Angular Router Series: Pillar 2— Understanding The Router's Navigation Cycle

Nate Lapinski Follow

Oct 10, 2018 · 8 min read

A Rainy Night in Tokyo Bay

Routing is essential for any frontend framework or library. It makes single page applications possible by letting us load the application once, and display different content to the user through client-side routing.

It's easy enough to get started with Angular's router, but have you ever wondered what really happens when you click a link in an Angular application? In this article, we'll answer this question and

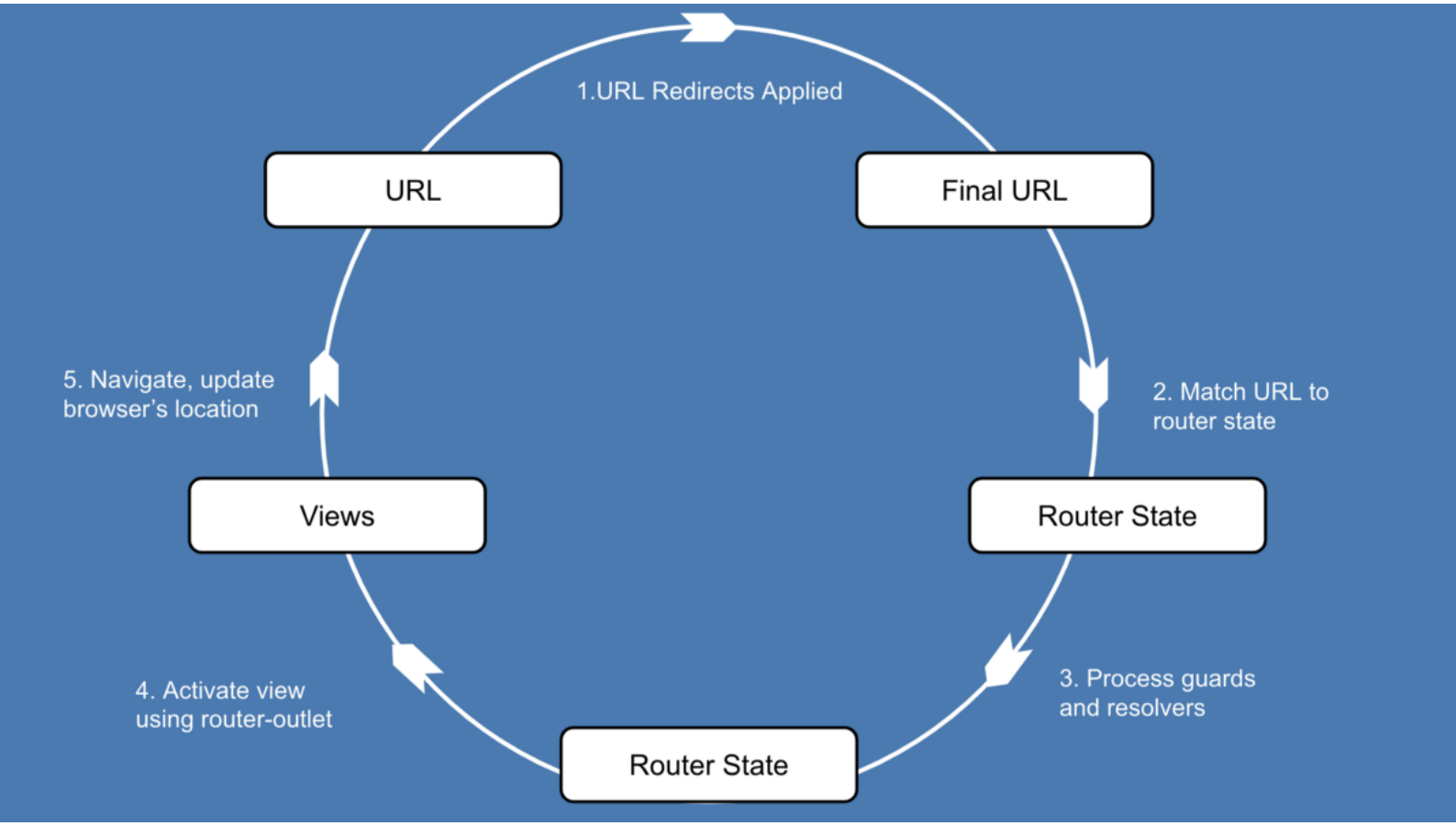
more. **A lot of insight into Angular can be gained through an understanding of the router's navigation cycle**.

