Standalone Components in Angular

Leave a Comment / 10 minutes of reading / March 9, 2024

← Angular Tutorial Angular Components →

Standalone components in Angular (SAC) are a new type of component that Angular released in Angular 14. These components do not require Angular Modules (NgModule) and they self-manage their dependencies. They are easier to build, can be lazy loaded via Angular router, easily tree-shakable, and reduce the final bundle size. Not only components, but we can also create standalone pipes and standalone directives. In this article, we will learn what are Standalone components, how to create them, and learn their benefits with examples.

Table of Contents

Standalone Components

The Angular Components are the main building block of an Angular application. They contain the data and user interaction logic that defines how the View looks and behaves.

Until Angular 14, every Component we create needs to be registered with an Angular Module (or NgModule). The Angular Module organizes the related components, directives, pipes, and services and arranges them into cohesive blocks of functionality. Each Angular Module focuses on providing a specific functionality or a feature. Whenever we want to use the components from another NgModule, we must import that NgModule into our current module.

With the introduction of standalone components, Angular now provides a way to create & use angular components, without using the NgModule. We can also create standalone directives & pipes similarly. These components are self-contained and manage their dependencies themselves.

We can import and use these components anywhere in the application. Angular routers can also be set up to load them. We can also bootstrap our application from the Standalone component instead of a root module.

We can combine standalone components with NgModule-based components in the same application. Both will co-exist peacefully. It is also possible to build an entire

application using only Standalone Components eliminating the NgModule.

Creating Standalone Components

First, **Upgrade your Angular Project to Angular 15.2**.

The standalone components are created in the same way we create the regular angular components. All you need to mark `standalone: true` in their component metadata. You also need to `import` the required dependencies via `imports` array.

The following is an example of the standalone component.

```
1
2  //hello.component.ts
3
4  import { Component } from "@angular/core";
5
6  @Component({
7    selector: "app-hello",
8    template: `Hello World`,
9    standalone: true,
10   imports:[CommonModule]
11 })
12 export class HelloComponent {
13 }
14
```

Note that `standalone:true` metadata property. This is what makes the component standalone component.

If the component requires any dependencies, you need to import them into the `imports` array.

```

imports:[CommonModule]

```

We will discuss more about importing dependencies later.

Angular 17 & above

From Angular 17 and above, all new projects default to using the standalone API. Angular also recommends using standalone API, over Module-based API for all future projects.

Hence, if you create a new Angular project using `ng new`, it will create an Angular project using the Standalone API. Also, `ng g c` uses the Standalone API.

If you want to create a Module-based Angular project & Components use the flag `--standalone=false`, while creating a new project or component.

```

ng new <project-name>  --standalone=false

```

Managing Dependencies in Standalone Components

In the older module-based API, we used to declare the Component in a NgModule. NgModule would import the necessary dependencies via its imports array. All the Components, Directives, and Pipes that are part of the NgModule would automatically get to use them.

But, in the Standalone API, there is no NgModule. Hence whatever the component needs, we import it into the component itself. Hence use the import metadata to import any dependencies this component requires.

```
1
2  imports:[CommonModule]
3
```

By importing `CommonModule`, we can use the ngFor, DecimalPipe, ngClass, etc in `HelloComponent`. Alternatively, we can also import only the necessary methods instead of importing the entire `CommonModule`.

For Example, the code below only imports the ngFor & ngClass.

```
1
2 imports:[ ngFor, ngClass ]
3
```

Using Standalone Components

Angular has designed standalone components in such a way that they are compatible with module-based Components. You can use both of them in a single project without any issues.

There are three scenarios that you need to consider

1. Using Standalone Components in Module Based Components
2. Using Module-Based Components in Standalone Components
3. Standalone Components in Standalone components

Standalone Components in Module-Based Components

Create a new Module-based Project. Add the Standalone `HelloComponent`. You can view the source code here.

```
1
2  //hello.component.ts
3
4  import { Component } from "@angular/core";
5
6  @Component({
7    selector: "app-hello",
8    template: `Hello World`,
9    standalone: true,
10   imports:[CommonModule]
11 })
12 export class HelloComponent {
13 }
14
```

Now, we would like to display `HelloComponent` in our `AppComponent` (which is a module-based root component).

