Angular Router Series: Pillar 3—Lazy Loading, AOT, and Preloading

Nate Lapinski Follow

Nov 21, 2018 · 7 min read

The Rainbow Bridge before a summer thunderstorm

Lazy loading is a useful technique for faster initial page loads. With lazy loading, your application loads faster by shipping only the essential startup code to the browser. Other code is placed inside of feature modules, which are loaded on demand.

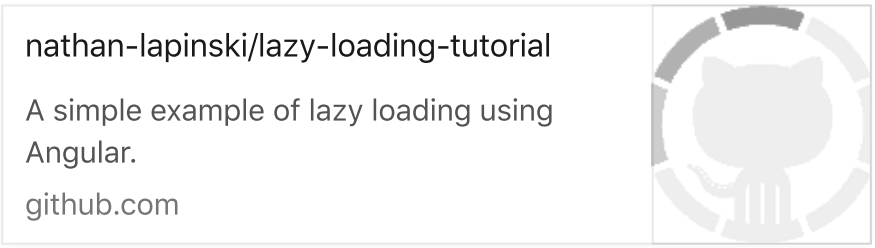
The basics of how to use lazy loading is explained well in the official docs.

In this article, we'll go under the hood and check out how the router implements some parts of lazy loading.

Topics

1. How lazy route configurations are merged into the root configuration
2. How lazy loading works with AOT and JIT compilation
3. How preloading works in the router

If you'd like to look at some code, there is a very basic example of lazy loading in this git repo:



Webpack, SystemJS, and Friends

When you're working with a big framework like Angular, it's easy to lose sight of what dynamically loading a module actually means. All we're really doing is using a tool like Webpack to split our application into separate bundles (Angular CLI handles this for us), and then later pulling those bundles into our application.

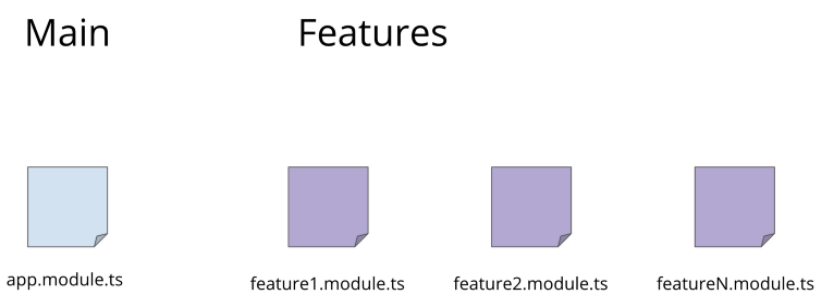
We won't focus on code splitting techniques and module loaders in this article. If you'd like to learn more about the internals of how Angular uses them, check out these fine articles:

Everything you need to know about dynamic components in Angular

As busy as a bee: lazy loading in Angular CLI

Lazy Loading and the Router Configuration

In order to use lazy loading, you must break your application into separate NgModules, often called *feature modules*.



app.module loads first. feature modules are lazy loaded

Assuming you've split your application into feature modules, it's very simple to set up lazy loading. The `loadChildren` property is used inside of a router configuration to indicate that a module should be lazily loaded.

```
const ROUTES = [
{path: 'lazy', loadChildren: './lazy-
module/lazy.module#LazyModule'}
];
```

The value passed to `loadChildren` is a string. Everything to the left of the `#` is the path to the module to lazy load, and everything to the right of the `#` is the name of the NgModule.

When the user navigates to this route (for example: `localhost:4200/lazy`), the router will see the `loadChildren` property and begin loading the feature module (in this example, LazyModule).

> *Care must be taken to avoid adding any sort of reference to the feature module anywhere in the main bundle. Otherwise, it will create a compile time dependency, and the compiler will include the feature module in the main bundle instead of lazy-loading it. That's why we pass a string as the value of* `loadChildren`*, instead of a module reference.*

Once the feature module is loaded, its router configuration must be merged with the application's main router configuration (which is defined in `RouterModule.forRoot()`, usually inside of `app.module` or a dedicated `app-routing.module`).

