs/5.0/components/navbar/).

We’ll be using it to help us create the links. Also, we’ll be using Bootstrap to help us center the page content. Update the app.component.html file to the following:

In the widget above, we’re mostly using Bootstrap. The <router-outlet> element is being wrapped with some classes to help center and space it.

The links aren’t using anything special. They’re standard links with href attributes telling the browser to navigate to the other pages.

If we run the example, everything works, but there is one problem. Every time a link is clicked, the browser will refresh the page. The assets for the browser will need to be reloaded.

## The history API

The Angular Router can use the history API to navigate around the application.

**What is the history API?**

The history API enables us to visit different pages without having to reload the assets. This is achieved using the history.pushState() method. It exposes the browser’s navigation history to JavaScript.

If you’d like to learn more about the history API, then check out this [link](https://developer.mozilla.org/en-US/docs/Web/API/History_API).

We’ll want to use it to make loading pages faster. Luckily, we don’t have to do anything to enable the history API. It’s automatically leveraged by Angular. The only thing we need to do is update the <a> element’s href attribute to the routerLink directive.

With this one change, Angular will override the default behavior by leveraging the history API. The URL in the address bar will still change, but the browser will not refresh the page. The assets no longer need to be reloaded. We can navigate around the application much faster this way.

## Base path

One additional thing I want to mention is how Angular constructs paths. It uses what’s known as a **base URL**. In the src/index.html file, you’ll find the following:

<base href="/">

This is a very important tag for the Angular router. It will tell Angular where to find images and other resources that use relative paths. It’s also used to construct routes.

For example, let’s say the base path is /app, and the path for the about page is /about. The URL to the page will be /app/about. The route path is appended to the base URL to determine the full path. In most cases, setting the base path to / will work.

**Active Link Styles**

In this lesson, we'll learn how to make links appear active when a user clicks on them.

**We'll cover the following**

* [Additional note](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5223188027408384#Additional-note)

The next objective is to add the active class to the link the user is currently visiting. While not required, it can help the user identify where they are. In Bootstrap, the active class can be applied to the <li> element wrapped around the <a> element.

Here’s the problem, though: we want to dynamically add the class. The active class shouldn’t be applied to the element if the URLs don’t match. Angular comes with a directive for this called routerLinkActive.

We’re applying the directive on the <li> elements with the nav-item class. This directive will check if the URL in the address bar matches the path in the routerLink directive. If there’s a match, it will apply to the class you pass in. In this case, we’re adding the active class.

If you run the example, the links will change class if the paths in the routerLink directive match the URL in the browser.

## Additional note

In some tutorials and examples, you’ll often see the routerLinkActive and routerlink directives on the same element.

<a routerLink="/example" routerLinkActive="active">Example</a>

In our project, they’re not on the same element because Bootstrap requires us to apply the active class on the <li> element. This can seem problematic because the directives for link and class are on two separate elements. One solution would be to override the CSS, but we don’t have to. The routerLinkActive can be applied to an ancestor element of an element with the routerLink directive. Angular is smart enough to associate the two together.

**Note:** Ancestor elements are parent elements of an element in the DOM tree.

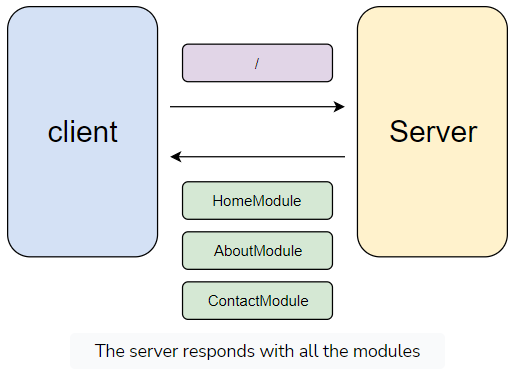
**Lazy Loading Modules**

In this lesson, we'll learn how to improve the performance of the application by utilizing lazy loading.

