ANGULAR ROUTER

VICTOR SAVKIN

Angular Router

The Complete Authoritative Reference

Victor Savkin

This book is for sale at http://leanpub.com/router

This version was published on 2018-12-05

Contents

Introduction

About the Author

Victor Savkin is an Angular core team member. He is one of the main contributors to Angular and is the main contributor to the Angular router. Apart from building Angular, Victor also likes to toy with eclectic programming technologies and obsesses over fonts and keyboards.

Nrwl.io - Angular consulting for enterprise customers, from core team members

Victor is co-founder of Nrwl, a company providing Angular consulting for enterprise customers, from core team members. Visit nrwl.io[1] for more information.

What is this book about?

Managing state transitions is one of the hardest parts of building applications. This is especially true on the web, where you also need to ensure that the state is reflected in the URL. In addition, we often want to split applications into multiple bundles and load them on demand. Doing this transparently is not trivial.

The Angular router solves these problems. Using the router, you can declaratively specify application states, manage state transitions while taking care of the URL, and load bundles on demand.

[1] http://nrwl.io

In this book I will talk about the router's mental model, its API, and the design principles behind it. To make one thing clear: this book is not about Angular. There is a lot of information about the framework available online. So if you want to get familiar with the framework first, I would recommend the following resources:

1. angular.io[2]. It's the best place to get started.
2. egghead.io[3]. The Angular Fundamentals on egghead.io is an excellent way to get started for those who learn better by watching.
3. vsavkin.com[4]. My blog contains many articles where I write about Angular in depth.

Therefore in this book I assume you are familiar with Angular, and so I won't talk about dependency injection, components, or bindings. I will only talk about the router.

Why would you read this book?

Why would you read this book if you can find the information about the router online? Well, there are a couple of reasons.

First, this book will be kept up to date with the future releases of Angular and the router. All the information will remain accurate and the examples will remain working. This book will never be stale.

Second, the book goes far beyond a how-to-get-started guide. It is a complete description of the Angular router. The mental model, design constraints, and the subtleties of the API-everything is covered. Understanding these will give you deep insights into why the router works the way it does and will make you an Angular router expert.

Let's get started!

[2] http://angular.io
[3] https://egghead.io/courses/angular-2-fundamentals
[4] http://vsavkin.com

Example

For all the examples in this book we will use MailApp, which is an application akin to Inbox or Gmail. At launch, the application displays a list of conversations, which we can browse through. Once we click on a conversation, we can see all its messages. We can also compose a new message, or view an existing message in a popup.

You can find the source code of the MailApp application here: MailApp[5].

[5] https://github.com/vsavkin/router_mailapp

Chapter 1: What Do Routers Do?

Before we jump into the specifics of the Angular router, let's talk about what routers do in general.

As you know, an Angular application is a tree of components. Some of these components are reusable UI components (e.g., list, table), and some are application components, which represent screens or some logical parts of the application. The router cares about application components, or, to be more specific, about their arrangements. Let's call such component arrangements router states. So a router state defines what is visible on the screen.

A router state is an arrangement of application components that defines what is visible on the screen.

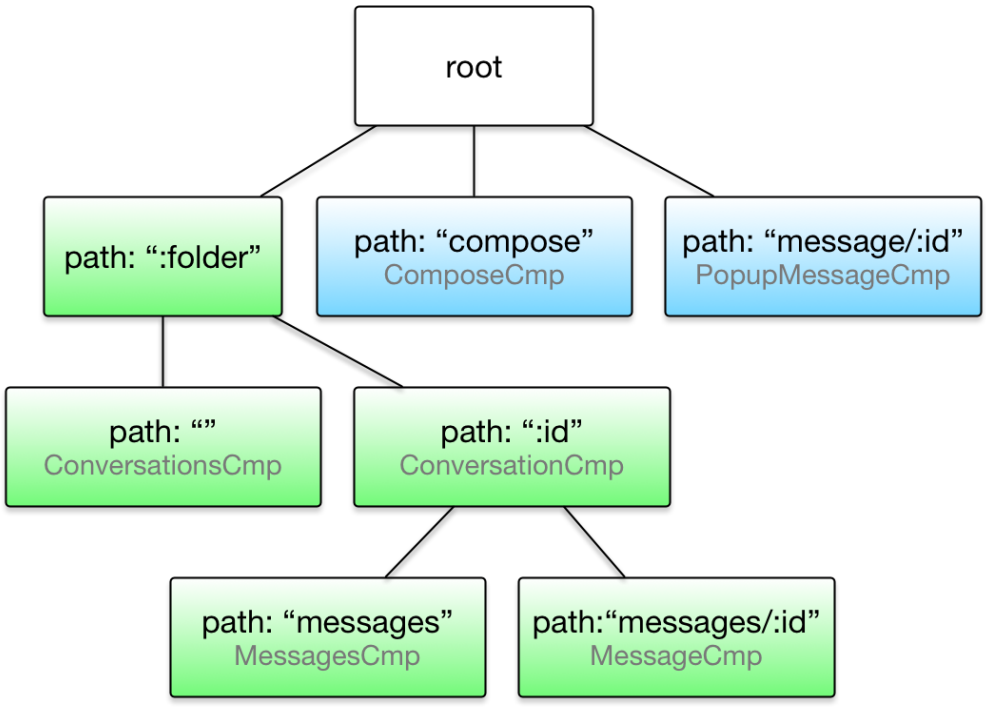
Router Configuration

The router configuration defines all the potential router states of the application. Let's look at an example.

```
[
  {
    path: ':folder',
    children: [
      {
        path: '',
        component: ConversationsCmp
      },
      {
        path: ':id',
        component: ConversationCmp,
        children: [
          { path: 'messages', component: MessagesCmp },
          { path: 'messages/:id', component: MessageCmp }
        ]
      }
    ]
  },
  {
    path: 'compose',
```

```
    component: ComposeCmp,
    outlet: 'popup'
  },
  {
    path: 'message/:id',
    component: PopupMessageCmp,
    outlet: 'popup'
  }
]
```

