The Three Pillars of the Angular Router—Router States and URL Matching

Nate Lapinski Follow
Sep 17, 2018 · 11 min read

A gray day in Tokyo Bay

In the introductory article for this series, we glanced over the architecture of Angular's router, and defined **three pillars of the router: router states, navigation, and lazy loading.** This article will delve into the first pillar, and discuss how the router matches a URL to a set of `{path:'',...}` objects in the router configuration, which define the *router states of the application*. **The goal for this article is to gain an in-depth understanding of what happens from the moment the router gets a new URL,**

until it is successfully matched against a route path. We'll learn about the following topics, in depth:

1. URL structure
2. URL redirects
3. Matching URLs to route configuration objects
4. Router state, activated routes, and state snapshots

• • •

A Tree of States

As discussed in the introduction to this series, the router views the routable portions of an application as a tree of *router states,* which are defined by router configuration objects:

```
{ path: '...', component: ...}
```

Router configurations are specified declaratively within an application by importing the RouterModule, and passing an array of Route configurations to `RouterModule.forRoot()`. Consider the configuration for the sample application shown below, which has its route configuration objects inside of the `ROUTES` array:

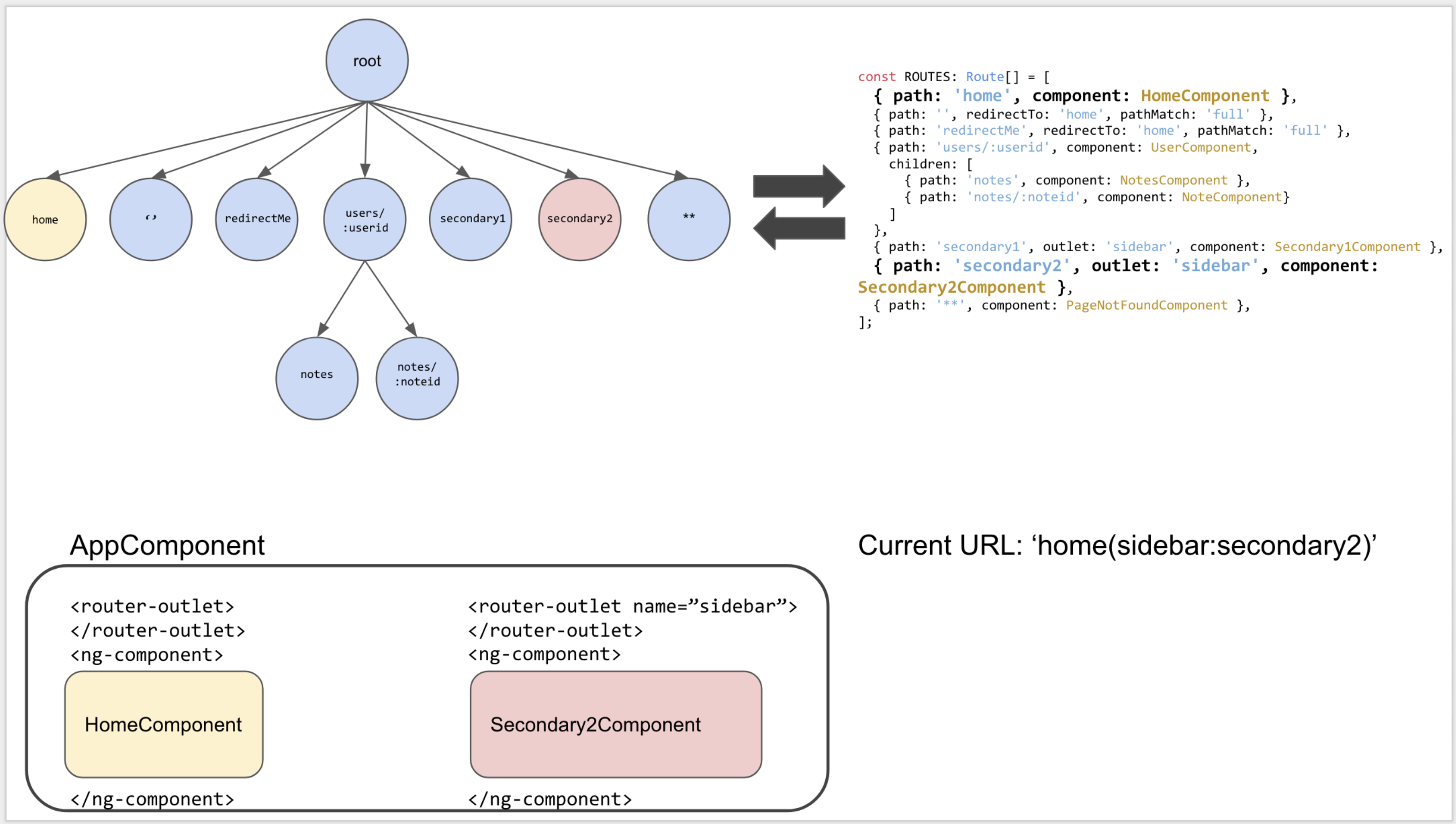
```
const ROUTES: Route[] = [
  { path: 'home', component: HomeComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'redirectMe', redirectTo: 'home', pathMatch:
  { path: 'users/:userid', component: UserComponent,
    children: [
      { path: 'notes', component: NotesComponent },
      { path: 'notes/:noteid', component: NoteComponen
    ]
  },
```

