The Three Pillars of Angular Routing. Angular Router Series Introduction.

Nate Lapinski Follow
Sep 4, 2018 · 8 min read

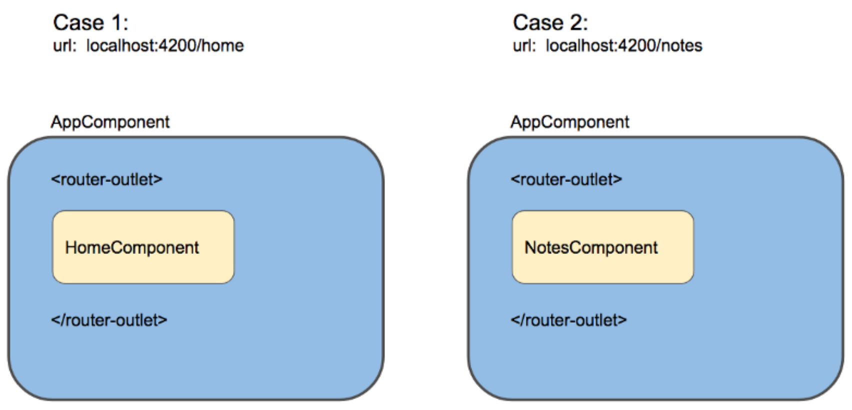
Tokyo’s Rainbow Bridge

The Angular router is a marvel of software engineering. From handling application navigation to enforcing route guards and facilitating lazy loading of modules, Angular’s router is indispensable for most applications. However, for many developers, the internal workings of the router remain a mystery. **This series aims to change that, by giving you, the developer, a deeper understanding of the router.** This introductory article will provide an overview of the router’s architecture, as well as some useful mental models for gaining an intuition of how routing works in Angular. The rest of this series will delve deeper into each part of the router’s architecture and implementation.

Having a deeper understanding of the router will enable you to customize it to your needs, should you ever need to change the default behavior. Who knows? Knowing the architecture of the router might even help you solve difficult problems at work.

A Tree of States

An Angular application is a tree of components. Some of those components, such as the root component, will remain in place over the course of the application. **However, we also want the ability to display certain components dynamically, and one way to achieve this is through use of the router.** Using the router module along with router-outlet directives, it's possible to define parts of our application which will display different sets of components based on the current url. For example, in a simple note-taking application, you might want to display a home component for one url, and a list of notes for a different url.



Depending on the url (home or notes), a different component will be rendered using a router-outlet directive.

Internally, these *routable* sets of components are known as router states. The router will model the routable components in an application **as a tree of *router states*.** Continuing with the example above, the home page would be a router state, while the components for displaying a list of notes would be another router state.

The core focus of the router is to enable navigation among routable components within an Angular application, which requires the router to **render a set of components using an outlet on the page, and then reflect the rendered state in the url.** In order to do this, the router needs some way to associate urls with the appropriate set of components to load. It accomplishes this by letting a developer define a router state configuration object, which describes which components to display for a given url.

Router states are defined within an application by importing the RouterModule, and passing an array of Route objects into its `forRoot` method. For example, an array of routes for a simple application might look like this:

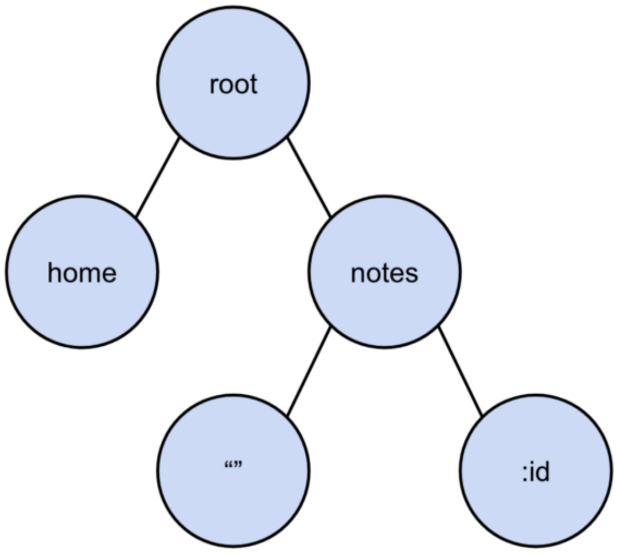
```
import { RouterModule, Route } from '@angular/router';

const ROUTES: Route[] = [
  { path: 'home', component: HomeComponent },
  { path: 'notes',
    children: [
      { path: '', component: NotesComponent },
      { path: ':id', component: NoteComponent }
    ]
  },
];