The AppComponent is not a standalone component. It is part of the AppModule. Hence to use the HelloComponent in AppComponent, we need to import it into AppModule. To do that import the HelloComponent to the imports metadata array.

```
1
2  //app.module.ts
3
4  import { NgModule } from '@angular/core';
5  import { BrowserModule } from '@angular/platform-browser';
6  import { FormsModule } from '@angular/forms';
7
8  import { AppComponent } from './app.component';
9  import { HelloComponent } from './hello.component';
10
11 @NgModule({
12   imports: [BrowserModule, FormsModule, HelloComponent],
13   declarations: [AppComponent],
14   bootstrap: [AppComponent],
15 })
16 export class AppModule {}
17
```

Now, open the app.component.ts and render the HelloComponent.

```
//<span style="background-color: initial; font-family: inherit; font-size: inherit;">app.comp

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <h1>Hello from {{ name }}!</h1>

    <app-hello></app-hello>
    <br><br><br>

  `,
})
export class AppComponent {
  name = 'Angular';
}
```

That's it.

You can view the Source Code from here.

Module-Based Components in Standalone Components

Create a new Angular Project and add a new module-based component

Test/TestComponent (Source Code)

```
//test.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'app-test',
  template: `<br>Hello World from Test`,
})
export class TestComponent {}
```

Create `TestModule` and declare `TestComponent` in it. Also, **remember to export it** in the exports metadata else you won't be able to use it in other modules or Standalone Components.

```
1
2  //test.module.ts
3
4  import { NgModule } from '@angular/core';
5  import { TestComponent } from './test.component';
6
7  @NgModule({
8    imports: [],
9    declarations: [TestComponent],
10   exports: [TestComponent],
11   bootstrap: [],
12 })
13 export class TestModule {}
14
```

Create Standalone `HelloComponent`. To use the `TestComponent` in `HelloComponent`, we need to Import the module it belongs i.e. `TestModule`. Once we import the module, we can use all exported components from the `TestModule` in this component.

```
1
2  //hello.componet.ts
3
4  import { CommonModule, NgClass, NgFor } from '@angular/common';
5  import { Component } from '@angular/core';
6  import { TestModule } from './test/test.module';
7
8  @Component({
9    selector: 'app-hello',
10   template: `Hello World
11         <app-test></app-test>
12   `,
13   standalone: true,
14   imports: [CommonModule, TestModule],
15   //imports: [NgFor, NgClass],
16 })
17 export class HelloComponent {}
18
```

```
//<span style="background-color: initial; font-family: inherit; font-size: inherit;">app.com

import { Component } from '@angular/core';
import 'zone.js';

@Component({
  selector: 'app-root',
  template: `
    <h1>Hello from {{ name }}!</h1>

    <app-hello></app-hello>
    <br><br><br>

    <a target="_blank" href="https://www.tektutorialshub.com/angular-tutorial/">
      Learn more about Standalone Components
    </a>

  `,
})
export class AppComponent {
  name = 'Angular';
}
```

```
//app.module.ts

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { HelloComponent } from './hello.component';

@NgModule({
  imports: [BrowserModule, FormsModule, HelloComponent],
  declarations: [AppComponent],
  exports: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

You can refer to the source code.

Standalone Components in Standalone components

To use Standalone components in another stand-alone component, import the component in the imports array of the standalone component.

For Example, create a Standalone `TestComponent` (Source code).

```
1
2  //test.component.ts
3
4  import { Component } from '@angular/core';
5
6  @Component({
7    selector: 'app-test',
8    standalone:true,
9    template: `<br>Hello World from Test`,
10 })
11 export class TestComponent {}
12
```

To use it in `HelloComponent`, add it to the Imports metadata.

