Lazy Loading Angular Modules and Preloading Strategies

## **Introduction**

In this guide, you'll learn about lazy loading. Lazy loading helps you render an application faster. It also improves user experience by enhancing the bootstrap and startup process.

This guide will give you hands-on experience with lazy loading implementation by covering the following topics:

* Eager loading and its limitations
* Introduction to lazy loading and its benefits
* Lazy loading implementation
* Preloading strategy

## **Eager Loading and Its Limitations**

Loading is necessary to load components that are required to render an application. Eager loading is the default loading strategy for components in Angular. It loads all the components registered in the app module for the first time. Once all the components are loaded, they will be rendered on the page.

But there is a limitation with this strategy. Eager loading loads all components at the initial render of your application. If your application consists of many components, then it takes a long time to load all components when an app is downloaded, which degrades the user experience. Remember, first impressions are last impressions.

## **Introduction to Lazy Loading**

To overcome the limitations of eager loading, Angular has introduced the lazy loading strategy. It loads only the components that are required to render the page. This saves time and renders the application much faster than eager loading.

Lazy loading has many benefits over eager loading.

1. It only loads components that are required to render the page.
2. It renders the page faster than eager loading .
3. It improves the user experience.

## **Implementing Lazy Loading**

To implement lazy loading, begin by creating an Angular project.

1

$ ng new lazy-loading-demo

shell

The project is ready. To apply the lazy loading feature, create a feature module in AppModule.

1

$ ng g m first --routing

shell

The above command is used to create a module, and --routing is used to generate a routing module.

Create a component in FirstModule. The component will not be part of AppModule, and you won't import it into AppModule. FirstModule will load when it's required.

1

$ ng g c /first/components/first --module first

shell

This will create a component and register it in FirstModule.

Now that the component is ready in FirstModule, you'ree good to go. It's time to configure AppRoutingModule.

**app-routing.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

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

import {

Routes,

RouterModule,

PreloadingStrategy,

PreloadAllModules

} from "@angular/router";

const routes: Routes = [

{

path: "first",

loadChildren: "./first/first.module#FirstModule"

}

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule {}

ts

To make FirstModule part of your application, load it in routes.

Define the path in the path field of the route's object and module path in the loadChildren field.

There are three parts to configure in the loadChildren field of the route's object.

1. Path of the module
2. #
3. Name of the module

So finally, loadChildren will look like this:

1

loadChildren: 'path#moduleName'

shell

In the above configuration of AppRoutingModule, you can see the exact configuration that I've explained above.

You have already made a component in FirstModule. So You're ready to configure FirstRoutingModule.

**first-routing.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

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

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

import { FirstComponent } from "./components/first/first.component";

const routes: Routes = [

{

path: "",

component: FirstComponent

}

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class FirstRoutingModule {}

ts

Next, configure the necessary settings to make FirstComponent part of the route. Then you'll be ready to run the application.

Create a link with the path of FirstComponent to show the first module to be loaded.

**app.component.html**

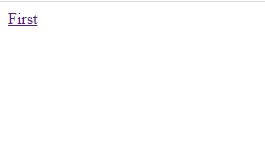
1

<a [routerLink]="['/first']">First</a> <router-outlet></router-outlet>

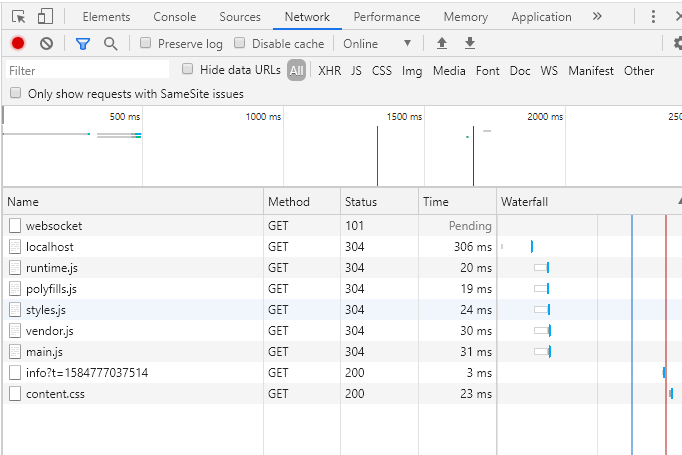
html

Now run the project and open the developer tools by clicking on **F12**, then go to the **Network** tab and refresh the page by hitting **F5**.