Simple application available at this stackblitz

A Route object defines a relationship between some routable state in your application (components, redirects,

etc), and a segment of a URL. The structure of a Route object is simple. In most cases, a `path` to match a URL segment against, and a `component` to load when that path is matched are all that is needed. As we'll see later, components are rendered using `<router-outlet>` directives. An application can have named `<router-outlet>` directives as well, which are known as *secondary outlets*. If you'd like to know more about secondary outlets, I've written a small primer on them.

By the end of this article, we will understand the following diagram, which shows an example of a URL being consumed and matched against configurations in the ROUTES array.



```
const ROUTES: Route[] = [
  { path: 'home', component: HomeComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'redirectMe', redirectTo: 'home', pathMatch: 'full' },
  { path: 'users/:userid', component: UserComponent,
    children: [
      { path: 'notes', component: NotesComponent },
      { path: 'notes/:noteid', component: NoteComponent}
    ]
  },
  { path: 'secondary1', outlet: 'sidebar', component: Secondary1Component },
  { path: 'secondary2', outlet: 'sidebar', component:
Secondary2Component },
  { path: '**', component: PageNotFoundComponent },
];
```

```
<router-outlet>
</router-outlet>
<ng-component>
</ng-component>
```

```
<router-outlet name=”sidebar”>
</router-outlet>
<ng-component>
</ng-component>
```

The relationship between the ROUTES config(top right), the tree representation of that config (top left), the current state of the application (bottom left), and the current URL being routed to (bottom right). Note that the root component is not part of a URL, it's only shown for illustration purposes. Also note that the router will place routed components inside of an <ng-component> element, as a sibling to the <router-outlet> directive.

You can play around with the above `ROUTES` at this stackblitz.

Our first task is to understand how the router handles URLs internally.

. . .

Urls and UrlSegmentGroups

Let's start by understanding the different parts of a URL, and how they are represented internally by the router.

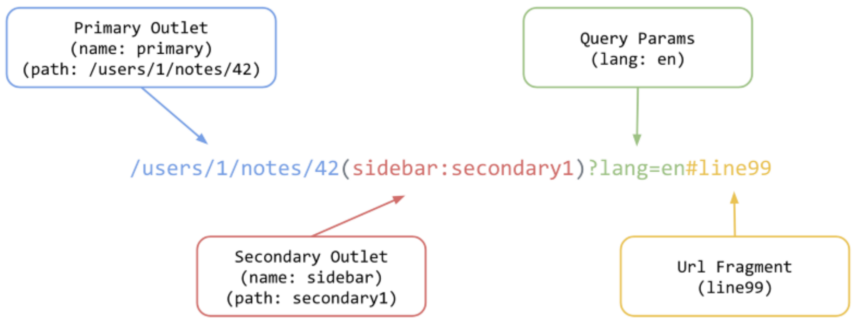
Consider the following simple URL:

```
/users/1/notes/42
```

It is composed of four separate *segments*: `users` , `1` , `notes` , and `42` . It does not contain any additional parameters, or any secondary router outlets.