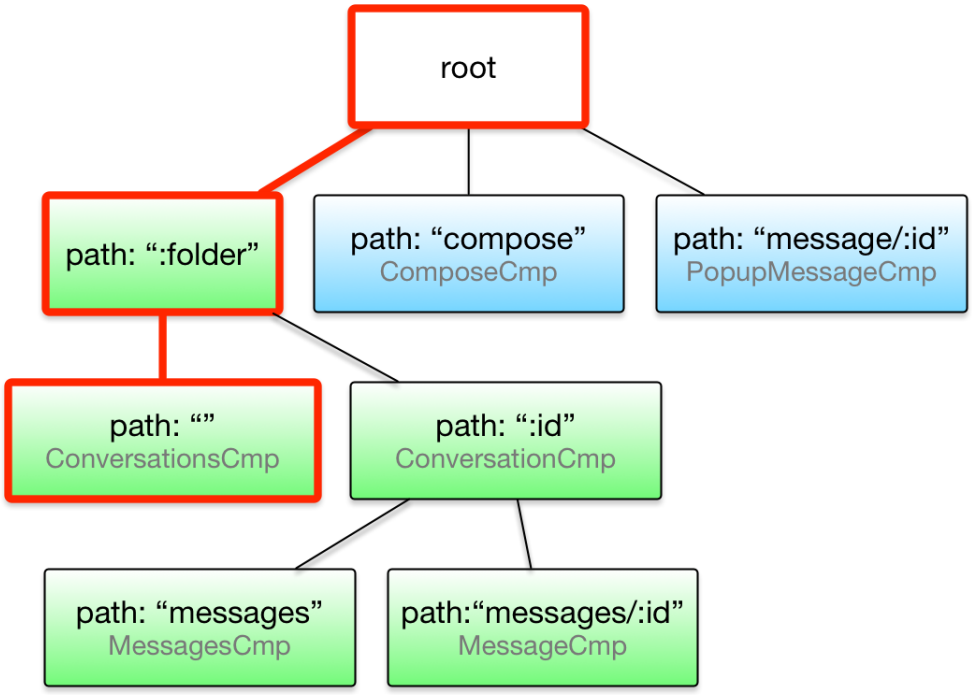
Don't worry about understanding all the details. I will cover them in later chapters. For now, let's depict the configuration as follows:



As you can see the router configuration is a tree, with every node representing a route. Some nodes have components associated with them, some do not. We also use color to designate different outlets, where an outlet is a location in the component tree where a component is placed.

Router State

A router state is a subtree of the configuration tree. For instance, the example below has `ConversationsCmp` activated. We say *activated* instead of *instantiated* as a component can be instantiated only once but activated multiple times (any time its route's parameters change).



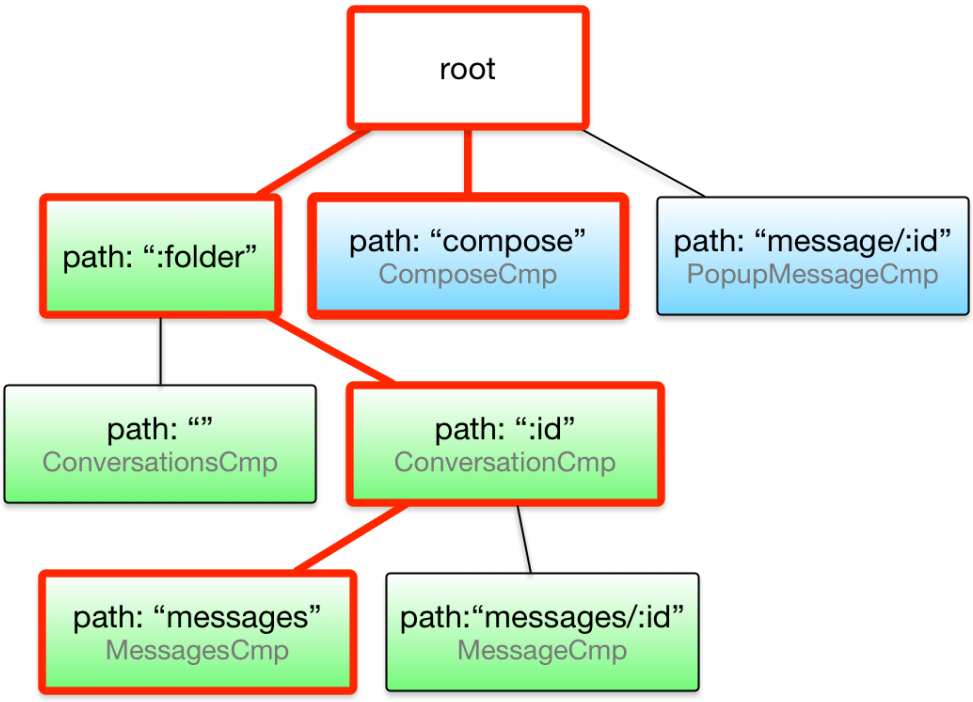
Not all subtrees of the configuration tree are valid router states. If a node has multiple children of the same color, i.e., of the same outlet name, only one of them can be active at a time. For instance, `ComposeCmp` and `PopupMessageCmp` cannot be displayed together, but `ConversationsCmp` and `PopupMessageCmp` can. Stands to reason, an outlet is nothing but a location in the DOM where a component is placed. So we cannot place more than one component into the same location at the same time.

Navigation

The router's primary job is to manage navigation between states, which includes updating the component tree.

Navigation is the act of transitioning from one router state to another.

To see how it works, let's look at the following example. Say we perform a navigation from the state above to this one:



Because `ConversationsCmp` is no longer active, the router will remove it. Then, it will instantiate `ConversationCmp` with `MessagesCmp` in it, with `ComposeCmp` displayed as a popup.