For reference, our LazyModule has only one route inside of `ROUTES`:

```
1  const ROUTES = [
2    { path: '', component: LazyComponent }
3  ];
4
5  @NgModule({
6    declarations: [
7      LazyComponent
8    ],
9    imports: [
```

feature modules always uses forChild, never forRoot.

The configuration specified in the feature module using `forChild` is merged into the router configuration under the `_loadedConfig` property, during the apply redirects phase of navigation (unless preloading, more on that later).

```
1  return this.configLoader.load(ngModule.injector, route)
2                    .pipe(map((cfg: LoadedRouterConfig) =
3                      route._loadedConfig = cfg;
4                      return cfg;
```

on line 3, the router config of the loaded module is added under route._loadedConfig

This can be verified by checking the `config` property of the Router service.

The takeaway here is that the router must load any lazy modules, **and then merge their configurations with the application's root router configuration.** Once the configurations are merged, our application can access routes inside of the lazy module as normal.

AOT and Lazy Loading

Lazy loading works perfectly well whether you use Angular's just-in-time (JIT) compilation or ahead-of-time compilation (AOT). If you are using **Angular CLI** to build your application, running

```
ng serve --aot
```

will build and serve the application using **AOT** compilation. All of your feature modules will be compiled ahead of time, but they still won't be loaded into your application until they are routed to.

If you use something like

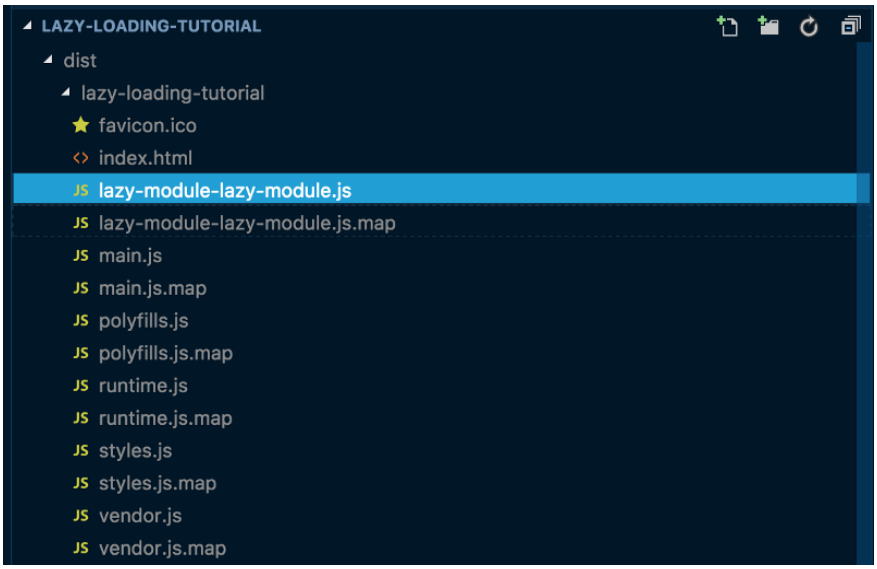
```
ng build --aot=true
```

then you can check your `/dist` folder to see the compiled files. If you are using my sample repo, you'll see something like:

modules have already been compiled into ngfactory

The `lazy-module-lazy-module-ngfactory.js` file is what will be lazy loaded into the application. It is an NgModule which has been compiled into a module factory ahead of time. This means that our application can begin using it immediately after loading it in, instead of having to compile the module at run-time.

The defaults for `ng serve` and `ng build` use JIT instead of AOT. When we build without AOT, `/dist` has modules instead of factories:



This means that during lazy loading, the router will have to compile the NgModule into a factory before it can be used.