Given the simplicity of that URL, we might expect the router to store URLs as strings internally. **But, since URLs are serializations of router state, which can be complex, the router needs a more sophisticated structure for representing URLs internally**.

To illustrate, consider the following URL, which contains a secondary outlet, as well as query parameters, and a fragment:



A more complicated URL. Secondary outlets are placed within parenthesis. Query parameters and fragments are common across routes.

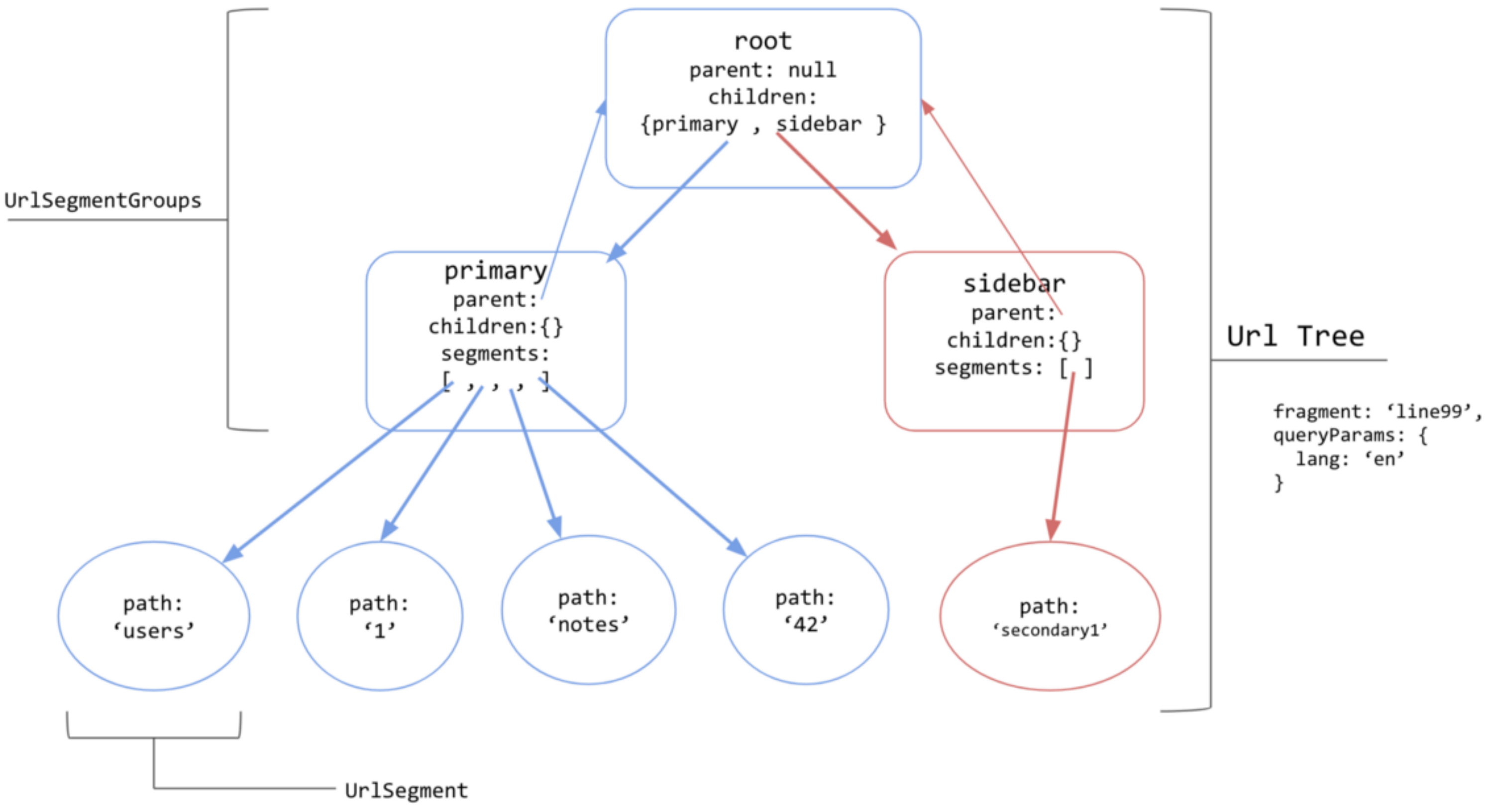
We can break this down using the Router service:

```
1   const url = '/users/1/notes/42(sidebar:secondary1)?lang
2   const tree = this.router.parseUrl(url); // '/users/1/nc
3
4   const fragment = tree.fragment;          // line99
5   const queryParams = tree.queryParams;    // lang=en
6   const primary: UrlSegmentGroup = tree.root.children[PRI
7   const sidebar: UrlSegmentGroup = tree.root.children['si
```

