Prefetching Data for an Angular Route

Introduction

*Prefetching* means getting data before it needs to be rendered on a screen. In this Guide you will learn how to fetch data even before a routing change. By the end of this Guide, you will be able to use a resolver, implement a resolver in an Angular app, and apply a common preloader navigation.

Why You Should Use a Resolver

A resolver is a middleware service that plays a role in between a route and a component. Suppose you have a form and want it to first show an empty form to the user with no data, then display a loader while it fetches the user's data, and then hide the loader and fill the form once the data has arrived.

Usually, we get the data using ngOnInit(), which is one of the lifecycle hooks of the Angular component. It means that after the component is loaded, we hit the request for the data and toggle the loader.

We attach a loader in every route that requires data to show just after the component is loaded. A resolver is used to minimize the use of the loader. Instead of attaching the loader in every route, you can add only one loader that works for every route.

This Guide will explain each point of the resolver with an example, so you can use it as it is in your project with the complete picture in mind.

Implementing a Resolver in an App

To implement the resolver, you'll need a couple of APIs for the app. Instead of creating an API here, you can use fake APIs from [JSONPlaceholder](https://jsonplaceholder.typicode.com/). JSONPlaceholder is one of the best API sources for learning frontend concepts without bothering about APIs.

Now that the API issues have been solved, you can start on the resolver. A resolver is nothing but a middleware service, so you'll create a service.

1

$ ng g s resolvers/demo-resolver --skipTests=true

shell

The src/app/resolvers folder has been created in the service. The resolver interface has a resolve() method with two parameters: route, the instance of ActivatedRouteSnapshot, and state, the instance of RouterStateSnapshot.

The loader used to write all AJAX requests in ngOnInit(), but that logic will be replaced with ngOnInit() in the resolver.

Next, create a service with logic to get the list of posts from JSONPlaceholder. Then call the function in the resolver and configure the route for a resolve that will wait until the resolver gets resolved. After resolving the resolver, we will get the data from the route and display it in the component.

Create a Service and Write Logic to Get the Post List

1

$ ng g s services/posts --skipTests=true

shell

Now that the service has been successfully created, it's time to write logic to make an AJAX request.

This model uses best practices to help minimize errors.

1

$ ng g class models/post --skipTests=true

shell

post.ts

1

2

3

4

5

6

export class Post {

id: number;

title: string;

body: string;

userId: string;

}

ts

Now the model is ready to get the list of posts.

post.service.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

import { Injectable } from "@angular/core";

import { HttpClient } from "@angular/common/http";

import { Post } from "../models/post";

@Injectable({

providedIn: "root"

})

export class PostsService {

constructor(private \_http: HttpClient) {}

getPostList() {

let URL = "https://jsonplaceholder.typicode.com/posts";

return this.\_http.get<Post[]>(URL);

}

}

ts

Now the service is ready to be called.

demo-resolver.service.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

import { Injectable } from "@angular/core";

import {

Resolve,

ActivatedRouteSnapshot,

RouterStateSnapshot

} from "@angular/router";

import { PostsService } from "../services/posts.service";

@Injectable({

providedIn: "root"

})

export class DemoResolverService implements Resolve<any> {

constructor(private \_postsService: PostsService) {}

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

return this.\_postsService.getPostList();

}

}

ts

The post list is returned from the resolver. Now you need a routing to configure the resolver, get the data from the route, and display it in the component. To make a routing, you'll need to make a component.

1

$ ng g c components/post-list --skipTests=true

shell

To make the route visible, add router-outlet in the app.component.ts.

1

<router-outlet></router-outlet>

html

Now you can configure the app-routing.module.ts. The following snippet helps you to understand the routing configuration in the resolver.

app-routing.module.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

import { NgModule } from "@angular/core";

import { Routes, RouterModule } from "@angular/router";

import { PostListComponent } from "./components/post-list/post-list.component";

import { DemoResolverService } from "./resolvers/demo-resolver.service";

const routes: Routes = [

{

path: "posts",

component: PostListComponent,

resolve: {

posts: DemoResolverService

}

},

{

path: "",

redirectTo: "posts",

pathMatch: "full"

}

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule {}

ts

A resolve has been added to the route configuration that will make an HTTP request and allow the component to initialize after the HTTP request is successful. The route will collect data fetched by the HTTP request.

