Dynamic Component Generation in Lazy-Loaded Routes

Jim Armstrong Follow
Jun 26, 2019 · 10 min read

— *Exploit Data Driven Component Layout, Loaded On-Demand in Angular*

One of the fun things about being an applied mathematician in front-end development is the never-ending stream of complex, interactive problems that users wish to solve. These range from gamification of business applications to data-driven user experiences. Two interesting applications of the latter topic include dynamic routing through an application and dynamic component layout. Both are examples of user experiences that vary based on factors such as user role and prior uses of the application. In other words, two different users may be presented with an entirely different experience based on both *a priori* knowledge of the user and how the user interacts with the application in real time.

The general solution I've applied to dynamic routing is a data-driven, adaptive decision tree. This is, unfortunately, a client-proprietary solution and can not be shared in public. However, I built the solution on top of two projects that I have released to the public in my Github, a general tree data structure and a lightweight expression engine. Both are written in Typescript. I believe any enterprising developer with some fundamental knowledge of decision trees could duplicate my work. So, here is the best I can do for you at present:

theAlgorithmist/TSTree

Typescript Math Toolkit General Tree Data Structure - theAlgorithmist/TSTree github.com

theAlgorithmist/TSExpressionEngine

Typescript Math Toolkit Expression Engine. Contribute to theAlgorithmist/TSExpressionEngine development by creating an…

github.com

Dynamic, Data-Driven Component Layout

This article discusses how to layout Angular components programmatically, based on a simple JSON data structure. A simple example I've worked on in the past is where components are vertically stacked in an order that is server-generated. Consider a case where three components, A, B, and C could be presented to a user. They might be displayed in the order A, B, C, or A, C, B, or perhaps C, B, A. In general, there are $n!$ display permutations of n components (displayed n at a time). One might be willing to struggle with a layout that could accommodate all possible scenarios for three components, but what about when the client later indicates there could be anywhere from three to eight components? And, we know how clients think, so that 3–8 range will not stay constant for very long. In short, this is a situation that is much better managed with an imperative instead of declarative approach.

Thankfully, the Angular team has provided everything we need to dynamically generate components at runtime. But, before we move on, here is the repo for this article so that you can follow along with the deconstruction, and have the code for experimentation and future usage in projects.

theAlgorithmist/Angular8LazyLoadDynamic

Angular 8 Dynamic Component Generation in a Lazy-Loaded Route - theAlgorithmist/Angular8LazyLoadDynamic

github.com

The Scenario

This demo simulates a scenario where a user logs into an application and then selects a navigational element that routes to another area of the application. The user experience, however, is tailored to each specific user based on information that is known about the user after login. A service call is to be made before activating the route. The service returns some JSON data that describes the order in which child components are to be displayed in the Angular Component associated with the selected route. The JSON data also provides some simple textual and numerical data that is used for binding within each of the child components.

Since the order of the components is not known in advance and the number of components can also vary, the child components are dynamically created and then rendered into the parent component's template. For demonstration purposes, the number of components in the code distribution is limited to three. The technique, however, is easily scaled to any number of components in any arbitrary order.

The Process

I'll treat the explanation in a cookbook fashion since additional information on each step in the process is readily available online for subsequent study.

1 — Each child component that could be rendered into a parent component must be provided as an *EntryComponent* into the *Module* associated with the route. This is why the route should be lazy-loaded as there is no guarantee that every user will navigate to this route.

2 — A route *Resolver* is used to ensure that the JSON data is loaded before the route activates. This is the mechanism a server would use to dynamically alter the experience for each individual user.

3 — In order for an Angular component to be dynamically displayed in a template, it must be added to a *ViewContainerRef* associated with a DOM container after creating the component. An *Attribute Directive* is used for this purpose.

4 — Each child component is to be generated by two *Factories*. One factory (that we write) generates component type and raw data instances based on a symbolic code and a

known number of components. So, if the component range is changed from 3–8 to 2–12 at a later point, the four new items must be added to the factory. Another factory (provided by Angular and discussed below) is used to create the actual Angular component at runtime.

5 — The template for the lazy-loaded component consists of a *ng-container* as the primary container with a *ngFor* that loops over the number of dynamic components specified in the JSON data.

6 — Each dynamic component is associated with a *ng-template* by using an *attribute directive*.

7 — A *QueryList* of dynamic item attribute directives is processed by the parent component. Each child component is created by an Angular Component *Factory* (provided by a factory resolver) and then added to the *ViewContainerRef* of the *ng-template*. Data for each component is then added to the newly created component for binding. This requires some handshaking between the parent component code and the attribute directive. The actual separation of concerns can be experimented with and adjusted to suit your specific desires.

Application Layout

The application structure for this demo is rather simple. There is a single application module and component. The main app component displays a button whose markup contains a *routerLink*. That is used to route the user to the single feature module, appropriately named ‘feature’ :)

The main app module provides a single route resolver that is used to ensure the JSON data for dynamic layout is loaded before the route is activated.