Data structures used to represent a url internally

You can experiment with the code at this Stackblitz link. I recommend taking the time to inspect the URL data structures in the console.

Calling `router.parseUrl(url)` on `line 2` will convert the URL string into the following tree structure:



Tree structure generated from the url '/users/1/notes/42(sidebar:secondary1)?lang=en#line99'. Some properties of objects have been omitted for brevity. Primary outlet is in blue, secondary sidebar outlet is in red.

1. The entire URL is represented as a UrlTree.

2. Interior nodes of the tree (those which have child nodes of UrlSegments) are represented as UrlSegmentGroups. These are usually associated with a specific router outlet, such as `primary` and `sidebar` in the example above.

3. Leaf nodes (those with no children) are represented as UrlSegments. A UrlSegment is any part of a URL occurring between two slashes, for instance `/users/1/notes/42` has four segments, `users` `1` `notes` and `42`. **These are what will be matched to `path` properties in the router configurations in `ROUTES`.** UrlSegments can also contain *matrix parameters*, which are data specific to a segment. Matrix parameters are separated by semicolons `;`, such as `name` and `type` in the example `/users;name=nate;type=admin/`.

4. The root node has a child UrlSegmentGroup for each outlet. In this case, it has two; one for the default outlet (primary), and one for the secondary outlet (sidebar). Internally, the router serializes secondary outlets in the URL within parenthesis, such as `(secondary_outlet_name:secondary_path_name)`, and matches them to configuration objects which have a matching `outlet` property, such as `{path: 'secondary_path_name', outlet: 'secondary_outlet_name'}`. **We'll see later that outlets are routed independently of each other.**

5. Fragments and query params live as properties on the UrlTree.

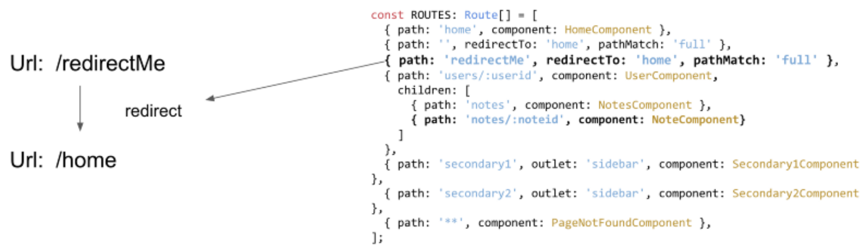
A new UrlTree is generated each time the URL changes. UrlTree creation happens synchronously, and independently from the task of matching the URL to something in the `ROUTES` configuration tree. **This is an important distinction because matching may be asynchronous.** For instance, matching might require a router configuration from a lazily-loaded module to be loaded asynchronously. We'll see more on this in the next section on redirects.