Summary

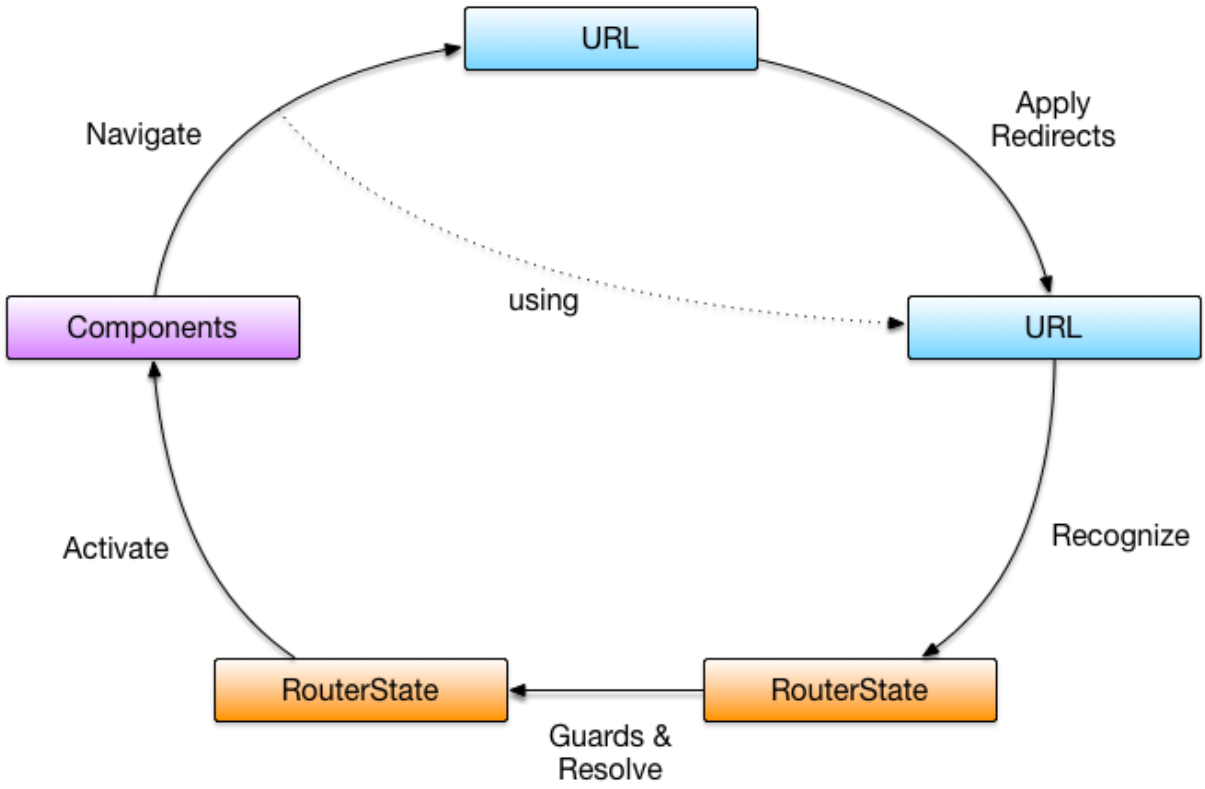
That's it. The router simply allows us to express all the potential states which our application can be in, and provides a mechanism for navigating from one state to another. The devil, of course, is in the implementation details, but understanding this mental model is crucial for understanding the implementation.

Isn't it all about the URL?

The URL bar provides a huge advantage for web applications over native ones. It allows us to reference states, bookmark them, and share them with our friends. In a well-behaved web application, any application state transition results in a URL change, and any URL change results in a state transition. In other words, a URL is nothing but a serialized router state. The Angular router takes care of managing the URL to make sure that it is always in-sync with the router state.

Chapter 2: Overview

Now that we have learned what routers do in general, it is time to talk about the Angular router.



The Angular router takes a URL, then:

1. Applies redirects
2. Recognizes router states
3. Runs guards and resolves data,
4. Activates all the needed components
5. Manages navigation

Most of it happens behind the scenes, and, usually, we do not need to worry about it. But remember, the purpose of this book is to teach you how to configure the router to handle any crazy requirement your application might have. So let's get on it!

URL Format

Since I will use a lot of URLs in the examples below, let's quickly look at the URL format.

```
/inbox/33(popup:compose)
```

```
/inbox/33;open=true/messages/44
```

As you can see, the router uses parentheses to serialize secondary segments (e.g., popup:compose), the colon syntax to specify the outlet, and the ';parameter=value' syntax (e.g., open=true) to specify route specific parameters.