All libraries, directives, and components for the single feature are provided in the *feature* folder.

The model for dynamically generated components is provided in *src/app/models*.

There is no relevant code in the main app component and the only item worth deconstructing is the main app routing module. Relevant code from the routing module is provided below.

/src/app/app-route-module.ts

```
const routes: Routes = [
  {
    path: `feature`,
    resolve: { model: AppRouteResolver },
    loadChildren: () => import(`./feature/feature.module`).then(m =>
m.FeatureModule)
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  providers: [AppRouteResolver],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Note the new Angular 8 dynamic import syntax for lazy-loaded routes. This module also provides the route resolver, *AppRouteResolver*. The *model* property is used when loading data from the activated route.

Now, we can deconstruct each of the above-listed steps.

(1) Feature Module

Look at */src/app/feature/feature.module.ts*. The important code is shown below.

```
export const DYNAMIC_COMPONENTS: Array<any> = [
  BaseComponent, Component1Component, Component2Component,
Component3Component
];

@NgModule({
  declarations: [
    DynamicItemDirective, FeatureComponent, ...DYNAMIC_COMPONENTS
  ],
  imports: [
    CommonModule,
    RouterModule,
    RouterModule.forChild(localRoutes),
    ],
    providers: [],
    entryComponents: DYNAMIC_COMPONENTS
    exports: [
      DynamicItemDirective, ...DYNAMIC_COMPONENTS
    ]
  })
```

The three dynamic components in this example are *Component1Component*, *Component2Component*, and *Component3Component*. Yes, those are stupid names, but slightly better than my original choice of Moe, Larry, and Curly :) Each of these components extends *BaseComponent*.

In particular, note the declaration of all of dynamic components in the *entryComponents* property of *NgModule*. Since there is no direct reference to any of these components in a template, Angular needs this information directly for compilation purposes. Without *entryComponents* Angular will tree-shake out those components because they are never referenced in a template.

The attribute directive, *DynamicItemDirective*, is used to associate a *ViewContainerRef* with a specific DOM element (*ng-template* in this example).

(2) Route Resolver

The resolver is used by the main app component and is provided in */src/app/app-route.resolver.ts*. This code implements the *Resolve<T>* interface by providing a concrete implementation of the *resolve()* method.

```
  @Injectable({providedIn: 'root'})
  export class AppRouteResolver implements Resolve<LayoutModel>
  {
    constructor(@Inject(DataService) protected _service: DataService)
    {
      // empty
    }

    resolve(): Observable<LayoutModel>
    {
      // load layout model
      return < Observable<LayoutModel> >
  this._service.getData('/assets/layout-model.json');
```

```
    }
  }
```

In this demo, the JSON data described by the *LayoutModel* interface is loaded from a file. An *Observable<LayoutModel>* is returned and used to resolve the route before activation. After activation, the model data is available from the activated route snapshot.

Here is the model loaded in the demo,

```
  {
    "layout": [
      {
        "component": "component2",
        "data": {
          "id": 2,
          "title": "Dynamic Component #2"
        }
      },
      {
        "component": "component3",
        "data": {
          "id": 3,
          "title": "Dynamic Component #3"
        }
      },
      {
        "component": "component1",
        "data": {
          "id": 1,
          "title": "Dynamic Component #1"
        }
      }
    ]
  }
```

The *component* property is the symbolic name used to uniquely identify a component. The *data* property provides some basic data to each of the programmatically created child components.

(3) DynamicItemDirective

This attribute directive is provided in */src/app/feature/directives/dynamic-item.directive.ts* . The important code is provided below; it provides a means to inject a *ViewContainerRef* and then add a component to that reference. The added component is generated from an Angular *Component Factory*.

```
constructor(protected _viewContainerRef: ViewContainerRef){}

public addComponent(componentFactory: ComponentFactory<any>):
ComponentRef<any>
{
  this._viewContainerRef.clear();

  return this._viewContainerRef.createComponent(componentFactory);
}
```

(4) ComponentItemFactory

The relevant code from the *ComponentItemFactory* class from */src/app/feature/libs* is provided below.

```
public static create(name: string, data: object): ComponentItem |
null
{
  switch (name)
  {
    case "component1":
      return new ComponentItem(Component1Component, data);

    case "component2":
      return new ComponentItem(Component2Component, data);

    case "component3":
      return new ComponentItem(Component3Component, data);

    default:
      return null;
  }
}
```

ComponentItem instances are created on the basis of a symbolic code. Note that the factory must accommodate all potential components in its current form, regardless of how many are actually created during any particular execution of the application.

One part of a *ComponentItem* is a reference to the Typescript class for the relevant Angular Component. The other part is the raw data to be passed to a newly-generated Angular Component in step 7.

The remaining work is provided in the constructor of the helper class, *ComponentItem*,

/src/app/feature/libs/ComponentItem.ts