**We'll cover the following**

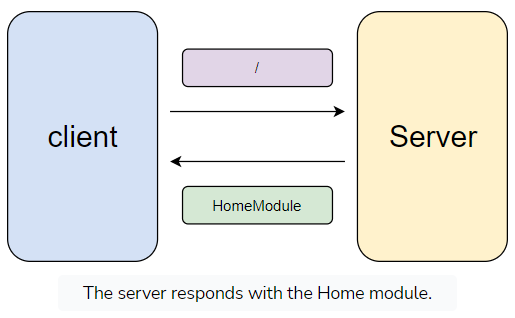
* [Removing the modules](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5057224059125760#Removing-the-modules)
* [Dynamically importing modules](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5057224059125760#Dynamically-importing-modules)
* [Updating the routing modules](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5057224059125760#Updating-the-routing-modules)
* [Verifying lazy loading](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5057224059125760#Verifying-lazy-loading)

Up until now, our app has been using eager loading by default, but what is eager loading? Loading of an application with all the components registered in the app module is called **eager loading**.



In the diagram above, anytime we make a request, the server will always respond with all the modules. The homepage only requires the HomeModule. The other two modules aren’t necessary at the time of the request. Regardless, we’ll still receive those modules from the server because the CLI will bundle the modules together in a single file.

Ideally, it would be better if we could optimize the response by splitting the modules into separate files. Angular allows us to implement lazy loading. Lazy loading is when we load components only when they’re required to render a page.



In this example, only the HomeModule is sent over because we’re requesting the homepage. If we make subsequent requests to the about or contact pages, then the server will respond with the AboutModule and ContactModule files, respectively.

Lazy loading is much more optimal because it means pages will load faster. The app will be able to load with the bare minimum. Additional requests can be made to load other sections of the application.

We can implement lazy loading by dynamically importing the modules when they’re requested. Angular makes this process painless.

**Removing the modules**

The first thing we’ll need to do is not import the modules when the application is initially loaded. The app.module.ts file is where we’re loading all the modules. We’ll remove the import statements for the modules and update the imports option to reflect that.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

import { BrowserModule } from '@angular/platform-browser';

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

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

@NgModule({

  declarations: [

    AppComponent,

    PageNotFoundComponent,

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

We’re not going to lazy load every route. We’ll leave the wildcard route with the default bundle. It’s not super impactful on the bundle size.

The modules we’ll dynamically load are the HomeModule, AboutModule, and ContactModule modules.

**Dynamically importing modules**

Next, we need to update the routes in the app-routing.module.ts file. This is the only routing module that will be bundled with the application on the initial request. It’s where we can tell Angular to dynamically load the other modules when a request is made to a specific resource.

1

2

3

4

5

6

const routes: Routes = [

  { path: '', loadChildren: () => import('./home/home.module').then(m => m.HomeModule) },

  { path: 'about', loadChildren: () => import('./about/about.module').then(m => m.AboutModule) },

  { path: 'contact', loadChildren: () => import('./contact/contact.module').then(m => m.ContactModule) },

  { path: '\*\*', component: PageNotFoundComponent },

];

There are two things to note. First, we’re defining the routes above the wildcard route. We don’t want to encounter the issue we had last time where the wildcard route took precedence over the other routes. It should always be the final route on the list.

Second, we’re not using the component property to tell Angular what component to load when the route is visited. Instead, we’re using the loadChildren property. This will tell Angular to lazy load a module.

We’re setting it to an arrow function, where we’ll request the module using the import() function. The import() function returns a promise. We chain the then() function to handle the response.

In the response, we’re passed in the module object, which we give an alias of m. In the function, we’re returning the respective module from the m object. Keep in mind that we’re using named exports in our module files. It’s required that we access the module by its name if we want to return it.

**Updating the routing modules**

Angular will load and process the module we’re dynamically importing. Every module we’re dynamically loading comes with a routing module. At the moment, we aren’t telling Angular which component to load. We’re just telling it to load a module. The module that’s being dynamically imported is responsible for telling Angular which component to load. We can do so in the respective routing module.