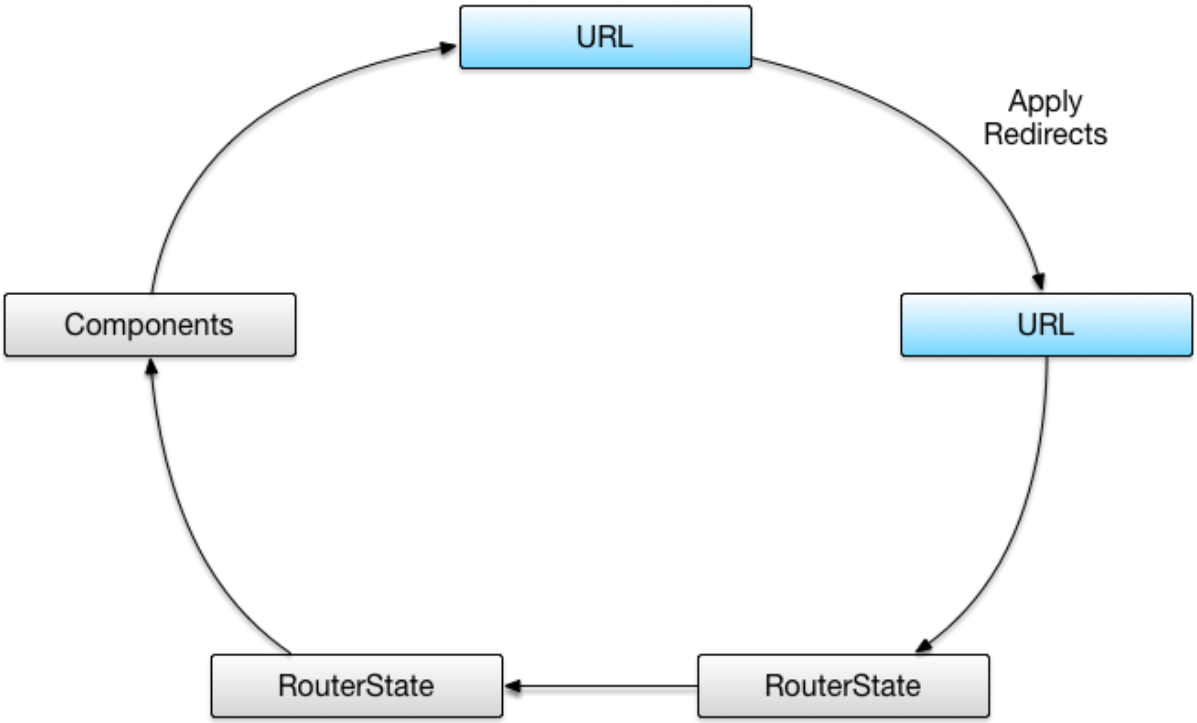
* * *

In the examples below we assume that we have given the following configuration to the router, and we are navigating to `'/inbox/33/messages/44'`.

```
[
  { path: '', pathMatch: 'full', redirectTo: '/inbox' },
  {
    path: ':folder',
    children: [
      {
        path: '',
        component: ConversationsCmp
      },
      {
        path: ':id',
        component: ConversationCmp,
        children: [
          { path: 'messages', component: MessagesCmp },
          { path: 'messages/:id', component: MessageCmp }
        ]
      }
    ]
  },
  {
    path: 'compose',
    component: ComposeCmp,
    outlet: 'popup'
  },
```

```
  {
    path: 'message/:id',
    component: PopupMessageCmp,
    outlet: 'popup'
  }
]
```

Applying Redirects



The router gets a URL from the user, either when she clicks on a link or updates the location bar directly. The first thing that router does with this URL is it will apply any redirects.

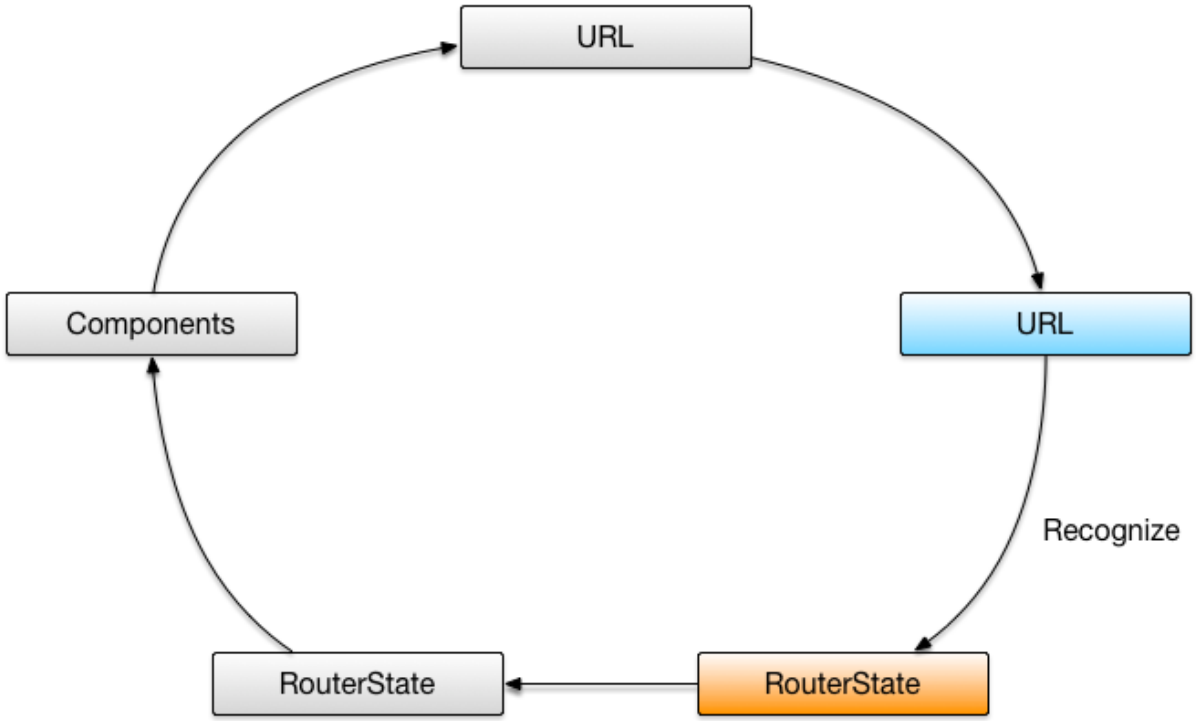
What is a redirect?

A redirect is a substitution of a URL segment. Redirects can either be local or absolute. Local redirects replace a single segment with a different one. Absolute redirects replace the whole URL. Redirects are local unless you prefix the url with a slash.

The provided configuration has only one redirect rule: `{ path: '', pathMatch: 'full', redirectTo: '/inbox' }`, i.e., replace `'/'` with `'/inbox'`. This redirect is absolute because the redirectTo value starts with a slash.

Since we are navigating to `'/inbox/33/messages/44'` and not `'/'`, the router will not apply any redirects, and the URL will stay as is.