```
protected _component: Type<any>;
protected _data: Object;

constructor(component: Type<any>, data: Object)
{
  if (component !== undefined) {
    this._component = component;
  }

  if (data !== undefined) {
    this._data = JSON.parse(JSON.stringify(data));
  }
}
```

(5–6) The Template

Now, we need to provide a container in the DOM for each of the dynamically generated components. This task is performed in the feature component's template for the lazy-loaded route,

/src/app/features/feature.component.html

```
<ng-container *ngFor="let item of items">
  <ng-template dynamic-item></ng-template>
</ng-container>
```

The dynamic item directive is associated with each *ng-template* instance by an attribute selector. In short, this provides the 'glue' by which the dynamically-created component is eventually rendered into the DOM. Note that the number of possible component permutations is really not relevant; the template markup remains just as simple for three components or a dozen!

(7) Query List and Angular Component Factory

The most important code in the entire demo is found in */src/app/feature/feature.component.ts*. Relevant segments are provided below.

```
// Collection of Component and data with an iterable for binding
public componentItems: Record<string, ComponentItem> = {};
public items: Array<string>;

protected _model: LayoutModel;

// reference to dynamic item directive instances and associated array
@ViewChildren(DynamicItemDirective)
protected _dynamicItems: QueryList<DynamicItemDirective>;
protected _dynamicItemsArr: Array<DynamicItemDirective>;

constructor(protected _activatedRoute: ActivatedRoute,
            protected _componentFactoryResolver:
ComponentFactoryResolver,
            protected _changeDetectorRef: ChangeDetectorRef)
{
  this.items = new Array<string>();
}
```

A *Record* of *ComponentItem* instances keyed to the symbolic code used to create the component is maintained by the class. An *Array* of those keys (*items*) is also created for easy template binding. The *Record* is useful for future modifications, but only the *items* array is required for the demo.

The code initially references the *model* property (specified in the resolver definition) of the activated route's snapshot and then accesses the *layout* property to load and parse the JSON data that specifies the component layout. This is performed in the *ngOnInit* lifecycle method,

```
public ngOnInit(): void
{
  this._model = <LayoutModel>
this._activatedRoute.snapshot.data['model'];

  const layout: Array<Object> = this._model['layout'];

  layout.forEach( (item: Object) => {
    this.componentItems[item['component']] =
ComponentItemFactory.create(item['component'], item['data']);
    this.items.push(item['component']);
  });
}
```

The *QueryList* of *DynamicItemDirective* instances is used to 'glue' each dynamically created component together with an associated *ViewContainerRef*. This list will not be available until the *ngAfterViewInit* lifecycle method is called, so the most important code is deferred to that handler. There is a single loop that iterates over each dynamic component specified in the JSON data,

```
  for (i = 0; i < n; ++i)
  {
    // this is the dynamic layout item or template
    dynamicItem = this._dynamicItemsArr[i];

    // this is the component that gets rendered into it
    item = this.componentItems[this.items[i]];

    // if there is another item to render ...
    if (item)
    {
      // add the component to the dynamic view and assign its data
      factory      =
  this._componentFactoryResolver.resolveComponentFactory(item.component
  );
      componentRef = dynamicItem.addComponent(factory);

      componentRef.instance.data = item.data;
    }
  }
```

The *ComponentFactoryResolver* is the Angular facility that takes our simple *ComponentItem* instance and provides an actual factory that can generate a complete Angular Component. This factory is provided to the *ViewContainerRef* in the attribute directive and the resulting component reference is the result of adding that component to the *ViewContainerRef*. Once we have that reference, it's easy to add data from the JSON file into the generated component.

Now, some of this code could be moved inside the attribute directive, so do not read anything into the example other than it's easier to deconstruct the process if all the relevant code is in one block.

Summary

This article covered a number of topics, many of which you should devote some additional study time before using the supplied code in an application. Thankfully, Angular is doing all the heavy lifting and there are abundant sources of information on each of the Angular features covered in this article. In particular, I recommend reading up on *ViewContainerRef* and *ComponentFactoryResolver*. Then, modify and experiment with the code. Change the component order in the JSON file and observe the result. Try adding a couple more components to the demo. Then, you are ready to rock and roll in an actual application!

Components are not the only thing that can be lazy-loaded and dynamically instantiated in Angular. We can also lazy-load and runtime-instantiate Typescript libraries. This adds an additional level of sophistication and customization to highly dynamic applications. Imagine, for example, one or more components that employ simple AI's to tailor their interactions with users. We can not only specify the layout of components in data, we could even load different AI's that are trained/optimized for specific user experiences in addition to the components. If there is sufficient interest, I will follow up with another article on this topic.

I hope you found something of value in this article and the associated code. Good luck with your Angular efforts!

EnterpriseNG is coming November 4th & 5th, 2021.

Come hear top community speakers, experts, leaders, and the Angular team present for 2 stacked days on everything you need to make the most of Angular in your enterprise applications.
Topics will be focused on the following four areas:

- Monorepos
- Micro frontends
- Performance & Scalability
- Maintainability & Quality

Learn more here >> https://enterprise.ng-conf.org/

Angular Typescript Dynamic Components