We’ll need to adjust the routes we have in the routing modules because of how Angular treats the routes. The routes in the child modules are concatenated with the route responsible for dynamically loading them.

For example, let’s look at the route for the about page in the app-routing.module.ts file.

{ path: 'about', loadChildren: () => import('./about/about.module').then(m => m.AboutModule) }

The routes from the about module will be appended to the path in the route. In the app-routing.module.ts file, we have the following routes:

const routes: Routes = [  
  { path: 'about', component: AboutMainComponent }  
];

Angular will concatenate the paths, which will result in the about/about route. To better understand, let’s try testing the application.

If we were to visit the about page, we’d be greeted with a blank page. This is because the path for the about page is now about/about.

If we want to revert the path to about, then we’ll need to update the routing modules for each page. In the about-routing.module.ts file, we’ll update the path property to an empty string.

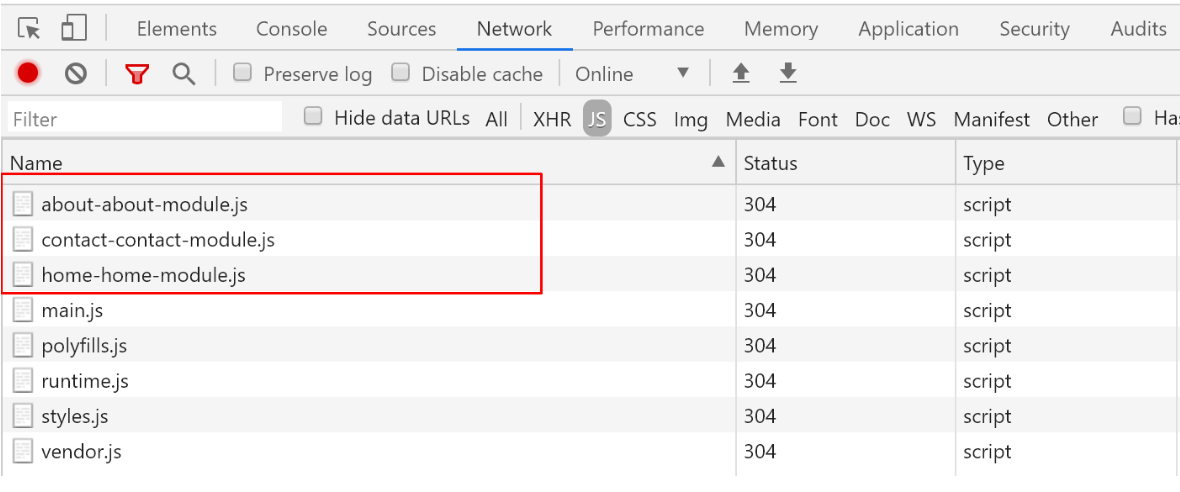
{ path: '', component: AboutMainComponent }

In the contact-routing.module.ts file, we’ll update the path property to an empty string.

{ path: '', component: ContactMainComponent}

We don’t need to update the routing module for the homepage because it’s already an empty string. This will fix the paths.

We can verify that lazy loading is working by checking the developer tools.



If we navigate to the other pages, we’ll find new requests for the other modules. The great thing about this is that Angular will keep track of what modules have already been loaded. If we visit a page that we’ve already visited, the module will not be requested again. Angular will simply load the module from memory.

**The NgContent Directive**

We'll learn how to load content in a child component from a parent component.

**We'll cover the following**

* [Base module](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5523330720858112#Base-module)
* [Alert component](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5523330720858112#Alert-component)
* [Using NgContent](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5523330720858112#Using-NgContent)

The app is fully navigable. Next, we’ll create some base components. Base components are generic components that we’ll most likely use more than once. It’s common to develop base components for things like buttons, alerts, forms, etc.

We’ll create a base component for an alert box. Bootstrap comes with a set of classes for an alert box. You can find more information [here](https://getbootstrap.com/docs/5.0/components/alerts/).