```

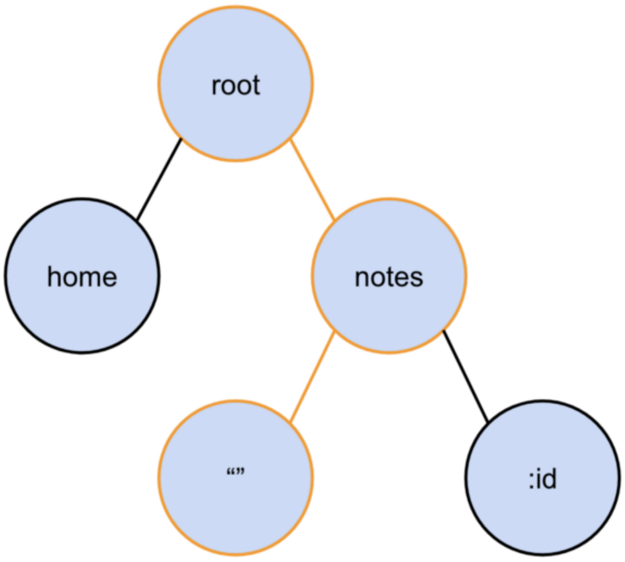
An array of Routes describes all possible router states for an application.

It will produce the following tree of router states when passed into `routerModule.forRoot()` :



The tree of router states generated from the above configuration.

An important point is that at any time, some router state (i.e. arrangement of components) is being displayed on screen to the user, based on the current url. This arrangement is known as the active route. **An active route is just some subtree of the tree of all router states**. For instance, the url `/notes` would be represented as the following active route:



The router state corresponding to the url /notes. The activated router state is highlighted in orange. Given our example router configuration, this would result in the NotesComponent being rendered on screen.

Some points of interest regarding route configurations:

1. The RouterModule has a `forChild` method, which also accepts an array of Routes. While both `forChild` and `forRoot` return modules containing all of the router directives and route configurations, `forRoot` also creates an instance of the Router service. **Since the Router service mutates the browser location, which is a shared global resource, there can be only one active Router service**. This is why you should use `forRoot` only once in your application, in the root app module. Feature modules should use `forChild` .
2. When a route’s path is matched, the components referenced inside of the router state’s `component` properties are rendered using router-*outlets,* which are dynamic elements that display an activated component. Technically, the components will be

rendered as a *sibling* to the router outlet directive, not inside of it. Router outlets can also be nested within one another, forming parent/child route relationships.

Whenever navigation occurs within the application, **the router will take the url it's navigating to, and try to match it against a path in the router state tree.** For example, given the same configuration as before:

```
const ROUTES: Route[] = [
  { path: 'home', component: HomeComponent },
  { path: 'notes',
    children: [
      { path: '', component: NotesComponent },
      { path: ':id', component: NoteComponent }
    ]
```

the url `localhost:4200/notes/15` would route to and load the `NoteComponent`. Internally, the NoteComponent will be able to access the parameter `15`, and display the appropriate note. A path value that starts with a colon, such as `:id,` is known as a required parameter, and will match almost anything (in this case, it matches 15). A path like `localhost:4200/iamerror` would fail to match any path, and would generate an error.

At any given point in time, **the URL represents a serialized version of the application's currently activated router state**. Changes in the router state will change the URL, and changes in the URL will change the router state. They are both representations of the same thing.