By the end of this article, you will understand the three questions the router must ask itself while navigating:

1. **Given a URL, which set of components should I navigate to?**
2. **Can I navigate to those components?**
3. **Should I prefetch any data for those components?**

Along the way, we'll see the following, in detail.

- The entire routing process, from start to finish
- How the router builds and uses a tree of ActivatedRouteSnapshot objects during and after navigation.
- Rendering content using `<router-outlet>` directives.



The Router's navigation lifecycle

Let's take a little journey, and see exactly what happens when we route in an Angular application.

• • •

Navigation

In Angular, we build single page applications. This means that we don't actually *load* a new page from a server whenever the URL changes. Instead, the router provides location-based navigation in the browser, which is essential for single page applications. It is what allows us to change the content the user sees, as well as the URL, all without refreshing the page.

Navigation (routing) occurs whenever the URL changes. We need a way to navigate between views in our application, and standard anchor tags with `href` will not work, since those would trigger full page reloads. This is why Angular provides us with the `[routerLink]` directive. When clicked, it tells the router to update the URL and render content using `<router-outlet>` directives, without reloading the page.

```
<!-- without a routerLink -->
<a href='localhost:4200/users'>Users</a>  <!-- not what
<!-- with a routerLink -->
<a [routerLink]="['/users']">Users</a>  <!-- router wil
```

For every navigation, a series of steps takes place before the router renders the new components on the screen. This is known as the router navigation lifecycle.

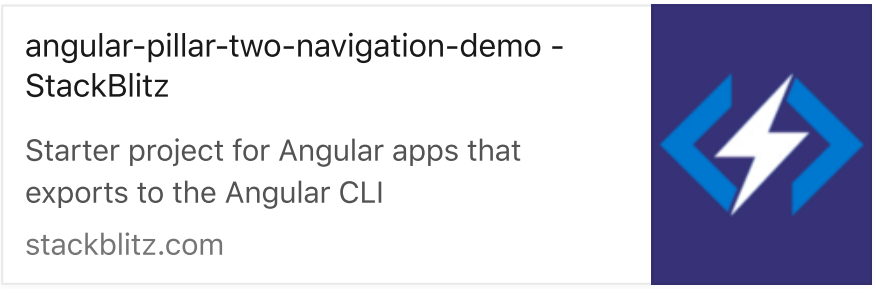
The output of a successful navigation is that new components will be rendered using `<router-outlet>`, and a tree of ActivatedRoute data structures will be created as a queryable record of the navigation. If you'd like to know more about activated routes and router states, I've written about them here. For our purposes, just know that activated routes are used by both the router and the developer to extract information about the navigation, such as query parameters and components.

Example Application

We'll use a very simple application as a running example. Here is the router configuration.

```
const ROUTES = [
  { path: 'users',
    component: UsersComponent,
    canActivate: [CanActivateGuard],
    resolve: {
      users: UserResolver
    }
```

The sample application can be found here:



The application consists of one route, `/users`, which checks a query parameter to see if a user is logged in (`login=1`), and then displays a list of usernames that it retrieves from a mock API service.

The specifics of the application aren't important. We just need an example to see the navigation cycle.

Navigation Cycle and Router Events

A great way to see the navigation cycle is by subscribing to the Router service's `events` observable:

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

During development, you can also pass along an option of `enableTracing: true` in the router configuration.

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

In the developer console, we can see the events emitted during a navigation to the `/users` route:

Navigation events. We are also passing along the login query parameter. More on this in the section on route guards.