```
1
2  //hello.component.ts
3
4  import { CommonModule, NgClass, NgFor } from '@angular/common';
5  import { Component } from '@angular/core';
6  import { TestComponent } from './test.component';
7
```

```
@Component({
  selector: 'app-hello',
  template: `Hello World
        <app-test></app-test>
  `,
  standalone: true,
  imports: [CommonModule, TestComponent],
})
export class HelloComponent {}

```

Source code

Advantages of Standalone Components

Now, we have learned how to create Standalone Components, Let us see some of its benefits.

Easy to Create, No need to learn anything about Modules

Angular Modules are a great feature. It allowed us to group related things. But it had a steep learning curve associated with it. By using standalone components, we can build an Angular application without creating an Angular Module.

Lazy Loading

Angular Modules can be lazy loaded delivering faster initial load time for the app. We use the `loadChilden` method of the Angular Router to lazy load them when the user navigates to that route. Loading a Module would load everything contained in the Module, including components, directives, pipes & services, etc. There was no way, we could lazy load a component.

Lazy Loading Angular Modules requires a little bit of planning and needs to be implemented correctly to get the best benefit out of it.

Standalone components make implementing lazy loading a simple task. Just use the `loadComponent` method of the router to lazy load every component and you are done with it. It is much faster than module-based lazy loading.

Tree shakable

Angular Compiler does a very good job of removing unused codes (or Tree shaking). The new standalone Angular components make it lot more easier, as they manage all their dependencies. It is easier for the compiler to analyze and remove the unused code, resulting in a much smaller bundle size.

Bootstrap Application with Standalone Component

The Module-based applications bootstrap with an AppModule (known as Root Module). You will find the relevant code in `main.ts`. The `bootstrapModule` method is responsible for loading the `AppModule`. The `bootstrapModule` is available in the library `@angular/platform-browser-dynamic`.

```
1
2 import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
3
4 import { AppModule } from './app/app.module';
5
6 platformBrowserDynamic().bootstrapModule(AppModule)
7   .catch(err => console.error(err));
8
```

The component to bootstrap is declared in the `AppModule` under the `bootstrap` property. Hence when `AppModule` bootstraps, it will display the `AppComponent`.

```
1
2 @NgModule({
3   declarations: [
4     AppComponent
5   ],
```

```
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})

```

To bootstrap the application using the Standalone components, we have a new method `bootstrapApplication`. This method is available in the `@angular/platform-browser` library.

```

import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { appConfig } from './app/app.config';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));

```

We need to pass the root component to load as the first argument to the bootstrapApplication method. The Syntax of the method is shown below.

```

bootstrapApplication(rootComponent: Type<unknown>,
                     options?: ApplicationConfig): Promise<ApplicationRef>
```

```
4
```

The `ApplicationConfig` is the optional second parameter, which is an interface consists only one property `providers`. Using the providers we can pass the Angular Providers.

```
1
2 interface ApplicationConfig {
3   providers: Array<Provider | EnvironmentProviders>
4 }
5
```

Migrate Existing Project to Standalone API

The CLI tool `ng generate` command provides an easy way to Migrate an existing Angular Project to use Standalone Components. It will also migrate the directives and pipes to Standalone API. The Angular CLI takes care of most of the migration work, but there might be a need for a few minor changes from your end.

```
1
2 ng generate @angular/core:standalone
3
4 or
5 ng g @angular/core:standalone
6
```

The Migration process is very simple & straightforward. It consists of **three steps**.

- Step 1: Convert all components, directives, and pipes to standalone
- Step 2: Remove unused NgModule classes
- Step 3: Bootstrap the application using standalone APIs

Step 1 Migration command will look for every component, directive & pipe and does the following

1. Add the standalone flag.
2. Populates the imports array
3. Removes the component from the declaration array of ngModule

Step 2 will remove those NgModules from the project that are **safe to remove**.

A module is considered “safe to remove” if it:

1. Has no declaration, providers & bootstrap components.
2. The module does not have a class Members
3. Has no imports that reference a **ModuleWithProviders** or reference a module that cannot be removed

It is very likely, that it will not be able to remove all the modules. Wherever it fails to remove it will add a TODO comment. You can manually review them and take the necessary action.

Step 3 will remove `AppModule` and configure the application to bootstrap using the standalone API. It will modify the main.ts file and use the `bootstrapApplication` to configure the app to bootstrap using the `AppComponent`. The Migration step will autofill the `providers` array from the existing `AppModule`.

We have a detailed article on How to Migrate to Standalone Components in Angular.

Summary

Standalone components are a new feature of Angular, which makes it easier to build Angular applications. It eliminates the Angular modules & makes the application load faster by way of lazy loading.