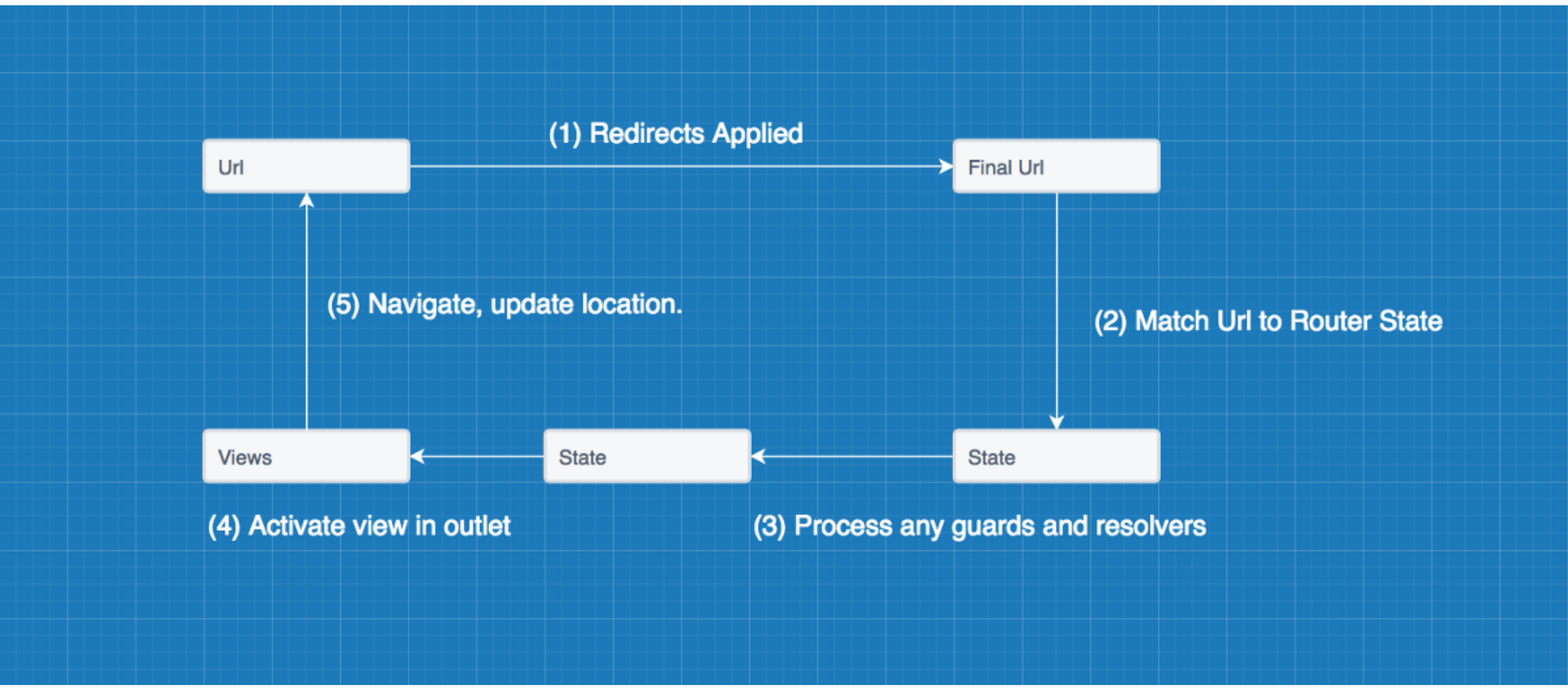
We'll see the internals of the algorithms that the router uses to match a path against a route in the next article in this series. For now, it's sufficient to know that it uses a first-match-wins strategy. Internally, this is implemented as a depth-first search, with the router matching the first path that consumes an entire url.

Understanding how Angular can model all routing possibilities in an application as a tree of router states is the first pillar of routing. The second pillar, navigation, describes how to move from one router state to another.

. . .

The Lifecycle of a Router

Similar to component lifecycles, the router also has a series of steps that it cycles through each time the router state changes.


(1) Redirects Applied
Url
Final Url
(5) Navigate, update location.
(2) Match Url to Router State
Views
State
State
(4) Activate view in outlet
(3) Process any guards and resolvers

The router navigation cycle runs whenever there is a change to router state or the url.

During this **navigation cycle, the router emits a series of events**. The Router service provides an observable for listening to router events, which can be used to define logic, such as running a loading animation, as well as aiding in debugging routing. Some noteworthy events during this cycle are:

`NavigationStart:` Represents the start of a navigation cycle. `NavigationCancel:` For instance, a guard refuses to navigate to a route. `RoutesRecognized:` When a url has been matched to a route. `NavigationEnd:` Triggered when navigation ends successfully.