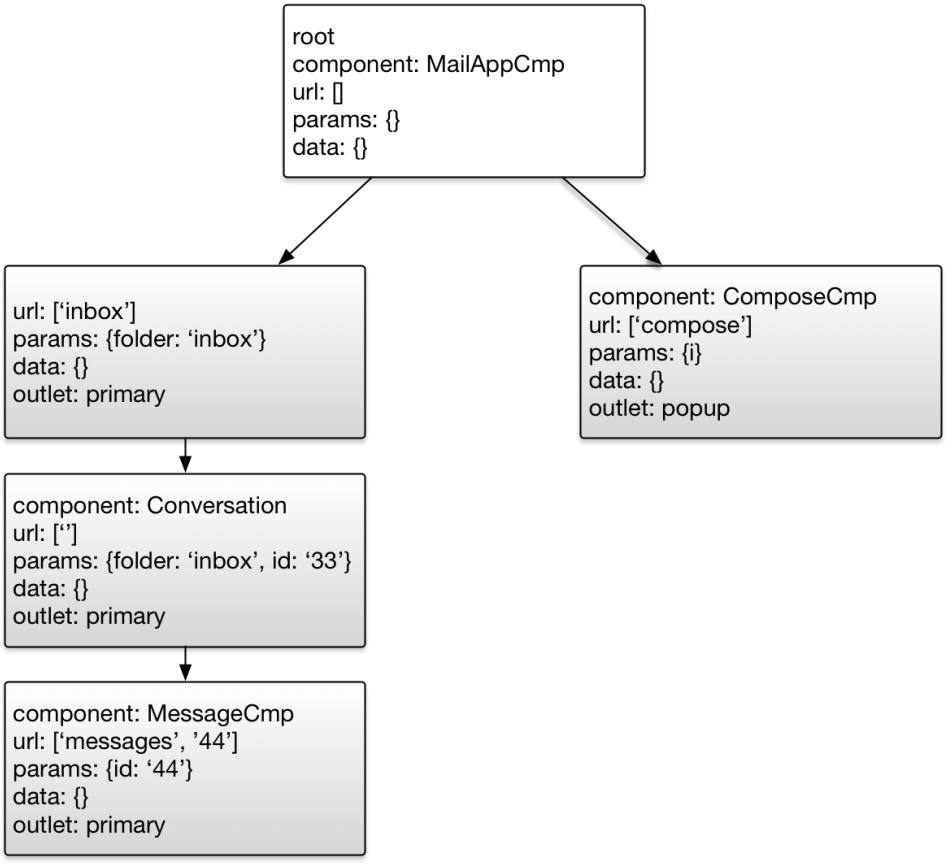
Recognizing States



Next, the router will derive a router state from the URL. To understand how this phase works, we need to learn a bit about how the router matches the URL.

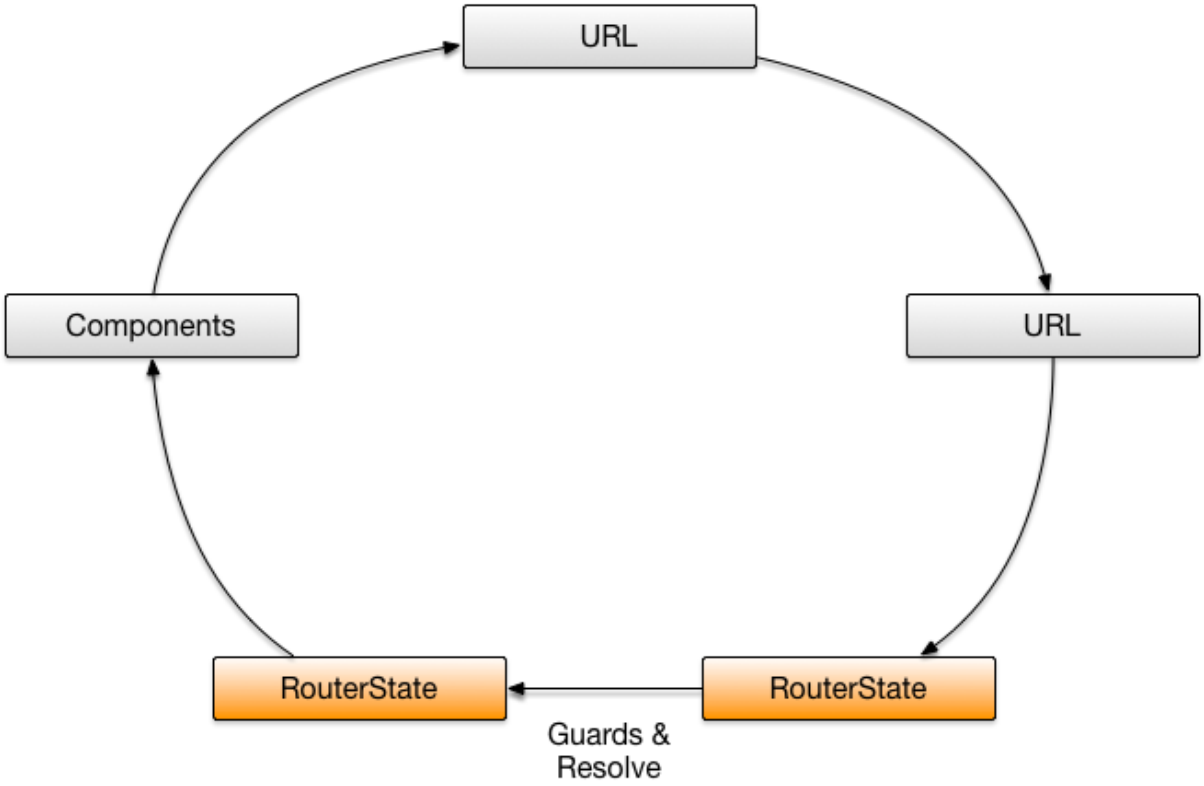
The router goes through the array of routes, one by one, checking if the URL starts with a route's path. Here it will check that `'/inbox/33/messages/44'` starts with ':folder'. It does, since ':folder' is what is called a 'variable segment'. It is a parameter where normally you'd expect to find a constant string. Since it is a variable, virtually any string will match it. In our case 'inbox' will match it. So the router will set the folder parameter to 'inbox', then it will take the children configuration items, the rest of the URL `'33/messages/44'`, and will carry on matching. As a result, the id parameter will be set to '33', and, finally, the `'messages/:id'` route will be matched with the second id parameter set to '44'.

If the taken path through the configuration does not "consume" the whole URL, the router backtracks to try an alternative path. If it is impossible to match the whole URL, the navigation fails. But if it works, the router state representing the future state of the application will be constructed.



A router state consists of activated routes. And each activated route can be associated with a component. Also, note that we always have an activated route associated with the root component of the application.

Running Guards



At this stage we have a future router state. Next, the router will check that transitioning to the new state is permitted. It will do this by running guards. We will cover guards in detail in next chapters. For now, it is sufficient to say that a guard is a function that the router runs to make sure that a navigation to a certain URL is permitted.

Resolving Data

After the router has run the guards, it will resolve the data. To see how it works, let's tweak our configuration from above.

```
[
  {
    path: ':folder',
    children: [
      {
        path: '',
        component: ConversationsCmp,
        resolve: {
          conversations: ConversationsResolver
        }
      }
    ]
  }
]
```

Where `ConversationsResolver` is defined as follows:

```
@Injectable()
class ConversationsResolver implements Resolve<any> {
  constructor(private repo: ConversationsRepo, private currentUser: User) {}

  resolve(route: ActivatedRouteSnapshot, state: RouteStateSnapshot):
      Promise<Conversation[]> {
    return this.repo.fetchAll(route.paramMap.get('folder'), this.currentUser);
  }
}
```

Finally, we need to register `ConversationsResolver` when bootstrapping our application.