. . .

Applying Redirects

Whenever the URL changes, the router will try to match it against routes in the `ROUTES` array. **The first thing the router does is apply any redirects defined for each segment of the URL.**

Redirects simply replace a URL segment with something else (or in the case of an absolute redirect, they replace the entire URL). Internally, a new UrlTree will be created, which reflects the redirect. You can define a redirect in a route configuration by specifying `{redirectTo: 'some_path'}` .



```
const ROUTES: Route[] = [
  { path: 'home', component: HomeComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'redirectMe', redirectTo: 'home', pathMatch: 'full' },
  { path: 'users/:userid', component: UserComponent,
    children: [
      { path: 'notes', component: NotesComponent },
      { path: 'notes/:noteid', component: NoteComponent}
    ]
  },
  { path: 'secondary1', outlet: 'sidebar', component: Secondary1Component
},
  { path: 'secondary2', outlet: 'sidebar', component: Secondary2Component
},
  { path: '**', component: PageNotFoundComponent },
];
```

Processing a simple redirect from /redirectMe to /home

Why would you ever want to do this? Redirect transformations are applied to a URL before it is matched against a router state, which means that **redirects are very useful for normalizing URLs or performing refactors.** Want both `legacy/user/name` and `user/name` to render the same component? Just use a redirect to normalize the URLs:

```
// normalize a legacy url
[
  { path: 'legacy/user/:name', redirectTo: 'user/:name'
  { path: 'user/:name', component: UserComponent}
```

Internally, the router uses a function called applyRedirects to process redirects:

```
function applyRedirects(
    moduleInjector: Injector, configLoader: RouterConfi
    urlTree: UrlTree, config: Routes): Observable<UrlTr
  return new ApplyRedirects(moduleInjector, configLoade
```

Seems like a lot of parameters just to apply a redirect! Let's highlight some of them.

configLoader: An instance of RouterConfigLoader. This is used for compiling and loading any lazily loaded modules encountered along the way. **You never know, the URL we are trying to match**

might take us to a module we haven't loaded yet. The loader will bring in the lazy module's router config (have a look at its load function).

urlSerializer: We've met this before. Used for transforming URL strings to UrlTrees and back again.

urlTree: The tree structure representing our URL.

config: This is the `ROUTES` array that we passed into `forRoot`. It is what the router will compare URL segments against.

For any URL segment**, the router has no idea if it should be redirected or not ahead of time,** so at each route whose `path` matches that segment, the router checks if that path has a `redirectTo` property. Redirects can happen at each level of nesting in the router config tree, but can happen only once per level. **This is to avoid any infinite redirect loops.**

```
if (allowRedirects && this.allowRedirects) {
    return this.expandSegmentAgainstRouteUsingRedirect(
        ngModule, segmentGroup, routes, route, paths, c
  }
```

For example:

```
{ path: 'redirectMe', redirectTo: 'home', pathMatch: 'full'
}
```

If `redirectTo` is set, and the `path` matches the current URL segment (explained in the next section), `expandSegmentAgainstRouteUsingRedirect` is called to apply the redirect.

The `pathMatch` property can be either `full` or `prefix`, and it determines how the router matches URL segments to `path`s. We'll cover matching in the next section, but for now, `prefix` just checks that the `path` is a prefix of the remaining URL segments, and is the default. A value of `full` will check that the `path` fully matches the remaining segments of the URL. For redirects, `full` is usually used, since we often want to redirect the empty path `path: ‘’` to some other route. If `prefix` were used in this case, `path: ''` will match everything, since the empty string is a prefix of every string. You can read more on the differences between the two here.

Once a redirect is applied, a new UrlTree is generated to match against the router config.

```
private applyRedirectCreatreUrlTree(
    redirectTo: string, urlTree: UrlTree, segments: Url
    posParams: {[k: string]: UrlSegment}): UrlTree {
  const newRoot = this.createSegmentGroup(redirectTo, u
  return new UrlTree(
      newRoot, this.createQueryParams(urlTree.queryPara
```

The input to the "Apply Redirects" phase of routing is a UrlTree, and the output is also a UrlTree, with redirects applied.