These events are very useful for studying or debugging the router. You could easily tap into them to display a loading message during navigation as well.

```
ngOnInit() {
  this.router.events.subscribe(evt => {
    if (evt instanceof NavigationStart) {
      this.message = 'Loading...';
      this.displayMessage = true;
    }
    if (evt instanceof NavigationEnd) this.displayMessa
```

excerpt from app.component.ts. Displays a loading message at the start of navigation, and clears the message once navigation has ended

Let's run through a navigation to `/users` .

Navigation Start

events: NavigationStart

In our sample application, the user starts by clicking on the following link:

```
1   <a [routerLink]="['/users']" [queryParams]="{'login': '
```

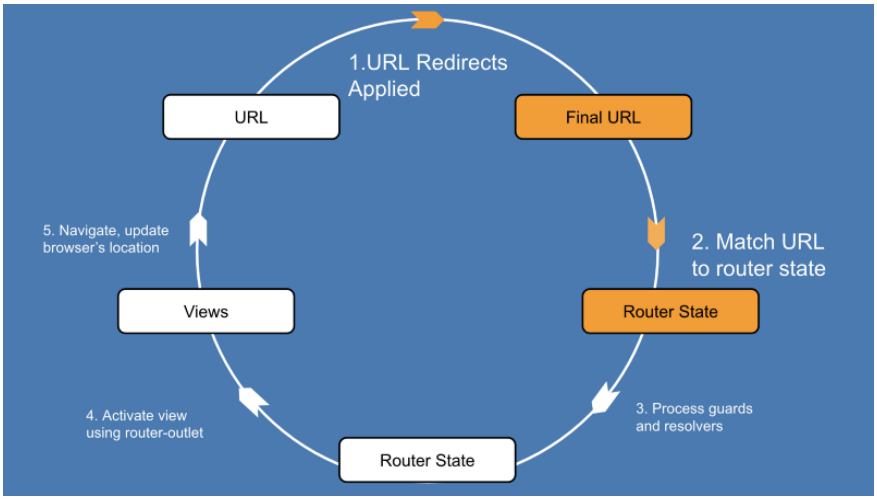
Navigate to /users, and pass along the query parameters login=1 (see the section on router guards)

Whenever the router detects a click on a router link directive, it starts the navigation cycle. There are imperative means of starting a navigation as well, such as the Router Service's `navigate` and `navigateByUrl` methods.

Previously, there could be multiple navigations running simultaneously (hence the need for a navigation id), but with this change, there can be only one navigation at a time.

URL Matching, and Redirects

events: RoutesRecognized



redirects and matching are steps one and two of the cycle

The router starts by doing a depth-first search through the array of router configurations (`ROUTES` in our example), and trying to match the URL `/users` to one of the `path` properties in the router configurations, while applying any redirects along the way. If you'd like to know about this process in detail, I've written about it here.

In our case, there are no redirects to worry about, and the URL `/users` will match the following configuration in `ROUTES` :

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

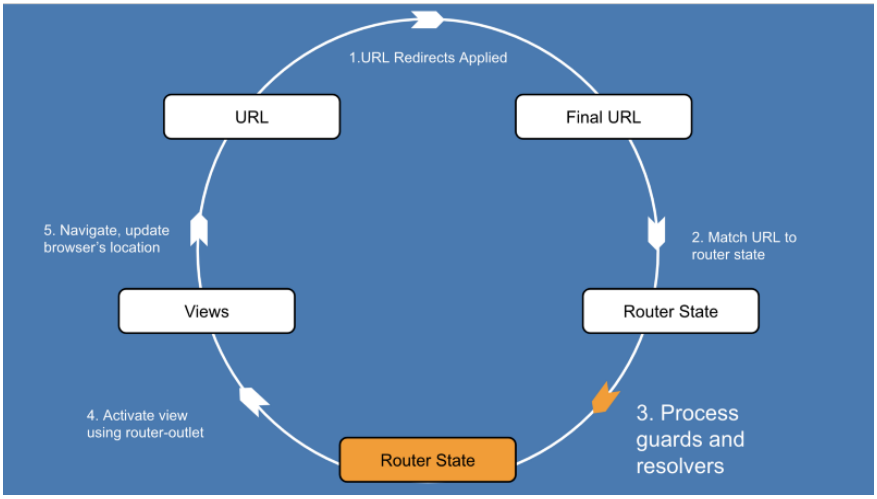
If the matched path requires a lazy loaded module, it will be loaded at this point.