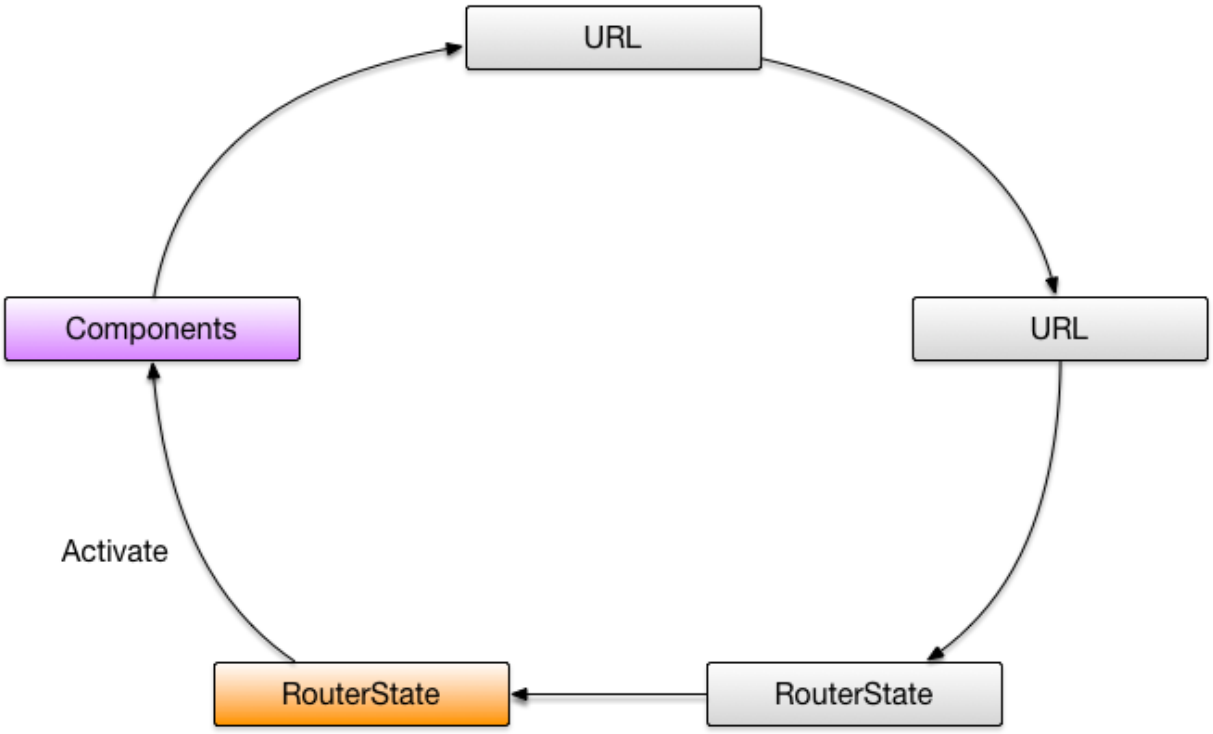
```
@NgModule({
  //...
  providers: [ConversationsResolver],
  bootstrap: [MailAppCmp]
})
class MailModule {
}

platformBrowserDynamic().bootstrapModule(MailModule);
```

Now when navigating to `'/inbox'`, the router will create a router state, with an activated route for the conversations component. That route will have the folder parameter set to 'inbox'. Using this parameter with the current user, we can fetch all the inbox conversations for that user.

We can access the resolved data by injecting the activated route object into the conversations component.

```
@Component({
  template: `
    <conversation *ngFor="let c of conversations | async"></conversation>
  `
})
class ConversationsCmp {
  conversations: Observable<Conversation[]>;
  constructor(route: ActivatedRoute) {
    this.conversations = route.data.pluck('conversations');
  }
}
```

Activating Components



At this point, we have a router state. The router can now activate this state by instantiating all the needed components, and placing them into appropriate router outlets.

To understand how it works, let's take a look at how we use router outlets in a component template.

The root component of the application has two outlets: primary and popup.

```
@Component({
  template: `
    ...
    <router-outlet></router-outlet>

    ...
    <router-outlet name="popup"></router-outlet>
  `
})
class MailAppCmp {
}
```

Other components, such as `ConversationCmp`, have only one.

```
@Component({
  template: `
    ...
    <router-outlet></router-outlet>
    ...
  `
})
class ConversationCmp {
}
```

Now imagine we are navigating to `'/inbox/33/messages/44(popup:compose)'`.

That's what the router will do. First, it will instantiate `ConversationCmp` and place it into the primary outlet of the root component. Then, it will place a new instance of `ComposeCmp` into the 'popup' outlet. Finally, it will instantiate a new instance of `MessageCmp` and place it in the primary outlet of the just created conversation component.

Using Parameters

Often components rely on parameters or resolved data. For instance, the conversation component probably need to access the conversation object. We can get the parameters and the data by injecting `ActivatedRoute`.