We now know how a URL is represented as a tree, and how redirects create new UrlTrees. Let's see how a URL is matched against an actual route path.

. . .

URL Matching

At the heart of the router lies a **powerful** URL matching engine. Without the ability to associate URLs with the appropriate set of components to render, **navigation within an application would not be possible**.

For this section on matching, we'll use the following array of `ROUTES`, since it will let us see the details of the matching algorithm clearly.

```
const ROUTES = [
  { path: 'view1', component: View1Component },
  { path: 'view2', component: View2Component,
    children: [
      { path: ':id', component: DisplayIdComponent }
    ]
  },
  { path: 'l1',
    children: [
      { path: 'l2',
        children: [
          { path: 'l3',
            children: [
              { path: 'view3', component: View3Compone
            ] }
        ] }
    ]
```

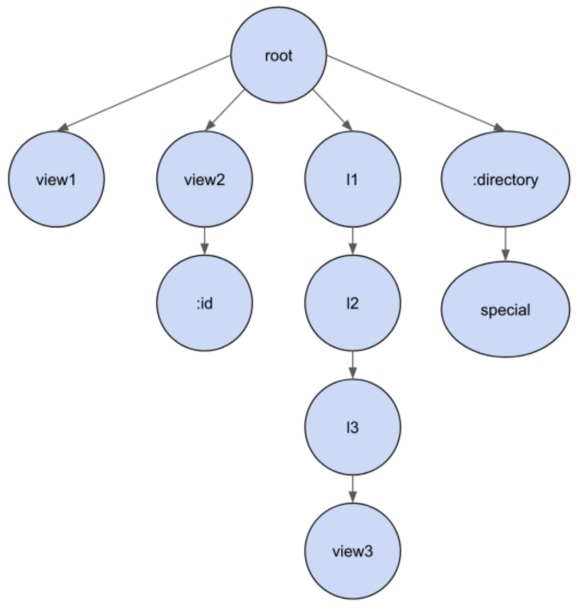
Example router configuration to demonstrate matching

Notice that a route can be broadly defined by the following:

1. Its `path`, or, **how to match against a URL segment**
2. Its `component`, or `children`, or `outlet`, etc. **What to do once it has matched a URL segment.**

There is a nice separation of concerns here. The task of matching a URL to a route is decoupled from the behavior of the route.

The new `ROUTES` array defined above can be represented as a tree:



Tree of ROUTES. Nodes display their path properties

It's no coincidence that both the objects in the `ROUTES` array, and the URL are represented as trees. Since the configuration objects in the `ROUTES` array form a tree of router states, and since the URL is just a serialization of a router state, the URL is also a tree. Matching any URL to a router state is nothing more than matching the segments of a UrlTree against some path in `ROUTES`.

Internally, Angular uses an instance of the Recognizer class to perform url-to-path matching.

The router will use `DefaultUrlMatcher`. An excerpt, the DefaultUrlMatcher's algorithm is shown below.

```
// Check each config part against the actual URL
for (let index = 0; index < parts.length; index++) {
  const part = parts[index];
  const segment = segments[index];
  const isParameter = part.startsWith(':');
  if (isParameter) {
    posParams[part.substring(1)] = segment;
  } else if (part !== segment.path) {
    // The actual URL part does not match the config,
    return null;
```

An excerpt of the matching algorithm — don't sweat the details too much

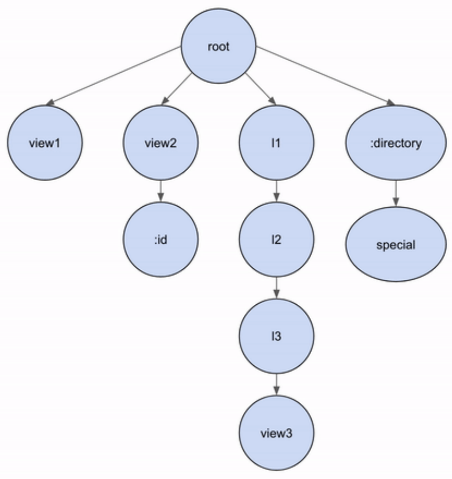
When trying to match a URL to a route, the router looks at the **unmatched segments of the URL** and tries to find a `path` that will match, or *consume* a segment. **Think of it as a depth first search through the route configurations defined in the `ROUTES` array**.