**Base module**

We’ll create a module that will house all our base components. This way, we can import them into any module that needs them. In the command line, run the following command:

ng generate module base

We aren’t adding the --routing option because this will be a module that won’t be tied to a specific resource.

**Alert component**

Next, we’ll create an alert component. In the command line, run the following command:

ng generate component base/alert

The base module will need to be updated to export the component.

angular/core';

import { CommonModule } from '@angular/common';

import { AlertComponent } from './alert/alert.component';

@NgModule({

  declarations: [AlertComponent],

  imports: [

    CommonModule

  ],

  exports: [

    AlertComponent

  ]

})

export class BaseModule { }

Next, we can import the BaseModule anywhere we want to use the AlertComponent. We’ll use the AlertComponent on the homepage. Let’s update the home.module.ts file to import the BaseModule.

import { CommonModule } from '@angular/common';

import { HomeRoutingModule } from './home-routing.module';

import { HomeMainComponent } from './home-main/home-main.component';

import { BaseModule } from '../base/base.module';

@NgModule({

  declarations: [HomeMainComponent],

  imports: [

    CommonModule,

    HomeRoutingModule,

    BaseModule

  ],

  exports: []

})

export class HomeModule { }

Let’s use the component in the home-main.component.html template file.

<app-alert></app-alert>

Lastly, let’s update the template for the alert component in the alert.component.html file.

<div class="alert alert-primary">  
  This is an alert component  
</div>

This works great, but there’s one flaw with the component. It will always render the same message. In most cases, we’ll want to display different content. One way to resolve this issue is passing down data from the parent component to the child component. However, this is far from ideal because we may want to insert buttons, links, and other elements in the alert box.

**Using NgContent**

Angular introduces a directive, called ng-content, that acts as a placeholder for the content provided from the parent component. In HTML, we can insert tags in between other tags. This same idea applies to custom components that we create.

Let’s see it in action to get a better understanding. In the alert.component.html template file, we’ll update the alert box to the following:

<div class="alert alert-primary">  
  <ng-content></ng-content>  
</div>

In the home-main.component.html file, we’ll update the template to the following:

<app-alert>  
  This is an alert. Learn more about alerts  
  <a href="https://getbootstrap.com/docs/4.5/components/alerts/">here</a>.  
</app-alert>

We can pass in whatever we want. The <ng-content> element will be replaced with whatever is passed down from the parent component. It’s an element that gives us the power to determine where to load the content, which makes the AlertComponent reusable.

**Selecting Content**

In this lesson, we'll learn how to insert content into specific locations in a child component.

**We'll cover the following**

* [The select attribute](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5441113101434880#The-select-attribute)
* [One final adjustment](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5441113101434880#One-final-adjustment)

In some cases, we may want to insert content into different locations in a component. We can use the NgContent directive to specify what content can be loaded in its place.

For this example, we’re going to add a heading section for the alert component. Bootstrap supports a heading by adding an <h4> element with the alert-heading class.

">

  <h4 class="alert-heading">Well done!</h4>

  <p>Aww yeah, you successfully read this important alert message. This example text is going to run a bit longer so that you can see how spacing within an alert works with this kind of content.</p>

</div>

It is possible to insert this heading without modifying the component.

<app-alert>

  <h4 class="alert-heading">Well done!</h4>

  <p>Aww yeah, you successfully read this important alert message.</p>

</app-alert>

This works, but the problem with it is that we have to memorize the specific structure and classes required to add a heading. We currently have a simple component. Can you imagine what we’d have to remember if the component was more complex?

**The select attribute**

Luckily, we can create multiple locations to insert content with the NgContent directive. We’ll update the alert.component.html.

<div class="alert alert-primary">

  <h4·class="alert-heading">

····<ng-content·select="[heading]"></ng-content>

··</h4>

  <ng-content></ng-content>

</div>

In the example above, we’re adding the select attribute on the <ng-content> element. Angular will search for an element based on the value of the attribute. If it finds an element, it will replace the placeholder with the content.