A complete list of events which extend the `RouterEvent` class can be found here.

```
const ROUTES: Route[] = [
  { path: 'home', component: HomeComponent },
  { path: 'notes',
    children: [
      { path: '', component: NotesComponent },
      { path: ':id', component: NoteComponent }
    ]
```

Given the above configuration, let's consider what happens when given the url `http://localhost:4200/notes/42` .The overview is as follows.

1. First, any redirects must be processed, since there is no sense in trying to match a url to a router state until we have a finalized version of that url. Since there are no redirects in this case, the url stays as is and is unchanged.

2. Next, the router uses a **first-match-wins-with-backtracking** strategy to match the url to a router state defined in the configuration. In this case, it will match `path: ‘notes'` , and then `path:’:id'` . The `NoteComponent` is associated with this route.

3. Since a matching router state was found, the router then checks if there are any guards associated with that router state, which might prevent navigation. For instance, maybe only users who are logged in can view notes. In this example, there are no guards. We're not using any resolvers to prefetch data for this route either, so the router proceeds with the navigation.

4. The router then activates the component associated with this route state.

5. The router finishes navigation. Then it waits for another change to the router state/url, and repeats the process all over again.

These events can be viewed in the browser console by passing an **enableTrace: true** option to the router's **forRoot** method.

```
RouterModule.forRoot(
  ROUTES,
  {
    enableTracing: true
  }
```

Alternatively, a component can access the stream of router events by injecting the `Router` service, and subscribing to its `events` observable:

```
constructor(private router: Router) {
  this.router.events.subscribe( (event: RouterEvent) =>
}
```

You can see some examples of router events by navigating between views in this Stackblitz, and checking the console:

router-navigation-lifecycle - StackBlitz

Starter project for Angular apps that exports to the Angular CLI

stackblitz.com

The navigation article in this series will explore this cycle and its events in-depth.

If all the router did was manage the navigation lifecycle and define the application's route states, it would be invaluable. But as we'll see in the third pillar, the router goes a step further and allows us to lazily load feature modules.

Lazy Loading Feature Modules

The third pillar of Angular routing is the concept of **lazy loading modules**. As an application grows over time, more and more of its functionality will be encapsulated in separate feature modules. For instance, a website that sells books might have modules such as books, users, etc. **Chances are, not all of this data will be displayed when the application first loads, so there is no reason to include all of it in the main bundle**. It will only bloat that file, and cause longer download times when loading the application. It is better to load these modules on demand whenever a user navigates to them, and it is through lazy loading that the Angular router achieves this.

An example of lazy loading a module looks like this:

```
// from the Angular docs https://angular.io/guide/lazy-
{
  path: 'customers',
  loadChildren: 'app/customers/customers.module#Custome
```

Using loadChildren to specify that the customers module should be lazily loaded.

Note that the value passed to `loadChildren` is a string, not a component reference. Care must be taken to avoid any reference to any part of the lazily loaded module (such as importing the module). Otherwise, there will be a compile-time dependency on that module, and Angular will have to include it in the main bundle, thereby defeating the purpose of lazy loading.

The router will start fetching any lazily loaded modules during the apply redirects / url matching phase of the navigation cycle:

```
/**
 * Returns the `UrlTree` with the redirection applied.
 *
 * Lazy modules are loaded along the way.
 */
export function applyRedirects(
    moduleInjector: Injector, configLoader: RouterConf
    urlTree: UrlTree, config: Routes): Observable<UrlT
```

As is mentioned in config.ts:

The router will use registered NgModuleFactoryLoader to fetch an NgModule associated with the loadChildren string. Then it will extract the set of routes defined in that NgModule, and will transparently add those routes to the main configuration.

So the routes defined in the lazily loaded module's configuration will be loaded into the main configuration, and can then be matched against and routed to.

We'll have a lot more to say on lazy loading later in this series.

. . .

The rest of this series will provide a deeper dive into the implementation details of each of the three pillars above, starting with router states and path matching. Please stay tuned!

Read the rest of the series here:

Router States and URL Matching

The Router’s Navigation Cycle

Lazy Loading and Preloading