You can see the output below.



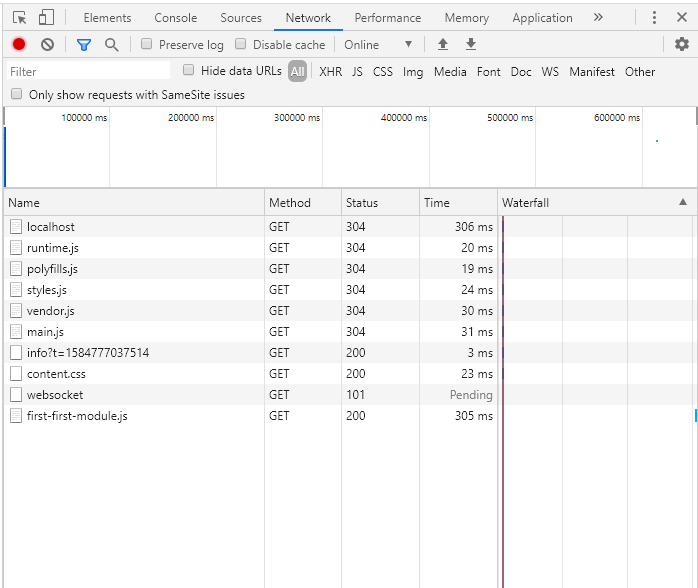
Similarly, in the **Network** tab, you will see this:



As you can see, the application loaded main.js with all the components of an application.

Click on the link that you created to see the first module to be loaded.

After clicking on the link, the **Network** tab will look like this:



You can see that FirstModule has loaded.

You can also see the expected output on the rendered page. But what if the route still has many components to be loaded? Users will again face the slow rendering issue. The preloading strategy now comes into the picture to solve the slow rendering issue.

## **Preloading Strategy**

The preloading strategy loads the first route components to render the page as quickly as possible. After the first route, it loads the other module in the back process. The other module will be ready to use, and the user won't have to wait for all the components to load.

To implement the preloading strategy, add a field in AppRoutingModule.

**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

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

import {

Routes,

RouterModule,

PreloadingStrategy,

PreloadAllModules

} from "@angular/router";

const routes: Routes = [

{

path: "first",

loadChildren: "./first/first.module#FirstModule"

}

];

@NgModule({

imports: [

RouterModule.forRoot(routes, {

preloadingStrategy: PreloadAllModules

})

],

exports: [RouterModule]

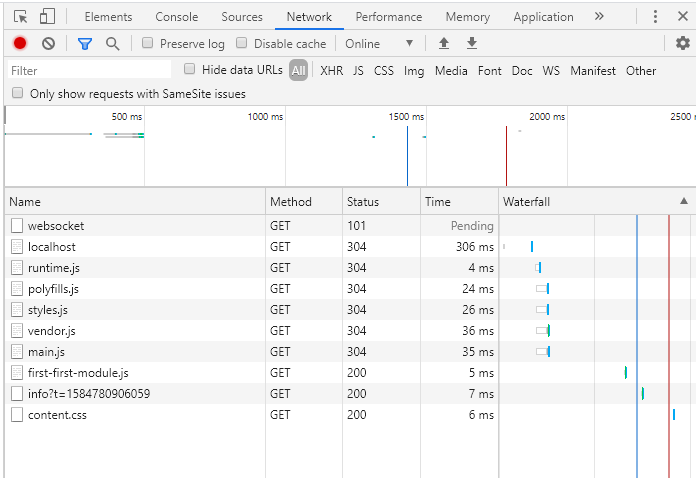
})

export class AppRoutingModule {}

ts

In the above snippet, you can see an object in the forRoot() method that contains preloadingStrategy: PreloadAllModules.

If you run the application now and look at the **Network** tab, it will look like this:



Here you can see, without clicking on the link, that FirstModule loaded after the **main.js**.

That's all you have to implement in your project to use lazy loading.

## **Conclusion**

Lazy loading is an essential feature in an application. It affects many other things, like user experience, bootstrapping an application, and quick rendering. It requires some extra configuration but gives you a lot of benefits. If you want to learn more about lazy loading, read the Angular documentation [here](https://angular.io/guide/lazy-loading-ngmodules).