The router emits a `RoutesRecognized` event to signal that it has found a match for the URL, and a component to navigate to (UsersComponent). This answers the router's first question, "What should I navigate to?". But not so fast, the router has to make sure that it is allowed to navigate to this new component.

Enter route guards.

Route Guards

events: GuardsCheckStart, GuardsCheckEnd



Route guards are boolean functions that the router uses to determine if it can perform a navigation. **As developers, we use guards to control whether a navigation can occur or not.** In our sample application, we use a canActivate guard to check if a user is logged in by specifying it in the route configuration.

```
{ path: 'users', ..., canActivate: [CanActivateGuard] }
```

The guard function is shown below.

```
canActivate(route: ActivatedRouteSnapshot, state: Route
    return this.auth.isAuthorized(route.queryParams.log
}
```

isAuthorized will return true when the query parameter login=1

This guard passes the `login` query parameter to an `auth` service (`auth.service.ts` in the example app).

If the call to `isAuthorized(route.queryParams.login)` returns `true`, the guard passes successfully. Otherwise, the guard fails, and the router emits a `NavigationCancel` event, and aborts the entire navigation.

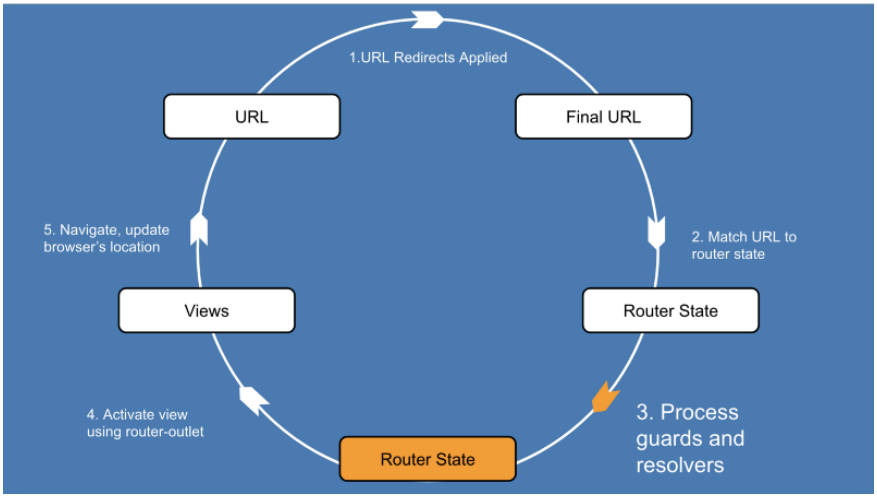
Other guards include canLoad (should a module be lazily-loaded or not), canActivateChild, and canDeactivate (which is useful for preventing a user from navigating away from a page, for instance, when filling out a form).

Guards are similar to services, they are registered as providers and are injectable. The router will run guards every time there is a change to the URL.

The canActivate guard is run before any data is fetched for the route, since there is no reason to fetch data for a route that shouldn't be activated. Once the guard has passed, the router has answered its second question, "Should I perform this navigation?". The router can now prefetch any data using route resolvers.