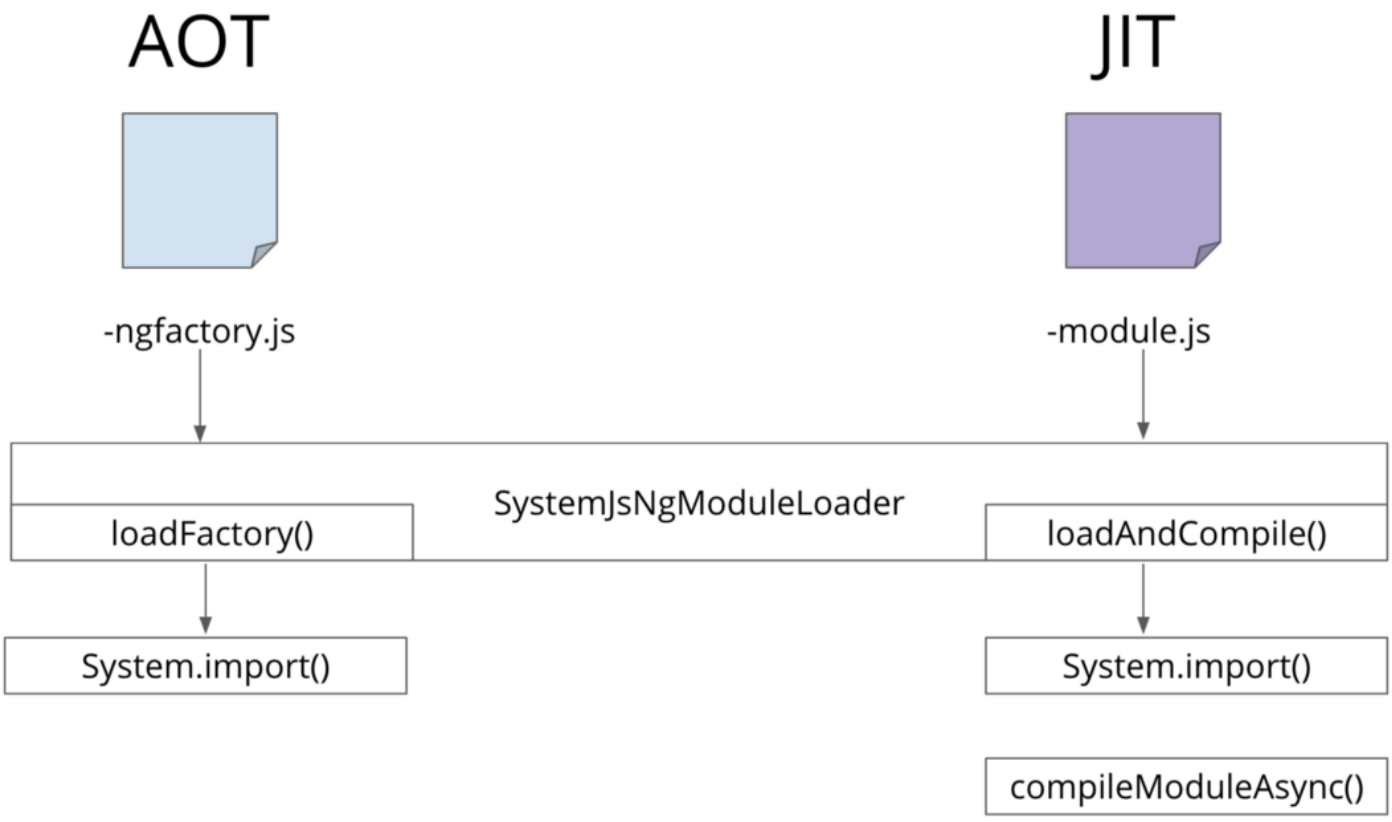
Whether you use AOT or JIT will affect some of the steps Angular takes at run-time during lazy loading. To see how this works, we have to look inside of SystemJsNgModuleLoader's `load` method:

```
export class SystemJsNgModuleLoader implements NgModule

  load(path: string): Promise<NgModuleFactory<any>> {
    const offlineMode = this._compiler instanceof Compi
    // offlineMode === true means AOT. The module is al
    return offlineMode ? this.loadFactory(path) : this.
  }
```

This class is used by Angular during lazy loading

When **AOT** is in use, `offlineMode` is set to `true`, and the compiled factory is loaded into the application. Otherwise, the `NgModule` will be pulled in and compiled using `loadAndCompile`, which uses Angular's `Compiler` service.