The value for the select attribute can be any valid CSS selector. In this example, we’re selecting an element that has the heading attribute.

As for the other <ng-content> element, Angular will replace it with any content that wasn’t selected. They are the leftovers. Let’s update the home-main.component.html file to use the new location.

<app-alert>

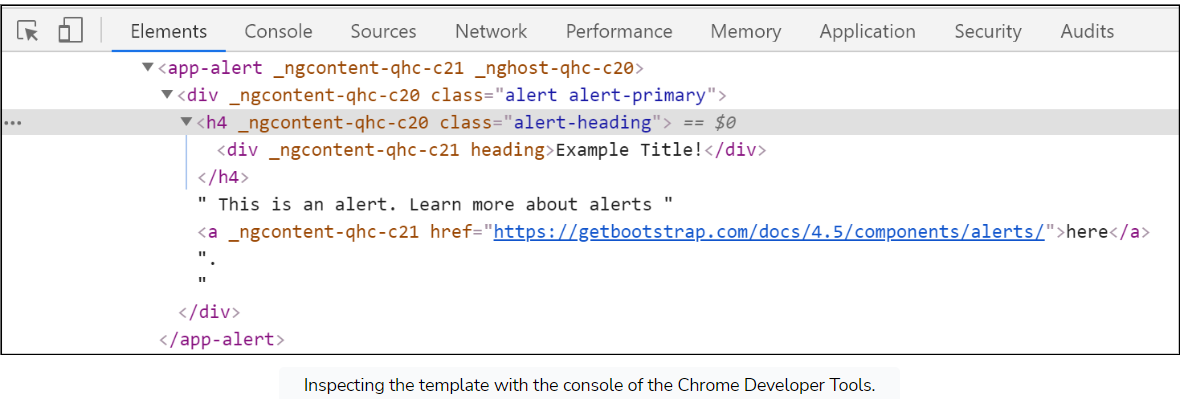
  <div·heading>Example·Title!</div>

··This·is·an·alert.·Learn·more·about·alerts

··<a·href="https://getbootstrap.com/docs/4.5/components/alerts/">here</a>.

</app-alert>

This works without a problem. However, the entire element is copied over. If we peek at the elements in the developer tools, we’ll find that the <div> element was inserted in the <h4> element.



We may not want this if we want to maintain the integrity of the HTML structure. We can replace the <div> element with the <ng-container> element.

<app-alert>

  <ng-container·heading>Example·Title!</ng-container>

··This·is·an·alert.·Learn·more·about·alerts

··<a·href="https://getbootstrap.com/docs/4.5/components/alerts/">here</a>.

</app-alert>

**One final adjustment**

Currently, we’re not making it a requirement to insert a heading. If we don’t insert a heading, the <h4> element will still appear in the document. This will take up space, which may make the alert box look awkward. We can fix this using CSS. We can use the :empty CSS selector to apply styles to elements that are empty.

h4.alert-heading:empty{

  display: none;

}

In the example above, we’re using the :empty pseudo selector to check if the heading is empty. If it is, we’ll hide the element—a straightforward, yet effective solution for hiding the heading if it’s empty.

**Child Routes**

In this lesson, we will learn how to add child routes to an existing route.

**We'll cover the following**

* [What is a child route?](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5265624764252160#What-is-a-child-route)
* [Creating the components](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5265624764252160#Creating-the-components)
* [Defining child routes](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5265624764252160#Defining-child-routes)
* [Loading the child route component](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5265624764252160#Loading-the-child-route-component)

We have an about page that we’re going to transform. We’ll add some dummy sections to it where visitors can view a short bio and learn about the team, and some past clients. We’ll take it a step further by putting each section into a child route.

**What is a child route?**

We can create a hierarchy out of our routes. In some cases, we’ll want routes to be children of other routes. For example, here’s what we’ll want the routes for the about page to look like.

* /about: display the bio
* /about/team: display the team
* /about/clients: Displays clients

We’ll create a component for each page. The components should only get rendered if they’re on the /about path.