Route Resolvers

events: ResolveStart, ResolveEnd



Route resolvers are functions that we can use to prefetch data during navigation, before the router has rendered anything. Similar to guards, we specify a resolver in the route configuration, using the `resolve` property:

```
{ path: 'users', ..., resolve: { users: UserResolver } }
```

```
export class UserResolver implements Resolve<Observable
  constructor(private userService: MockUserDataService)

  resolve(): Observable<any> {
    return this.userService.getUsers();
  }
```

Once the router has matched a URL to a path, and all guards have passed, it will call the `resolve` method defined in the `UserResolver` class to fetch data. The router stores the results on the `ActivatedRoute` service's `data` object, under the key `users`. This information can be read by subscribing to the ActivatedRoute service's `data` observable.

```
activatedRouteService.data.subscribe(data => data.users);
```

The ActivatedRoute service is used inside of the UsersComponent, to retrieve data from the resolver.

```
export class UsersComponent implements OnInit {
  public users = [];

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.data.subscribe(data => this.users = data
```

Resolvers let us prefetch component data during routing. This technique can be used to avoid displaying partially loaded templates to the user by prefetching any data. Remember, a component's template will be visible to the user during `OnInit`, so fetching any data that needs to be rendered in that lifecycle hook can lead to a partial page load.

However, it is often better to just let a page partially load. When done well, it will increase the user's perceived performance of the site. The decision of whether or not to prefetch data is up to you, but it's usually best to have a partial page load with a nice loading animation instead of using resolvers.

Internally, the router has a runResolve method which will execute the resolver, and store its results on the ActivatedRoute snapshot.

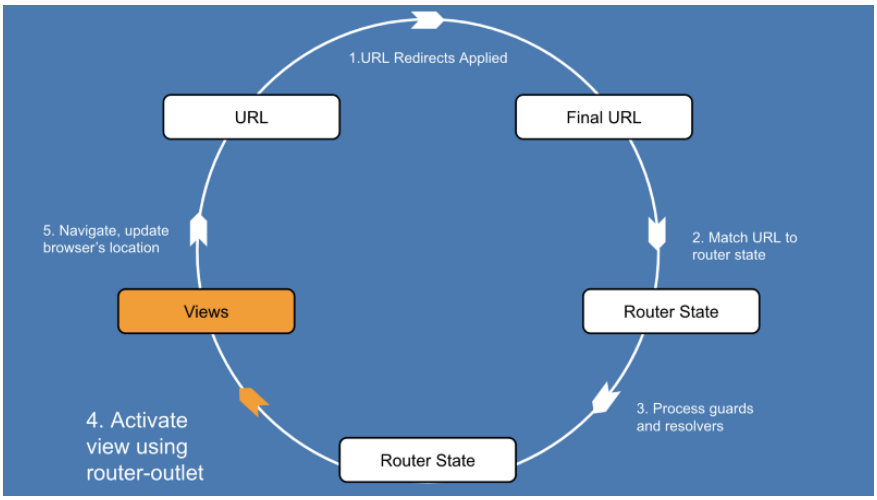
```
future.data = {...future.data,
               ...inheritedParamsDataResolve(future, pa
```

"future" is as ActivatedRouteSnapshot

Once the router has processed all resolvers, the next step is to start rendering components using the appropriate router outlets.

Activating Routes

events: ActivationStart, ActivationEnd, ChildActivationStart, ChildActivationEnd



Now it's time to activate the components, and display them using a `<router-outlet>` . The router can extract the information it needs about the component from the tree of ActivatedRouteSnapshots that it built during the previous steps of the navigation cycle.

```
▼ ActivationStart {snapshot: ActivatedRouteSnapshot}
  ▼ snapshot: ActivatedRouteSnapshot
      children: (...)
    ▶ component: ƒ UsersComponent(route)
    ▼ data:
      ▶ users: (4) ["Freddie", "Brian", "Roger", "John"]
      ▶ __proto__: Object
      firstChild: (...)
      fragment: undefined
      outlet: "primary"
      paramMap: (...)
    ▶ params: {}
      parent: (...)
      pathFromRoot: (...)
      queryParamMap: (...)
    ▶ queryParams: {login: "1"}
      root: (...)
    ▶ routeConfig: {path: "users", component: ƒ, canActivate: Array(1), resolve: {…}}
    ▶ url: [UrlSegment]
      _lastPathIndex: 0
    ▶ _resolve: {users: ƒ}
    ▶ _resolvedData: {users: Array(4)}
    ▶ _routerState: RouterStateSnapshot {_root: TreeNode, url: "/users?login=1"}
    ▶ _urlSegment: UrlSegmentGroup {segments: Array(1), children: {…}, parent: UrlSegmentGroup}
    ▶ __proto__: Object
  ▶ __proto__: Object
```

The component property tells the router to create and activate an instance of UsersComponent. We can also see the users we fetched under the data.users property.

