Dynamically Loading Components with Angular CLI

Chaz Gatian Follow
Jan 10, 2018 · 6 min read

This post and the code for it was a team effort, including my teammates Zack Ream, Ryan Kara, Ben Kindle, and Jason Lutz.

The code below was also inspired by the work of George Kalpakas, from PR#18428.

• • •

When moving from a multi-page application to a SPA, one of the problems that presents itself is the payload size upon initial load. By default, in an Angular application everything is bundled into one payload, which means as the application grows, so does the time that it takes to load.

The typical way to solve this problem is by utilizing the router to lazy load modules, as is shown in the Angular documentation. This is great when the content has a route, but what about situations where the components are *not* a part of a separate route?

In this post we will show how to leverage the Angular CLI to split components into their own bundles, which will allow them to only be loaded when needed.

Duping Angular CLI

The Angular CLI, more specifically the `@ngtools/webpack` package, performs static analysis on an application during build time to locate all the lazy-loaded router paths. As each router path is analyzed Webpack creates a chunk that can then be loaded when route is activated. Contained within this chunk is the `ModuleFactory` for the given route.

Traditionally, being able to split *without* utilizing the router has been a difficult task, and is something that's been requested. We can, however, take advantage of the static analysis and trick Angular CLI into splitting out our component modules, which enables us to dynamically load components.

Let's dive in to how we can achieve this by creating a dynamic `MessageComponent`!

Creating our first Dynamic Component

First, we create a folder called "dynamic-modules" to hold our components.

Within here, create another folder called "message", to hold our `MessageComponent`. Next, add the following code:

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-message',
5    template: 'Hello World',
6  })
7  export class MessageComponent implements OnInit {
8    constructor() { }
```

Now we need to create a module that *declares* this component. We will call this `MessageModule.`

```
import { NgModule } from '@angular/core';

import { DYNAMIC_COMPONENT } from '../../dynamic-compo
import { MessageComponent } from './message.component'

@NgModule({
  declarations: [
    MessageComponent,
  ],
  imports: [
  ],
  providers: [
```

Note that, in order for a `ComponentFactory` to be generated, the component must also be added to the module's `entryComponents`.

Dynamic Component Manifest

To aid in the consumption of dynamic components, we will create a simple manifest interface which resembles the `Route` interface in `@angular/router`.

```
export interface DynamicComponentManifest {
  componentId: string;
  path: string;
  loadChildren: string;
```

The idea here is simple: we will add a new manifest entry for each component we wish to dynamically load.

It's important that this interface resembles the `Route` type. When static analysis kicks off during the build process and the `ROUTES` token is analyzed, `@ngtools/webpack` will create factories for each path with a configured `loadChildren` property. Note that the `path` property can really be **any** unique string, as long as it doesn't conflict with an existing route in the application. In addition, `componentId` is added to uniquely represent the component we wish to load.

Dynamic Component Loader

Next, we will need a module that will be responsible for loading the manifests (for the compiler's static analysis), and providing a service to locate and retrieve the `ComponentFactory` objects.

Let's start by creating a new "dynamic-component-loader" folder to house this code. In this folder, we create a new module called `DynamicComponentLoaderModule`.

Our module consists of a `forRoot()` function that accepts the array of `DynamicComponentManifest`. With this, consumers of the module are now required to provide a list of manifests during bootstrap of the application.

Now we must trick the CLI into parsing this array, as part of its static analysis.

With the `DynamicComponentManifest` array passed in, we can simply provide it to the `ROUTES` multi-provider.