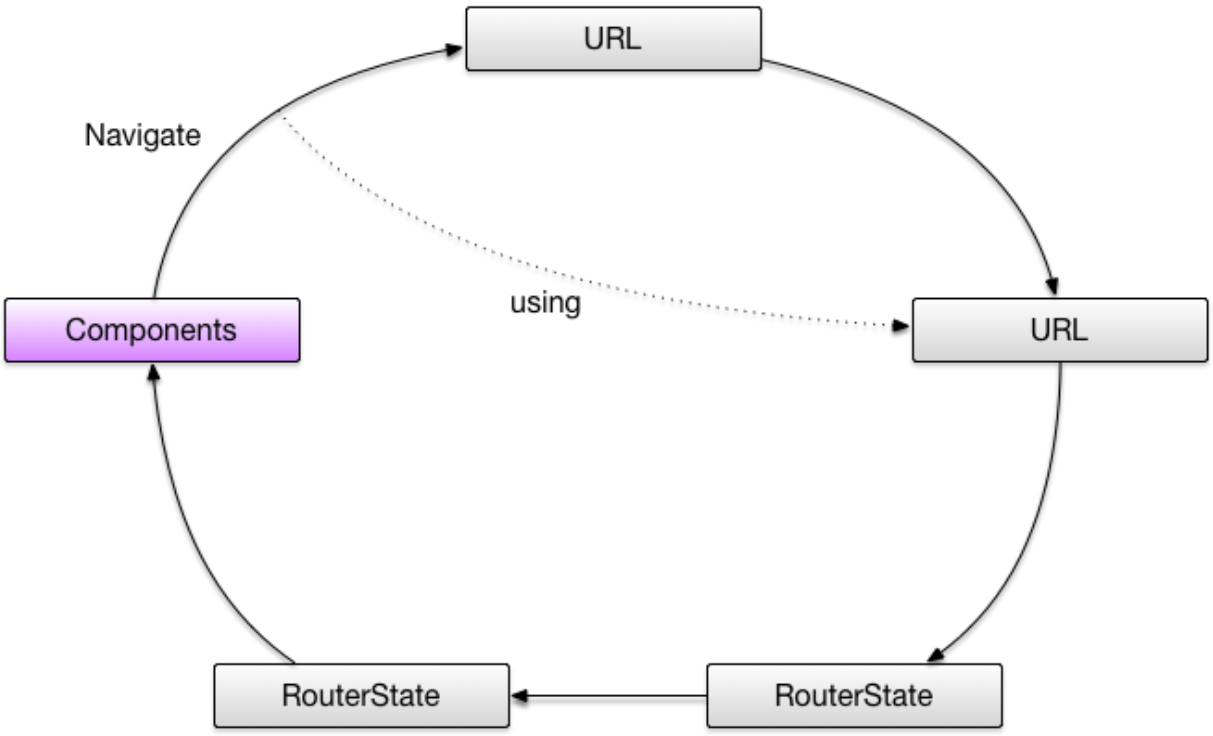
```
@Component({...})
class ConversationCmp {
  conversation: Observable<Conversation>;
  id: Observable<string>;

  constructor(r: ActivatedRoute) {
    // r.data is an observable
    this.conversation = r.data.map(d => d.conversation);

    // r.paramMap is an observable
    this.id = r.paramMap.map(p => p.get('id'));
  }
}
```

If we navigate from `'/inbox/33/messages/44(popup:compose)'` to `'/inbox/34/messages/45(popup:compose)'`, the data observable will emit a new 'map' with the new object, and the conversation component will display the information about Conversation 34.

As you can see the router exposes parameters and data as observables, which is convenient most of the time, but not always. Sometimes what we want is a snapshot of the state that we can examine at once.

```
@Component({...})
class ConversationCmp {
  conversation: Conversation;
  constructor(r: ActivatedRoute) {
    const s: ActivatedRouteSnapshot = r.snapshot;
    this.conversation = s.data['conversation']; // Conversation
  }
}
```

Navigation



So at this point the router has created a router state and instantiated the components. Next, we need to be able to navigate from this router state to another one. There are two ways to accomplish this: imperatively, by calling `router.navigate`, or declaratively, by using the `RouterLink` directive.

Imperative Navigation

To navigate imperatively, inject the `Router` service and call navigate.

```
@Component({...})
class MessageCmp {
  public id: string;
  constructor(private route: ActivatedRoute, private router: Router) {
    route.paramMap.subscribe(_ => this.id = _.get('id'));
  }

  openPopup(e) {
    this.router.navigate([{outlets: {popup: ['message', this.id]}}]).then(_ => {
      // navigation is done
```

```
    });
  }
}
```

RouterLink

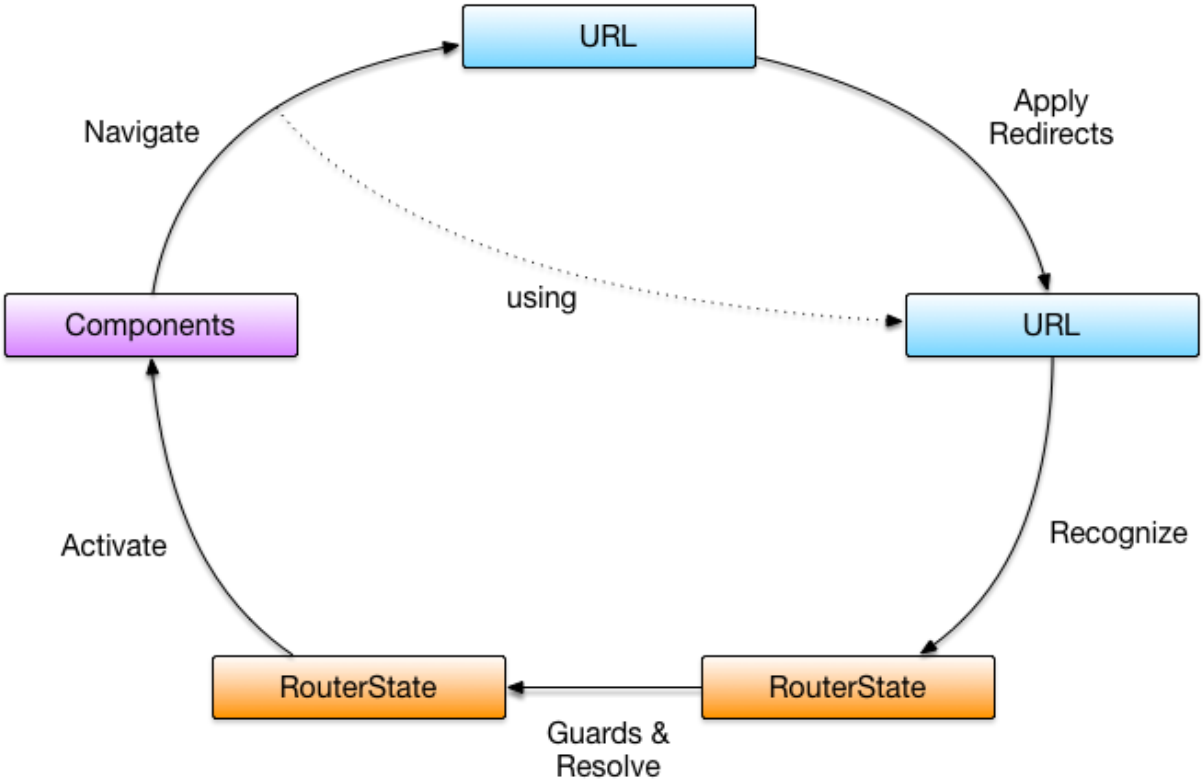
Another way to navigate around is by using the `RouterLink` directive.

```
@Component({
  template: `
    <a [routerLink]="['/', {outlets: {popup: ['message', this.id]}}]">Edit</a>
  `
})
class MessageCmp {
  public id: string;
  constructor(private route: ActivatedRoute) {
    route.paramMap.subscribe(_ => this.id = _.get('id'));
  }
}
```

