Route-level code splitting in Angular

- Why code splitting matters
- Code splitting techniques
  - Sample application
  - Route-level code splitting
  - Show a spinner
- Conclusion

This post explains how to set up route-level code splitting in an Angular application, which can reduce JavaScript bundle size and dramatically improve Time to Interactive.

You can find the code samples from this article on GitHub. The eager routing example is available in the eager branch. The route-level code splitting example is in the lazy branch.

{% Aside %} This post assumes understanding of the Angular router. For a guide on how to use it, visit Angular's official documentation. {% endAside %}

Why code splitting matters

The ever growing complexity of web applications has led to a significant increase in the amount of JavaScript shipped to users. Large JavaScript files can noticeably delay interactivity, so it can be a costly resource, especially on mobile.

The most efficient way to shrink JavaScript bundles without sacrificing features in your applications is to introduce aggressive code splitting.

Code splitting lets you divide the JavaScript of your application into multiple chunks associated with different routes or features. This approach only sends users the JavaScript they need during the initial application load, keeping load times low.

{% Aside 'note' %}

By using code splitting, Twitter and Tinder observed improvements of up to 50% for their Time to Interactive.

{% endAside %}

Code splitting techniques

Code splitting can be done at two levels: the **component level** and the **route level**.

- In component-level code splitting, you move components to their own JavaScript chunks and load them lazily when they are needed.
- In route-level code splitting, you encapsulate the functionality of each route into a separate chunk. When users navigate your application they fetch the chunks associated with the individual routes and get the associated functionality when they need it.

This post focuses on setting up route-level splitting in Angular.

Sample application

Before digging into how to use route level code splitting in Angular, let's look at a sample app:

{% IFrame { src: 'https://stackblitz.com/github/mgechev/code-splitting-web-dev/tree/eager?embed=1&file=src/app/app.component.ts&view=preview' } %}

Check out the implementation of the app's modules. Inside `AppModule` two routes are defined: the default route associated with `HomeComponent` and a `nyan` route associated with `NyanComponent`:

```
@NgModule({
  ...
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {
        path: '',
        component: HomeComponent,
        pathMatch: 'full'
      },
      {
        path: 'nyan',
        component: NyanComponent
      }
    ])
  ],
  ...
```

```
})
export class AppModule {}
```

Route-level code splitting

To set up code splitting, the `nyan` eager route needs to be refactored.

Version 8.1.0 of the Angular CLI can do everything for you with this command:

```
ng g module nyan --module app --route nyan
```

This will generate:

- A new routing module called `NyanModule`
- A route in `AppModule` called `nyan` that will dynamically load the `NyanModule`
- A default route in the `NyanModule`
- A component called `NyanComponent` that will be rendered when the user hits the default route

Let's go through these steps manually so we get a better understanding of implementing code splitting with Angular!

When the user navigates to the `nyan` route, the router will render the `NyanComponent` in the router outlet.

To use route-level code splitting in Angular, set the `loadChildren` property of the route declaration and combine it with a dynamic import:

```
{
  path: 'nyan',
  loadChildren: () => import('./nyan/nyan.module').then(m => m.NyanModule)
}
```

There are a two key differences from the eager route:

1. You set `loadChildren` instead of `component`. When using route-level code splitting you need to point to dynamically loaded modules, instead of components.
2. In `loadChildren`, once the promise is resolved you return the `NyanModule` instead of pointing to the `NyanComponent`.

The snippet above specifies that when the user navigates to `nyan`, Angular should dynamically load `nyan.module` from the `nyan` directory and render the component associated with the default route declared in the module.

You can associate the default route with a component using this declaration:

```
import { NgModule } from '@angular/core';
import { NyanComponent } from './nyan.component';
import { RouterModule } from '@angular/router';

@NgModule({
  declarations: [NyanComponent],
  imports: [
    RouterModule.forChild([{
      path: '',
      pathMatch: 'full',
      component: NyanComponent
    }])
  ]
})
export class NyanModule {}
```

This code renders `NyanComponent` when the user navigates to `https://example.com/nyan`.

To check that the Angular router downloads the `nyan.module` lazily in your local environment:

{% Instruction 'devtools-network', 'ol' %}

1. Click **NYAN** in the sample app.
2. Note that the `nyan-nyan-module.js` file appears in the network tab.

{% Img src="image/admin/wT4xLV2OkrZ2b7QaQz8L.png", alt="Lazy-loading of JavaScript bundles with route-level code splitting", width="800", height="524" %}

Find this example on GitHub.

Show a spinner

Right now, when the user clicks the **NYAN** button, the application doesn't indicate that it's loading JavaScript in the background. To give the user feedback while loading the script you'll probably want to add a spinner.

To do that, start by adding markup for the indicator inside the `router-outlet` element in `app.component.html`:

```
<router-outlet>
  <span class="loader" *ngIf="loading"></span>
</router-outlet>
```

Then add an `AppComponent` class to handle routing events. This class will set the `loading` flag to `true` when it hears the `RouteConfigLoadStart` event and set the flag to `false` when it hears the `RouteConfigLoadEnd` event.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  loading: boolean;
  constructor(router: Router) {
    this.loading = false;
    router.events.subscribe(
      (event: RouterEvent): void => {
        if (event instanceof NavigationStart) {
          this.loading = true;
        } else if (event instanceof NavigationEnd) {
          this.loading = false;
        }
      }
    );
  }
}
```

In the example below we've introduced an artificial 500 ms latency so that you can see the spinner in action.

{% IFrame { src: 'https://stackblitz.com/github/mgechev/code-splitting-web-dev/tree/lazy?embed=1&file=src/app/app.component.ts&view=preview' } %}

{% Aside 'warning' %} Code splitting can significantly improve an app's initial load time, but it comes at the cost of slowing down subsequent navigation. In the next post on route preloading you'll see how to work around this problem! {% endAside %}

Conclusion

You can shrink the bundle size of your Angular applications by applying route-level code splitting:

1. Use the Angular CLI lazy-loaded module generator to automatically scaffold a dynamically loaded route.
2. Add a loading indicator when the user navigates to a lazy route to show there's an ongoing action.

□□□ #2
□ □□□ □□□ 2026-10-31 21:34:17 UTC
□ □□□ □□□ 2026-10-31 21:34:18 UTC