**Creating the components**

We’ll quickly run through the process of creating the components. In the command line, run the following command for the bio section:

ng generate component about/bio

For the team section:

ng generate component about/team

For the clients section:

ng generate component about/clients:

**Defining child routes**

One solution for creating child routes is to add them to the app routing module.

const routes: Routes = [

  { path: '', loadChildren: () => import('./home/home.module').then(m => m.HomeModule) },

  { path: 'about', loadChildren: () => import('./about/about.module').then(m => m.AboutModule) },

  { path: 'about/team', component: TeamComponent },

  { path: 'about/clients', component: ClientsComponent },

  { path: 'contact', loadChildren: () => import('./contact/contact.module').then(m => m.ContactModule) },

  { path: '\*\*', component: PageNotFoundComponent },

];

For smaller apps, this works. However, it can clutter the app routing module. For organization reasons, we’ll want to group routes related to one another in a central location. We can take an approach different from the one above.

We can define child routes by adding the children property to a currently existing route. We’ll add this property to the about route in the about-routing.module.ts file since we’re creating child routes for the about page.

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

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

import { AboutMainComponent } from './about-main/about-main.component'

import { BioComponent } from './bio/bio.component';

import { TeamComponent } from './team/team.component';

import { ClientsComponent } from './clients/clients.component';

const routes: Routes = [

  {

    path: '',

    component: AboutMainComponent,

    children: [

      { path: '', component: BioComponent },

      { path: 'team', component: TeamComponent },

      { path: 'clients', component: ClientsComponent },

    ]

  }

];

@NgModule({

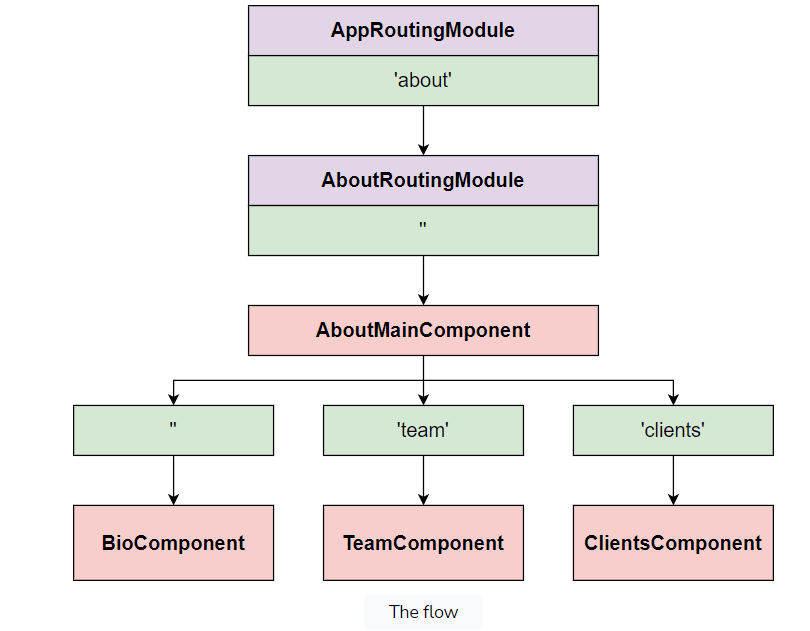
  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule]

})

export class AboutRoutingModule { }

The children property will be set to an array of routes. We can add the path to match and the component to render if the URL matches the path. The value for the path property will be appended to the parent route. Therefore, we don’t need to write the full path. For example, if a parent route is /member and the child route /joe, then the full route would be /member/joe.



The diagram above shows the flow Angular will go through to determine what component to render. Let’s run through an example in which we visit the /about/team route.

1. Angular will check the AppRoutingModule. It will find a partial match since we’re trying to visit a child route of about.
2. The AboutRoutingModule gets loaded.
3. The routing module will load the AboutMainComponent because it partially matches the URL.
4. Angular will then proceed to check the child routes. It will take whatever is after the /about path and match it against the routes. In this scenario, the team path gets matched. Therefore, it will load the TeamComponent.