```
@NgModule({
  providers: []
})
export class DynamicComponentLoaderModule {
  static forRoot(manifests: DynamicComponentManifest[]
    return {
      ngModule: DynamicComponentLoaderModule,
      providers: [
        // provider for Angular CLI to analyze
        { provide: ROUTES, useValue: manifests, multi:
```

Creating an factory locator service

Now, we need to implement the service that can locate and retrieve the compiled `ComponentFactory` objects.

Our service will make use of the `SystemJsNgModuleLoader`, so add it to the module's list of providers:

```
providers: [
{ provide: NgModuleFactoryLoader, useClass:
SystemJsNgModuleLoader }]
```

To make things a bit easier, let's also create a new `InjectionToken` to allow our application to read the manifests.

Create the following token:

```
export const DYNAMIC_COMPONENT_MANIFESTS = new
InjectionToken<any>('DYNAMIC_COMPONENT_MANIFESTS');
```

Then, update `DynamicComponentLoaderModule` with a new provider that maps the token to the manifests.

```
@NgModule({
  providers: [
    { provide: NgModuleFactoryLoader, useClass: System
  ],
})
export class DynamicComponentLoaderModule {
  static forRoot(manifests: DynamicComponentManifest[]
    return {
      ngModule: DynamicComponentLoaderModule,
      providers: [
        { provide: ROUTES, useValue: manifests, multi:
```

We can now create our service, `DynamicComponentLoader`, making sure to declare it in our module.

In the constructor of this service, we need to inject the `DYNAMIC_COMPONENT_MANIFESTS` token and the `NgModuleFactoryLoader`.

```
@Injectable()
export class DynamicComponentLoader {

  constructor(
    @Inject(DYNAMIC_COMPONENT_MANIFESTS) private manife
    private loader: NgModuleFactoryLoader
```

Then, create a new public method called `getComponentFactory`.

This function will, given a `componentId`, locate the appropriate component module using the manifest array. Then load the module using the `NgModuleFactoryLoader`, and create a new instance of the module.

```
getComponentFactory<T>(componentId: string, injector?:
  const manifest = this.manifests
    .find(m => m.componentId === componentId);

  const p = this.loader.load(manifest.loadChildren)
    .then(ngModuleFactory => {
      const moduleRef = ngModuleFactory.create(injecto

      // Problem! How do we get at the component this
```

However, there's a problem. We have a moduleRef instance that contains the `ComponentFactory` we need, but we can only resolve the Component by type.

In order to locate the correct component factory we are going to need the dynamic component modules to specify the default component we wish to create. So we create a convention: **each module should specify a token that represents the dynamic component type**. That way, when we resolve a module, we can use the `Injector` to locate this token and thus the appropriate component type.

Start by creating a new `InjectionToken`:

```
export const DYNAMIC_COMPONENT = new InjectionToken<any>
('DYNAMIC_COMPONENT');
```

We must go back to the `MessageModule` and provide this token mapped to the `MessageComponent`.

```
@NgModule({
  declarations: [
    MessageComponent,
  ],
  providers: [
    { provide: DYNAMIC_COMPONENT, useValue: MessageCom
  ],
  entryComponents: [
    MessageComponent
```

We must now update the `DynamicComponentLoader` service's `getComponentFactory` method to now utilize the `moduleRef`'s injector

to locate the token.

Once the token is located we can then call into the `ComponentFactoryResolver` on the `moduleRef` find the appropriate `ComponentFactory`.

```
getComponentFactory<T>(componentId: string, injector?:
    const manifest = this.manifests
      .find(m => m.componentId === componentId);

    const p = this.loader.load(manifest.loadChildren)
      .then(ngModuleFactory => {
        const moduleRef = ngModuleFactory.create(injec

        // Read from the moduleRef injector and locate
        const dynamicComponentType = moduleRef.injecto
        // Resolve this component factory
```

Providing the manifest

With the `DynamicComponentModule` plumbing complete, we can now create a manifest in our `AppModule`.

Open `app.module.ts` and create a new `manifests` object.