Once all segments of the URL have been consumed, we say that a match has occurred. For example, given the configuration above, the URL `l1/l2/l3/view3` will be consumed as follows:

1. The router starts stepping through the entries in `ROUTES` . The first entry has `path: 'view1'` . `view1` does not equal `l1` , so it moves on. `view2` does not equal `l1` , so it moves on. `l1` equals `l1` , so the url segment `l1` has now been matched or consumed.

2. Since the URL has not been fully consumed yet (there's still `l2/l3/view3`), the router will recurse down the `children` of `{ path: 'l1' }` .

3. It will eventually consume the remaining segments, since `l2` equals `l2` , `l3` equals `l3` , and `view3` equals `view3` . So `View3Component` will be displayed in the primary router outlet.



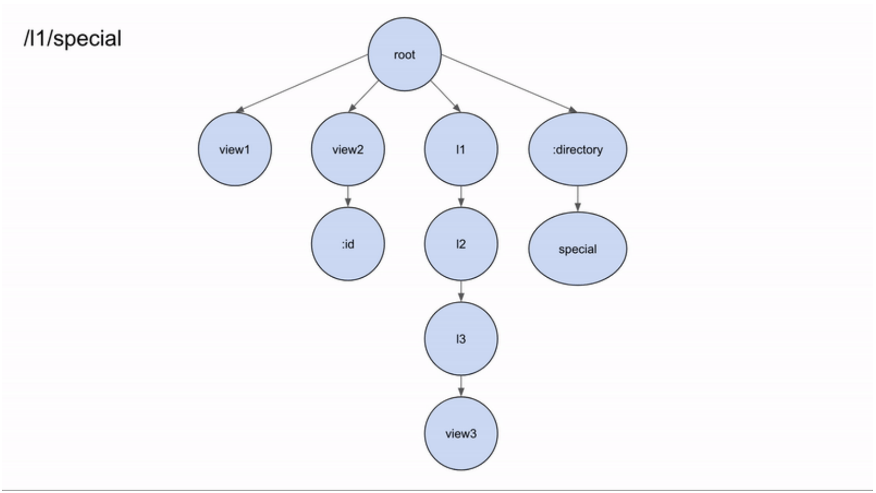
Matching the url '/l1/l2/l3/view3'

Sometimes, the router will have to backtrack when matching. For instance, consider the path `l1/special` . In this case:

1. The router loops through its `ROUTES` . `view1` does not equal `l1` , so it moves on. `view2` does not equal `l1` , so it moves on.

`l1` equals `l1`, so the url segment `l1` has now been matched or consumed.

2. Since the URL has not been fully consumed yet (there’s still the segment `special`), the router will recurse down the children of `{path: ‘l1'}`.
3. In this case, it will **not match any child path**, since the only path is `l2`, and `l2` does not match `special`. The router will **back up a level in the configuration** and see if anything else would have matched `l1`.
4. In this case, the router will see `:directory` as the next possible path. Paths which are prefixed with a colon will match anything, so `:directory` will match `l1`.
5. Since the URL has not been fully consumed yet (there’s still `special`), the router will recurse down this path’s `children`.
6. `path: 'special'` will match `special` so the URL has now been fully consumed, and `SpecialComponent` will be displayed in the primary outlet.



An example of backtracking

The router takes a **depth-first approach** to matching URL segments with paths. **This means that the first path of routes to fully consume a URL wins.** You must take care with how you structure your router configuration, as there is no notion of specificity or importance amongst routes—the first match always wins. Order matters.