With JIT, there is an extra step to compile the module into a factory

The only difference between the two is that with JIT the lazy module must be compiled into a module factory at runtime.

Lazy loading will work regardless of when you choose to compile your application. As always, AOT will be a little faster and is usually the recommended approach.

Everything we have looked at so far happens during the router's navigation cycle. **However, it is possible to programmatically load modules outside of a navigation cycle**. This topic, along with some applications (like dynamic routes), will be explored in a future article.

Preloading Modules

We've seen how lazy loading makes our initial bundle smaller. This means our application will load faster for our users. But our work isn't done yet.

Let's say we have a feature module that most of our users will access. Once we've pulled down the initial bundle and loaded our application, there's no reason to wait for a user to navigate to that popular feature before starting to load it—it's better to start loading it in the background. This is where preloading comes into play.

Preloading works with lazy loading. It's a way to tell Angular when to start loading your feature modules. Angular comes with two default preloading strategies: preload everything (PreloadAllModules), or don't preload anything (NoPreloading).

You can use one of the two default preloading strategies mentioned above, or you can write your own custom preloading strategy.

Custom strategies are very useful when you only want to preload certain modules, or you want to conditionally preload a module or add a delay.

You specify which preloading strategy you want to use by passing an option into `RouterModule.forRoot`:

```
1  RouterModule.forRoot(ROUTES, {
2    preloadingStrategy: CustomPreloadingStrategy | Preloa
3  })
```

You can specify a custom strategy, or use one of the defaults

CustomPreloading strategies are written as services which implement the PreloadingStrategy interface. Adrian Fâciu has already provided a great example of how to implement a custom preloading strategy here, which I strongly encourage you to check out. In this article, we'll instead focus on how and when preloading occurs within the router.

How the Router Schedules Preloading

The router needs some sort of cue to know when to attempt preloading. Internally, the router uses an instance of `RouterPreloader`, which subscribes to the router's `events` observable and listens for navigation events. Each time a `NavigationEnd` event happens, the preloader checks to see if any modules can be preloaded.

```
1  setUpPreloading(): void {
2    this.subscription =
3        this.router.events
4            .pipe(filter((e: Event) => e instanceof Navig
5            .subscribe(() => {});
```

Listens for NavigationEnd events from the router, then calls this.preload to recurse through the route configurations

Fun fact: you can also subscribe to `this.router.events` anywhere in your application to see all router events.

If you are using a custom strategy, then the exact mechanics of when and how your module is preloaded depends on how your custom strategy implements PreloadingStrategy's abstract `preload` method. However, the router always runs checks to see if it can preload anything when it sees `NavigationEnd`.

Understanding the CanLoad guard

Last but not least, it's worth noting that lazy loading gets its own router guard, called **CanLoad**. It determines whether or not a module can be lazily loaded. If it returns `false`, the module won't even be loaded into the browser.

Note that this guard blocks all preloading. The two cannot be used together.

If this were not the case, we could run into issues where the user is on a page, and in the background we try to preload a module. This activates the canLoad guard check, which could fail and redirect the user to a different page (such as log in) which is definitely not a scenario we want for our users.

If you want to know more about route guards, I've written about them in this article on the router's navigation cycle.

Conclusion

We've learned how the router merges configurations, works with JIT and AOT, and schedules preloading strategies.

This concludes the Three Pillars of the Angular Router series. If you haven't already, please check out the rest of the series:

0. Series Overview
1. Router States and URL Matching
2. The Router Navigation Cycle

There are many more articles to come regarding the router. This series was just the foundation. Stay tuned!