How to Apply a Common Preloader Navigation

To show the user that a request is in progress, write a common simple loader in AppComponent. You can customize it per your requirements.

app.component.html

1

2

3

4

<div class="loader" \*ngIf="isLoader">

<div>Loading...</div>

</div>

<router-outlet></router-outlet>

html

app.component.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

import { Component } from "@angular/core";

import {

Router,

RouterEvent,

NavigationStart,

NavigationEnd

} from "@angular/router";

@Component({

selector: "app-root",

templateUrl: "./app.component.html",

styleUrls: ["./app.component.scss"]

})

export class AppComponent {

isLoader: boolean;

constructor(private \_router: Router) {}

ngOnInit() {

this.routerEvents();

}

routerEvents() {

this.\_router.events.subscribe((event: RouterEvent) => {

switch (true) {

case event instanceof NavigationStart: {

this.isLoader = true;

break;

}

case event instanceof NavigationEnd: {

this.isLoader = false;

break;

}

}

});

}

}

ts

When navigation starts, the isLoader value will be true, and you'll see the following output.

imgur

After the resolver resolves, it will be hidden.

Now it's time to fetch the value from the route and display it on the list.

port-list.component.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

import { Component, OnInit } from "@angular/core";

import { Router, ActivatedRoute } from "@angular/router";

import { Post } from "src/app/models/post";

@Component({

selector: "app-post-list",

templateUrl: "./post-list.component.html",

styleUrls: ["./post-list.component.scss"]

})

export class PostListComponent implements OnInit {

posts: Post[];

constructor(private \_route: ActivatedRoute) {

this.posts = [];

}

ngOnInit() {

this.posts = this.\_route.snapshot.data["posts"];

}

}

ts

Here, the value comes from the data in the snapshot of ActivatedRoute. The value is available through the same route that you've configured in routing.

This is what the value looks like in HTML.

1

2

3

4

5

6

<div class="post-list grid-container">

<div class="card" \*ngFor="let post of posts">

<div class="title"><b>{{post?.title}}</b></div>

<div class="body">{{post.body}}</div>

</div>

</div>

html

This CSS snippet will make it prettier.

port-list.component.css

1

2

3

4

5

6

7

8

9

.grid-container {

display: grid;

grid-template-columns: calc(100% / 3) calc(100% / 3) calc(100% / 3);

}

.card {

margin: 10px;

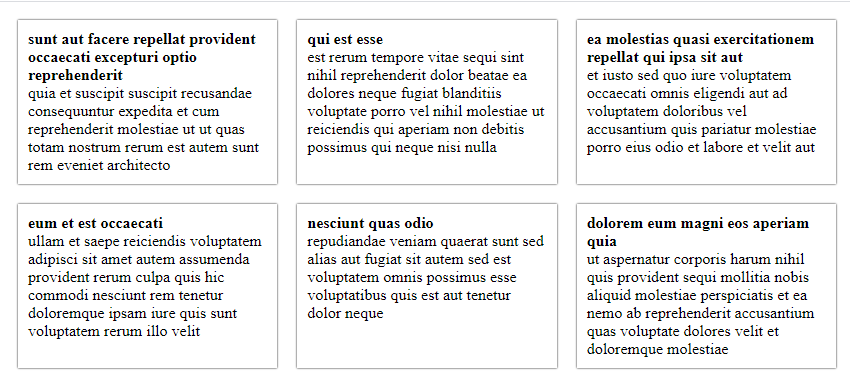
box-shadow: black 0 0 2px 0px;

padding: 10px;

}

css

After the data is fetched from the route, it will be displayed in HTML. The list will look like the below snippet.



At this point you've done everything you have to do to implement the resolver in your project.

Conclusion

You can improve your app's performance as well as UX with the help of a resolver. It's not possible to cover all the conditions in one article, but you can explore it more [here](https://angular.io/guide/router#resolve-pre-fetching-component-data).