In URLs which have secondary outlets, such as:

```
'/users/1/notes/42(sidebar:secondary1)?lang=en#line99'
```

The outlets are routed independently of each other, so navigating from `secondary1` to `secondary2` would not affect the part of the URL associated with the primary outlet, `/users/1/notes/42` . You can see this in action in this stackblitz.

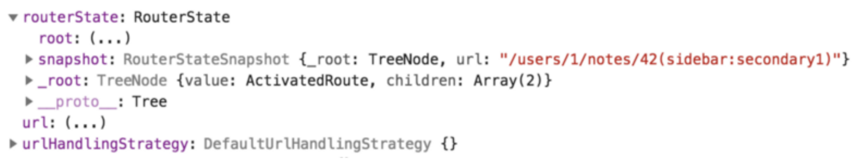
· · ·

Router States

The result of successfully matching a URL is that some set of components will be routed to, and rendered on screen through the use of router-outlet directives. **But there is also a useful side effect to this operation—the creation of RouterState and state snapshot objects.**

After routing has occurred, we might want to access information about the URL and the set of components that were routed to—known as the current **router state**. The term *router state* is somewhat overloaded, as the objects inside of the `ROUTES` array are said to define the possible *router states* for an application, that is, the sets of components that can be routed to for a particular URL. However, `routerState` is also a property on the Router Service. In this section, *router state* will refer to the `routerState` property on the Router service, which lets us access information about what URL and components are currently routed to.

For instance, we may need to access query parameters, or other data encoded in the URL from within a component or service. The Router service provides a property called `routerState: RouterState`, which lets you access everything about the current state of the router. The `routerState` has two properties of interest to us; `snapshot` , and `root` .



```
▼ routerState: RouterState
    root: (...)
  ▶ snapshot: RouterStateSnapshot {_root: TreeNode, url: "/users/1/notes/42(sidebar:secondary1)"}
  ▶ _root: TreeNode {value: ActivatedRoute, children: Array(2)}
  ▶ __proto__: Tree
  url: (...)
▶ urlHandlingStrategy: DefaultUrlHandlingStrategy {}
```

A snippet of the Router service

Both are trees representing the current *router state* (components that have been routed to, along with URL segments and parameters), but they differ in one key way, `snapshot` is a tree of ActivatedRouteSnapshot objects—**which are static,** `root` is a tree of ActivatedRoute objects—**which are dynamic.**

Sometimes a snapshot of state is enough, and other times, you want to subscribe to an observable to listen for state changes.

For example, when a URL changes from `/users/15/notes/41` to `/users/15/notes/42` , the router will recognize that only the `:noteid` parameter has changed, so it will simply reuse the current set of components on screen, and will not create a new tree of snapshots. In this case, it's better to use the observable approach, if you know route parameters are likely to change.

The ActivatedRoutes are constructed inside of a function called `processSegmentAgainstRoute` , which is called during the matching phase of navigation, when a URL segment is being matched against a route's path:

```
const result: MatchResult = match(rawSegment, route, se
consumedSegments = result.consumedSegments;
rawSlicedSegments = segments.slice(result.lastChild);

snapshot = new ActivatedRouteSnapshot(
    consumedSegments, result.parameters, Object.freeze(
    this.urlTree.fragment !, getData(route), outlet, ro
```

Note that a routerState can have multiple trees of ActivatedRoutes at once—one for each outlet.

When we say that the URL is just a serialization of the router state, this is what is meant.

. . .

We've seen how a URL is represented as a UrlTree, and how the router matches that URL to a route, and creates a tree of ActivatedRoutes. In the next article, we'll see the mechanics of how the router actually renders the appropriate components on screen, and handles any route guards or route resolvers along the way. Thanks for reading, and stay tuned!

Special thanks to Jason Aden, Max Wizard Koretskyi, Alex Okrushko, and Tim Deschryver for all of their help with this article!