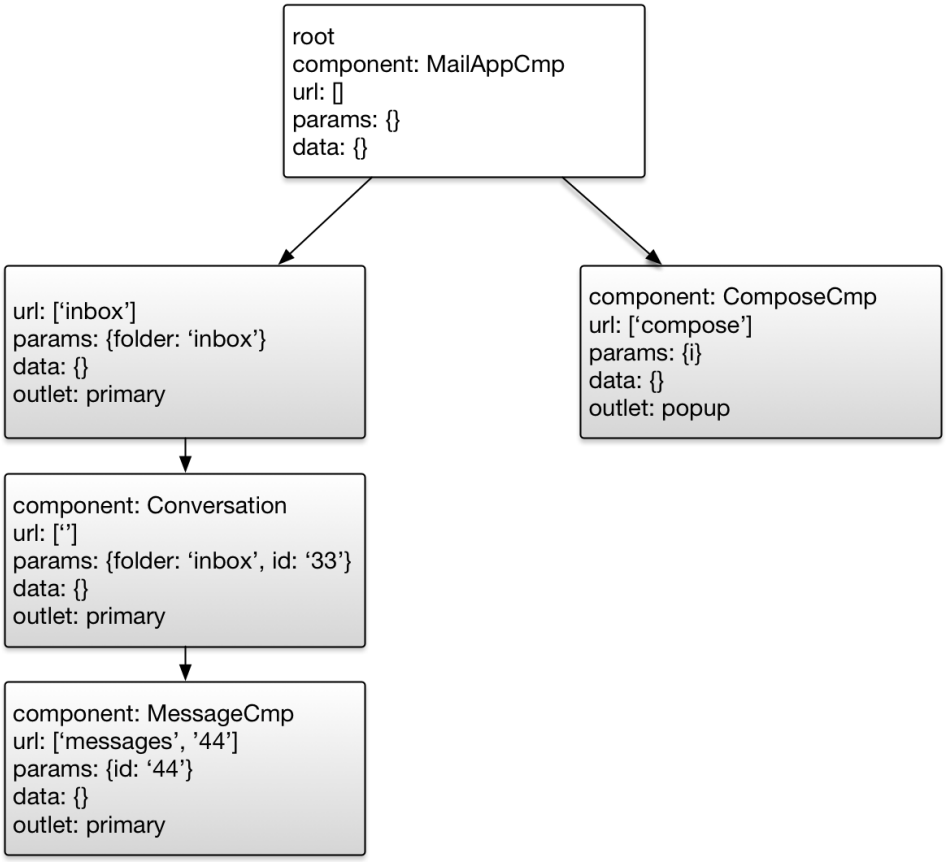
This directive will also update the href attribute when applied to an `<a>` link element, so it is SEO friendly and the right-click open-in-new-browser-tab behavior we expect from regular links will work.

Summary

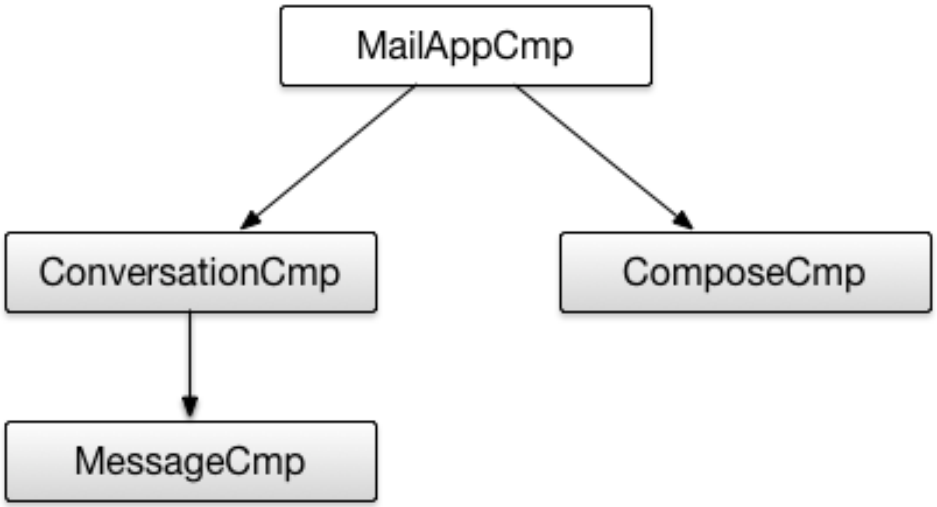
Let's look at all the operations of the Angular router one more time.



When the browser is loading `'/inbox/33/messages/44(popup:compose)'`, the router will do the following. First, it will apply redirects. In this example, none of them will be applied, and the URL will stay as is. Then the router will use this URL to construct a new router state.



Next, the router will instantiate the conversation and message components.



Now, let’s say the message component has the following link in its template:

```
<a [routerLink]="[{outlets: {popup: ['message', this.id]}}]">Edit</a>
```

The router link directive will take the array and will set the href attribute to

`'/inbox/33/messages/44(popup:message/44)'`.

Now, the user triggers a navigation by clicking on the link. The router will take the constructed URL and start the process all over again: it will find that the conversation and message components are already in place. So no work is needed there. But it will create an instance of `PopupMessageCmp` and place it into the popup outlet. Once this is done, the router will update the location property with the new URL.

That was intense–a lot of information! But we learned quite a few things. We learned about the core operations of the Angular router: applying redirects, state recognition, running guards and resolving data, component activation, and navigation. Finally, we looked at an e2e example showing the router in action.

In the rest of this book we will discuss the same operations one more time in much greater depth.