Child routes are concatenated onto their parent route. Here’s a short summary of how the routes are concatenated together:

* /about = 'about' + '' + ''
* /about/team = 'about' + '' + 'team'
* /about/clients = 'about' + '' + 'clients'

**Loading the child route component**

The last piece of the puzzle is to load the components for the child route. We can do so using the <router-outlet> placeholder. In the about-main.component.html template file, we’ll update the template to the following:

<h1>About Page</h1>  
  
<router-outlet></router-outlet>

We don’t have to do anything special in the template. Angular is intelligent enough to understand that we have child routes. The most important thing is to place the <router-outlet> placeholder in the component where we defined the child routes. The rest will be taken care of for us.

**RouterLink Configuration**

In this lesson, we'll learn how we can configure the RouterLink directive.

**We'll cover the following**

* [Relative links](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5783051117789184#Relative-links)
* [Exact matching](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5783051117789184#Exact-matching)
* [Alternative RouterLink syntax](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5783051117789184#Alternative-RouterLink-syntax)

The last step for creating child routes is to add some links. We can use the routerLink directive to generate the links. First, let’s add the elements to create tabs. Bootstrap comes with classes for creating tabs. You can find more information [here](https://getbootstrap.com/docs/5.0/components/navs-tabs/#javascript-behavior).

For the about page, we’ll be implementing the routerLink directive slightly different from before because we’re dealing with child routes.

**Relative links**

In the about-main.component.html file, we’ll update the template.

<h1>About Page</h1>

<ul class="nav nav-tabs">

  <li class="nav-item">

    <a class="nav-link" routerLink="" routerLinkActive="active">

      Bio

    </a>

  </li>

  <li class="nav-item">

    <a class="nav-link" routerLink="team" routerLinkActive="active">

      Team

    </a>

  </li>

  <li class="nav-item">

    <a class="nav-link" routerLink="clients" routerLinkActive="active">

      Clients

    </a>

  </li>

</ul>

<router-outlet></router-outlet>

In the example above, we’ve created some links with the routerLink directive. The values for the directive aren’t ideal. If we were to use the links above, the visitor would be redirected to the home page.

This is because an empty value will tell Angular to redirect the visitor to the root path. If we want to redirect the visitor to /about, then we can use relative links.

Angular allows us to use ./ to point to the current route. We can update our links to use ./ so that we’re always referencing /about when navigating around.

<h1>About Page</h1>

<ul class="nav nav-tabs">

  <li class="nav-item">

    <a class="nav-link" routerLink="./" routerLinkActive="active">

      Bio

    </a>

  </li>

  <li class="nav-item">

    <a class="nav-link" routerLink="./team" routerLinkActive="active">

      Team

    </a>

  </li>

  <li class="nav-item">

    <a class="nav-link" routerLink="./clients" routerLinkActive="active">

      Clients

    </a>

  </li>

</ul>

<router-outlet></router-outlet>

**Exact matching**

We’re getting closer to what we want. One last problem is the routerLinkActive directive will always apply the active class to the <a> element, even when we change pages. This is because the directive will allow for partial matching. Since it’s checking if the path *contains* /about, then it will always match with the other child routes.

We can override this behavior by telling Angular to use exact matching instead of partial matching. An additional directive, routerLinkActiveOptions, can be added to configure how the router works. We’ll add it to the <a> element.

<h1>About Page</h1>

<ul class="nav nav-tabs">

  <li class="nav-item">

    <a class="nav-link" routerLink="./" routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }">

      Bio

    </a>

  </li>

  <li class="nav-item">

    <a class="nav-link" routerLink="./team" routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }">

      Team

    </a>

  </li>

  <li class="nav-item">

    <a class="nav-link" routerLink="./clients" routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }">

      Clients

    </a>

  </li>

</ul>

<router-outlet></router-outlet>

In the example above, we’re binding the directive because we want to set the value of the directive to an object. We’re setting the exact property to true. This will tell Angular to use exact matching.