If you are unfamiliar with the process of creating dynamic components in Angular, there are great explanations here and here. All of the magic happens within the router's `activateWith` function:

```
1  activateWith(activatedRoute: ActivatedRoute, resolver:
2    if (this.isActivated) {
3      throw new Error('Cannot activate an already activa
4    }
5    this._activatedRoute = activatedRoute;
6    const snapshot = activatedRoute._futureSnapshot;
7    const component = <any>snapshot.routeConfig !.compon
8    resolver = resolver || this.resolver;
9    const factory = resolver.resolveComponentFactory(com
10   const childContexts = this.parentContexts.getOrCreat
11   const injector = new OutletInjector(activatedRoute,
12   this.activated = this.location.createComponent(facto
```

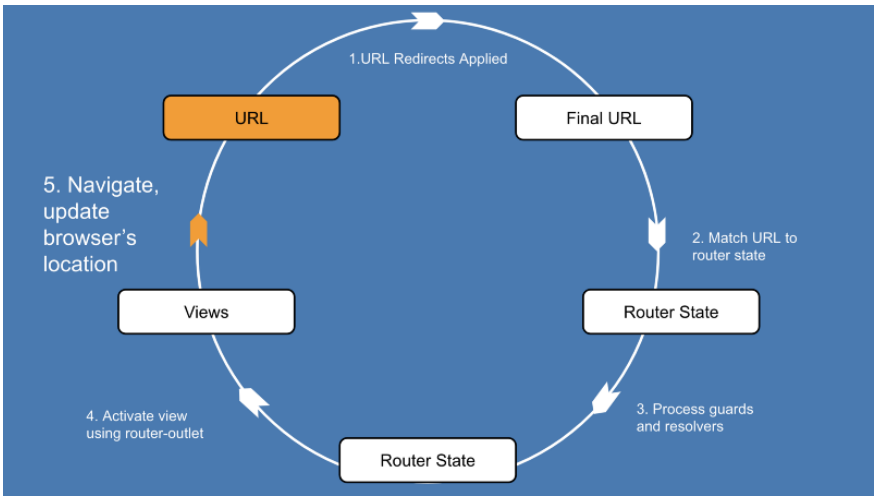
excerpt from router_outlet.ts

Don't stress the details, I'll summarize the main points of the code here:

- On Line 9, A ComponentFactoryResolver is used to create an instance of the `UsersComponent`. The router pulls this information off of the ActivatedRouteSnapshot in line 7.
- On line 12, the component is actually created. `location` is a ViewContainerRef for the `<router-outlet>` that is being targeted. If you've ever wondered why the rendered content is placed as a sibling to the `<router-outlet>` as opposed to inside of it, the details can be found by following the details inside of `createComponent`.
- After the component is created and activated, `activateChildRoutes` (not shown) is called. This is done to account for any nested `<router-outlet>`, known as child routes.

The router will render a component on the screen. If the rendered component has any nested `<router-outlet>` elements, the router will go through and render those as well.

Updating the URL



The last step in the navigation cycle is to update the URL to `/users`.

```
1  private updateTargetUrlAndHref(): void {
2    this.href = this.locationStrategy.prepareExternalUrl(
3  }
```

The router is now ready to listen for another URL change, and start the cycle all over again.

• • •

In the final installment of this series, we'll take an in-depth look at the router's mechanism for lazy-loading modules. Thanks for reading, and stay tuned!