```
const manifests: DynamicComponentManifest[] = [
 {
   componentId: 'message',
   path: 'dynamic-message',
   loadChildren: './dynamic-modules/message/message.mc
 }
```

We've given our component the `componentId` of "message". This is the key that will be used to return the matching `ComponentFactory`. The `loadChildren` property points to the relative location of the module, much like a lazy-loaded route. The `path` property can be anything, provided it *doesn't* collide with any other existing routes.

Update the `AppModule` to include our manifests object in the `forRoot` call of `DynamicComponentLoaderModule`.

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    DynamicComponentLoaderModule.forRoot(manifests),
  ],
  providers: [],
  bootstrap: [
```

We are now ready to lazy load this component! 🎊

Injecting the component into a template

At this point, we have a service that will give us a direct reference to a `ComponentFactory`, automatically performing the necessary lazy-loading. All we have to do now is use that factory to create our component, by any means possible (`ViewContainerRef`, Angular CDK Portals, etc.).

In this case, let's use a `ViewContainerRef`.

Add the following element to your `app.component.html` file:

```
<button type="button" (click)="loadComponent()">Load!</

<div #testOutlet></div>
```

Then, in the `app.component.ts` file, add the following:

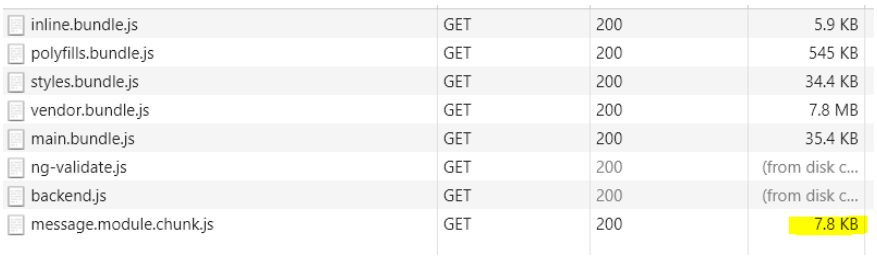
```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {

  @ViewChild('testOutlet', {read: ViewContainerRef}) t

  constructor(
    private dynamicComponentLoader: DynamicComponentLo
  ) { }

  loadComponent() {
    this.dynamicComponentLoader
      .getComponentFactory<MessageComponent>('message'
```

Run the application and click the “Load!” button. Not only does the component load, but if you inspect the network traffic you will see Webpack has created a separate chunk for this component.



inline.bundle.js	GET	200	5.9 KB
polyfills.bundle.js	GET	200	545 KB
styles.bundle.js	GET	200	34.4 KB
vendor.bundle.js	GET	200	7.8 MB
main.bundle.js	GET	200	35.4 KB
ng-validate.js	GET	200	(from disk c...
backend.js	GET	200	(from disk c...
message.module.chunk.js	GET	200	7.8 KB

Why would you do this?

Now that we have seen how to dynamically load a component, you may be asking yourself: why you would want to do this? There are several scenarios where this could prove useful, but before going through a few, we should first say that this dynamic strategy is not something you should do with all of your components. The overhead required would eventually not be worth the gain you are getting. However, when used appropriately, it could be a nice boost to your application.

The two most obvious use cases would be for large or rarely-used components. In the case of a component with a large payload size, it may be beneficial to let the rest of the application load and save the component’s payload for a later time. In the same vein, if you know that 95% of users are never going to use certain components, these

could be great candidates for loading in this dynamic way in order to avoid the initial payload hit.

There are other less obvious cases for dynamically loading content. One such way is a situation solved by the pull request mentioned at the top of this PR for Angular.io. Angular.io has a static HTML page with tags that are dynamically hydrated with the appropriate Angular components when required. The benefit here is that although the page uses a lot of Angular components, the page can load initially with the bare minimum and then incur the payload cost when it is actually required.

Angular.io has a lot of content, but the typical user in a given session will not view it all.

I'm sure there are other use cases, so let us know in the comments how you have used it!

A working demonstration and code can be found here:
https://github.com/devboosts/dynamic-component-loader

Deeper learning

For more about dynamic components checkout Max NgWizard's post on Dynamic Components.